



HOEPLI  
TECNICA  
PER LA SCUOLA

Paolo Camagni  
Riccardo Nikolassy

# TECNOLOGIE E PROGETTAZIONE DI SISTEMI INFORMATICI E DI TELECOMUNICAZIONI

Nuova Edizione **OPENSCHOOL**

3

Per l'articolazione INFORMATICA  
degli Istituti Tecnici  
settore Tecnologico

Edizione **OPENSCHOOL**

- |   |                |
|---|----------------|
| 1 | LIBRO DITESTO  |
| 2 | E-BOOK+        |
| 3 | RISORSE ONLINE |
| 4 | PIATTAFORMA    |

i  
LIBRO IN  
CHIARO



**HOEPLI**



**PAOLO CAMAGNI**

**RICCARDO NIKOLASSY**

# **Tecnologie e progettazione di sistemi informatici e di telecomunicazioni**

**Nuova Edizione OPENSCHOOL**

**Per l'articolazione Informatica  
degli Istituti Tecnici settore Tecnologico**

**VOLUME 3**



**EDITORE ULRICO HOEPLI MILANO**

**Copyright © Ulrico Hoepli Editore S.p.A. 2017**  
Via Hoepli 5, 20121 Milano (Italy)  
tel. +39 02 864871 – fax +39 02 8052886  
e-mail [hoepli@hoepli.it](mailto:hoepli@hoepli.it)

[www.hoepli.it](http://www.hoepli.it)



Tutti i diritti sono riservati a norma di legge  
e a norma delle convenzioni internazionali

# Presentazione

Completamente revisionato, il terzo volume di *Tecnologie e progettazione di sistemi informatici e di telecomunicazioni* recepisce le indicazioni ricevute dai docenti che hanno in uso la precedente edizione; in particolare il testo è stato arricchito di due nuovi argomenti in modo da offrire più alternative per realizzare percorsi didattici personalizzati:

- ▶ **JSON**, il nuovo formato standard, aperto, adatto a immagazzinare varie tipologie di informazioni e a scambiare queste informazioni tra applicazioni, sia **standalone** che nel **web**;
- ▶ lo sviluppo di applicazioni per dispositivi mobili con sistema operativo **Android**.

È stata anche effettuata una riorganizzazione dei contenuti più funzionale: le **servlet** e la **JSP** sono state collocate in due unità di apprendimento separate, integrando la parte teorica ma, soprattutto, arricchendo la **sezione laboratoriale** con nuove lezioni ed esempi completamente svolti.

Ogni lezione termina con una sezione **CLIL** che riporta talvolta una sintesi dei contenuti esposti nella lezione, talvolta approfondimenti o glossari in lingua inglese.

Il presente volume è strutturato secondo **4 sezioni tematiche**:

- ▶ la prima descrive i **sistemi distribuiti** a partire dalla loro storia ed evoluzione, con la presentazione dei diversi **modelli architetturali**;
- ▶ la seconda sezione, aggiunta in questa nuova edizione, propone la programmazione per dispositivi mobili **Android**, con un insieme di lezioni che permette di realizzare **app** anche complesse connesse a database remoti;
- ▶ la terza analizza i **socket** e la modalità di comunicazione con i **protocolli TCP/ UDP** utilizzando il **linguaggio Java** e **C**; in particolare una unità di apprendimento è dedicata alle applicazioni lato server in **Java (servlet e JSP)**;
- ▶ una quarta sezione propone **applicazioni lato server** utilizzando il **linguaggio PHP**, in particolar modo gli aspetti avanzati con l'utilizzo di **AJAX**, degli oggetti e delle **API di Google** in **JavaScript**.

A ogni argomento è stata riservata una sezione dedicata alle **attività di laboratorio** proponendo esercitazioni specifiche e **numerosi esempi** con difficoltà graduale in tutti i linguaggi di programmazione utilizzati nel volume: **C, Java e JavaScript, PHP e AJAX**.

## Metodologia e strumenti didattici

Le finalità e i contenuti dei diversi argomenti affrontati sono descritti dagli **obiettivi generali** e dalle indicazioni **In questa lezione impareremo**; alla fine di ogni lezione, per lo studente sono presenti **esercizi**, anche **interattivi**, di **valutazione delle conoscenze** e delle **competenze** raggiunte, suddivisi in **domande a risposta multipla, a completamento, esercizi con procedure guidate**.

## Caratteristiche della nuova edizione

La Nuova Edizione Openschool consente di:

- ▶ **scaricare gratuitamente il libro digitale arricchito (eBook+)** che permette in particolare di:



eseguire tutti gli **esercizi** a risposta chiusa in modo **interattivo**;



scaricare gli **approfondimenti teorici** e i **laboratori integrativi**;

- ▶ disporre di ulteriori esercitazioni online utilizzabili a discrezione del docente per classi virtuali gestibili attraverso la **piattaforma didattica**.

## Aspetti caratterizzanti

- Testo pienamente in linea con le recenti indicazioni ministeriali in merito alle nuove **caratteristiche tecniche e tecnologiche dei libri misti e digitali** e al loro stretto coordinamento con la **piattaforma didattica**.
- Totale **duttilità di utilizzo in funzione delle scelte didattiche o dotazioni tecnologiche**:
  - il libro cartaceo + DVD consente di svolgere lezioni complete e attività di laboratorio con l'apparato offline presente nel DVD;
  - l'eBook+, le risorse online e la piattaforma offrono il pieno supporto per una didattica multimediale, a discrezione delle scelte del docente.
- **Lezioni autoconclusive** ricche di esempi ed esercizi, adatte a essere svolte in una lezione o al massimo due.
- **Teoria ridotta al minimo per privilegiare l'aspetto pratico.**
- Ricchezza dei materiali didattici presenti nel testo e online.

## Materiali online

Sul sito [www.hoepliscuola.it](http://www.hoepliscuola.it) sono disponibili numerose risorse online. In particolare, per lo studente: **approfondimenti**, utili integrazioni del testo e un numero elevato di **esercizi** sia per il **recupero e il rinforzo** sia per l'**approfondimento** degli argomenti trattati. Per il docente, una sezione riservata presenta alcune **unità didattiche per l'approfondimento** delle tematiche affrontate e un insieme di **proposte di esercizi** e **progetti** per la verifica dei livelli di apprendimento degli studenti. Materiali ed esercizi possono essere usati anche per creare attività didattiche fruibili tramite la piattaforma didattica accessibile dal sito.

## DVD per lo studente

Il **DVD per lo studente** allegato al volume contiene i file di tutti i programmi presentati nel libro oltre ai pacchetti software utilizzati per realizzarli e collaudarli. Per meglio usufruire dei programmi sorgente sono stati predisposti due menu in modo da poterli richiamare direttamente e/o visualizzare il codice. Nella directory del **DVD** è presente il file **leggimi.pdf** con le indicazioni necessarie per installare e configurare tutto il contenuto sulla propria workstation.



Il primo menu presenta gli esercizi descritti nelle UdA 1 e UdA 6

Il secondo menu presenta gli esercizi descritti nelle UdA 4 e UdA 5



Selezionando il link corrispondente ci viene proposta la relativa pagina con l'elenco degli esercizi



# Struttura dell'opera

The screenshot shows the first page of a lesson titled "Dispositivi e reti mobili". It includes a navigation bar with "UNITÀ DI APPRENDIMENTO" and "LEZIONE 1". The main content area has a title "Dispositivi e reti mobili" and a subtitle "In questa lezione impareremo...". It lists learning objectives: "I tipi di reti mobili", "il dispositivo mobile", and "il software per dispositivi mobili". Below this are sections for "Conoscenze", "Competenze", and "Abilità", each with a list of specific learning points. A sidebar on the right contains a "Sintesi/riassunto tra lezioni precedenti" section with a "ZOOM SU..." heading and a "Piccole sezioni di approfondimento" section.

This screenshot shows a "Attenzione" (Attention) section with the following text:  
"Questo articolo illustra come un utente o una persona che non è un esperto possa interagire con il sistema che sta progettando e, quindi, chi non sono degli utenti finali, ma non è altro che la tecnologia che si basa sullo studio di un sistema".  
Below this is a "Caso d'uso (use case)" section with a small icon and some descriptive text.  
The bottom part of the page contains a "DEFINIZIONI" section with the text: "Spiegazione delle proprietà essenziali dei principali termini, funzioni e concetti trattati nel testo".

## DEFINIZIONI

Spiegazione delle proprietà essenziali dei principali termini, funzioni e concetti trattati nel testo

## OSSERVAZIONI

Un aiuto per comprendere e approfondire

This screenshot shows a "Prova adesso!" (Try it now!) section with a "Scenario" titled "Inizio esercizio". It includes a list of steps: "1. Inizia l'esercizio", "2. Crea un account", "3. Compra un prodotto", "4. Invia un messaggio", and "5. Conferma l'ordine". Below this is a "Scenario" titled "Inizio esercizio" with the text: "Ampio spazio per scrivere una descrizione operativa tra uno o più effetti e il singolare e applicativo".

## PROVA ADESSO!

Per mettere in pratica, in itinere, quanto appreso nella lezione

This screenshot shows the second laboratory exercise section with the title "Esercitazioni di Laboratorio 2" and the sub-section "JAVA SOCKET: REALIZZAZIONE DI UN SERVER TCP". It includes a "Realizzazione di un server" section with a note about port 22091 and a screenshot of a terminal window showing socket communication. Below this is a "Java Server" section with a note about port 22090 and a screenshot of a terminal window showing socket communication. At the bottom, there is a "Lavoro finale" section with a note about port 22092 and a screenshot of a terminal window showing socket communication.

## LABORATORI

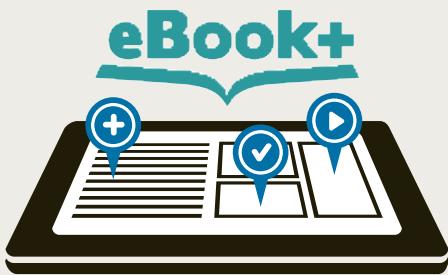
Per il rafforzamento dei concetti assimilati, attraverso esercitazioni operative

This screenshot shows a "Verifichiamo le conoscenze" section with a "Risposte sequenziali e parallele" table. It includes a "Problemi" section with a note about defining a class "Cognitivo" and a "Soluzione" section with a note about defining a class "Cognitivo". Below this is a "Verifichiamo le conoscenze" section with a "Risposte sequenziali e parallele" table. It includes a "Problemi" section with a note about defining a class "Cognitivo" and a "Soluzione" section with a note about defining a class "Cognitivo".

## ESERCIZI

Ampia sezione di esercizi per la verifica delle conoscenze e delle competenze

This screenshot shows a "Verifichiamo le conoscenze" section with a "Risposte sequenziali e parallele" table. It includes a "Problemi" section with a note about defining a class "Cognitivo" and a "Soluzione" section with a note about defining a class "Cognitivo". Below this is a "Verifichiamo le conoscenze" section with a "Risposte sequenziali e parallele" table. It includes a "Problemi" section with a note about defining a class "Cognitivo" and a "Soluzione" section with a note about defining a class "Cognitivo".



L'eBook+ riproduce le pagine del libro di testo in versione digitale e interattiva. È utilizzabile su tablet, LIM e computer e consente di annotare, sottolineare ed evidenziare il testo, salvando il proprio lavoro per poterlo consultare e sincronizzare sui diversi dispositivi. Apposite icone attivano i contributi digitali integrativi.

**APPROFONDIMENTI**  
Contenuti, lezioni e unità integrative; esercizi per il recupero e l'approfondimento

**VIDEO**  
Video tutorial per esemplificare azioni e procedimenti

**ESERCIZI**  
Esercizi interattivi di varia tipologia con funzione di autocorrezione

**IMMAGINI  
E GALLERIE  
DI IMMAGINI**  
Per esemplificare e rappresentare visivamente i contenuti

**LINK**  
Rimandi interni al volume per navigare agevolmente tra i contenuti

# L'OFFERTA DIDATTICA HOEPLI

L'edizione **Openschool** Hoepli offre a docenti e studenti tutte le potenzialità di Openschool Network (ON), il nuovo sistema integrato di contenuti e servizi per l'apprendimento.

## Edizione **OPENSCHOOL**



### LIBRO DI TESTO



Il libro di testo è l'**elemento cardine** dell'offerta formativa, uno strumento didattico **agile** e **completo**, utilizzabile **autonomamente** o in combinazione con il ricco **corredo digitale** offline e online. Secondo le più recenti indicazioni ministeriali, volume cartaceo e apparati digitali sono **integrità** in un unico percorso didattico. Le espansioni accessibili attraverso l'eBook+ e i materiali integrativi disponibili nel sito dell'editore sono puntualmente richiamati nel testo tramite apposite icone.

### eBOOK+



L'**eBook+** è la versione digitale e interattiva del libro di testo, utilizzabile su **tablet**, **LIM** e **computer**. Aiuta a comprendere e ad approfondire i contenuti, rendendo l'apprendimento più attivo e coinvolgente. Consente di leggere, annotare, sottolineare, effettuare ricerche e accedere direttamente alle numerose **risorse digitali integrative**. ➔ Scaricare l'eBook+ è molto **semplice**. È sufficiente seguire le istruzioni riportate nell'ultima pagina di questo volume.

### RISORSE ONLINE



Il sito della casa editrice offre una ricca dotazione di **risorse digitali** per l'approfondimento e l'aggiornamento. Nella pagina web dedicata al testo è disponibile **MyBookBox**, il contenitore virtuale che raccolge i materiali integrativi che accompagnano l'opera. ➔ Per accedere ai materiali è sufficiente registrarsi al sito **www.hoepliscuola.it** e inserire il codice coupon che si trova nella terza pagina di copertina. Per il docente nel sito sono previste ulteriori risorse didattiche dedicate.

### PIATTAFORMA DIDATTICA



La **piattaforma didattica** è un ambiente digitale che può essere utilizzato in modo duttile, a misura delle esigenze della classe e degli studenti. Permette in particolare di **condividere contenuti** ed **esercizi** e di partecipare a **classi virtuali**. Ogni attività svolta viene salvata sul **cloud** e rimane sempre disponibile e aggiornata. La piattaforma consente inoltre di consultare la versione online degli eBook+ presenti nella propria libreria. ➔ È possibile accedere alla piattaforma attraverso il sito **www.hoepliscuola.it**.



# Indice

## UNITÀ DI APPRENDIMENTO 1

### Architettura di rete e formati per lo scambio dei dati

#### L1 I sistemi distribuiti

I sistemi distribuiti .....	2
Classificazione dei sistemi distribuiti .....	3
Benefici della distribuzione .....	4
Svantaggi legati alla distribuzione .....	6
 .....	7

<b>Verifichiamo le conoscenze .....</b>	8
---	---

#### L2 Evoluzione dei sistemi distribuiti e dei modelli architetturali

Premessa .....	9
Architetture distribuite hardware: dalle SISD al cluster di PC .....	10
Architetture distribuite software: dai terminali remoti ai sistemi completamente distribuiti .....	15
Architettura a livelli .....	17
Conclusioni .....	18
 .....	18

<b>Verifichiamo le conoscenze .....</b>	19
---	----

#### L3 Il modello client-server

Il modello client-server .....	21
Distinzione tra server e client .....	22
Livelli e strati .....	24
 .....	27

<b>Verifichiamo le conoscenze .....</b>	28
---	----

#### L4 Le applicazioni di rete

Il modello ISO/OSI e le applicazioni .....	29
Applicazioni di rete .....	30
Scelta dell'architettura per l'applicazione di rete ...	32
Servizi offerti dallo strato di trasporto alle applicazioni .....	34
Conclusioni .....	36
 .....	36

<b>Verifichiamo le conoscenze .....</b>	37
---	----

## Esercitazioni di laboratorio

1 Il linguaggio XML .....	38
2 JSON .....	45

 3 Web server Apache .....	45
 4 Web server Apache in Linux .....	46
 5 Web server IIS su Windows .....	47

## AREA digitale

 Esercizi .....	
 Creiamo cluster HPC con ParallelKnoppix .....	
▶ Applicazioni P2P .....	
▶ Comparison chart TPC vs UDP .....	
▶ Browser e XML .....	
▶ Definizione di JSON dal suo creatore Douglas Crockford .....	
▶ Select dinamiche con PHP e JSON .....	
▶ Esercizi per il recupero .....	
▶ Esercizi per l'approfondimento .....	

## UNITÀ DI APPRENDIMENTO 2

### Android e i dispositivi mobili

#### L1 Dispositivi e reti mobili

Premessa .....	56
Reti mobili .....	56
Software per dispositivi mobili .....	60
Sistemi operativi per dispositivi mobili .....	61
Ambienti di sviluppo per dispositivi mobili .....	63
 .....	65
<b>Verifichiamo le conoscenze .....</b>	66

<b>L2 Android: un sistema operativo per applicazioni mobili</b>	
Android .....	67
La struttura di un'applicazione Android .....	69
Il ciclo di vita di una activity .....	71
Il file APK .....	72
 .....	73
<b>Verifichiamo le conoscenze .....</b>	74

## Esercitazioni di laboratorio

- 1 Android Studio: installazione e configurazione ..... 75
- 2 L'intefaccia grafica di Android Studio ..... 82
- 3 Utilizziamo i widget nelle app Android ..... 96
- 4 Un'app completa: la calcolatrice ..... 108
-  5 Utilizziamo i sensori
-  6 La connessione a database locali in Android

## AREA digitale

-  ▶ Esercizi
-  ▶ Orizzonte 2020
- ▶ I sistemi operativi Palm OS, Symbian e Linux
- ▶ Set che compongono Xcode
- ▶ Versioni di Android
- ▶ I diversi tipi di tocco su display touch
- ▶ Il layout degli elementi grafici
- ▶ Esercizi per il recupero
- ▶ Esercizi per l'approfondimento

## UNITÀ DI APPRENDIMENTO 3

### I socket e la comunicazione con i protocolli TCP/UDP

#### L1 I socket e i protocolli per la comunicazione di rete

Generalità .....	112
Le porte di comunicazione e i socket .....	114
 .....	119
<b>Verifichiamo le conoscenze.....</b>	120
<b>L2 La connessione tramite socket</b>	
Generalità .....	121

Famiglie e tipi di socket .....	122
Trasmissione multicast .....	125

 .....	127
---	-----

<b>Verifichiamo le conoscenze .....</b>	128
---	-----

## Esercitazioni di laboratorio

- 1 Java socket .....
- 2 Java socket: realizzazione di un server TCP .....
- 3 Realizzazione di un server multiplo in Java .....
- 4 Java socket: un'animazione client-server .....
- 5 Il protocollo UDP nel linguaggio JAVA .....
- 6 Applicazioni multicast in Java .....
- 7 Un esempio completo con le Java socket: "la chat" .....
- 8 I socket nel linguaggio C .....
- 9 Server TCP in C .....
- 10 Client TCP in C .....
- 11 Il protocollo UDP nel linguaggio C .....
- 12 Un esempio completo con i socket: asta online .....

## AREA digitale

-  ▶ Esercizi
-  ▶ Confronto ISO/OSI e Internet
- ▶ Connectionless e connection-oriented
- ▶ API – Application Programming Interface
- ▶ Un esempio di verifica
- ▶ Funzioni di conversione Big-Endian/ Little-Endian
- ▶ Abilitare telnet in Windows 8 e Windows 8.1
- ▶ Funzioni GETSOCKOPT() e SETSOCKOPT()

## UNITÀ DI APPRENDIMENTO 4

### Applicazioni lato server in Java: servlet

#### L1 Le servlet

Servlet e CGI .....	202
Struttura di un servlet .....	203

La classe HttpServlet .....	206
Ciclo di vita di una servlet .....	207
Output sul client .....	208
Deployment di un'applicazione web .....	209
Il Context XML descriptor o Deployment descriptor .....	211
Esecuzione di una servlet .....	213
Servlet concorrenti .....	213
Vantaggi e svantaggi delle servlet .....	214
 .....	214
<b>Verifichiamo le conoscenze</b> .....	215
<b>L2 JDBC: Java DataBase Connectivity</b>	
JDBC .....	216
Tipi di driver JDBC .....	217
Utilizzare JDBC standalone .....	219
Servlet con connessione a MySQL .....	222
Applicazione e servlet con connessione ad Access ..	223
 .....	225
<b>Verifichiamo le conoscenze</b> .....	226

## Esercitazioni di laboratorio

- 1 XAMPP e il server engine Tomcat .227
- 2 L'inizializzazione della servlet .....236
- 3 L'interazione tra client e servlet  
get/post con le servlet .....240
- 4 La permanenza dei dati  
con le servlet: i cookie .....246
- 5 La permanenza dei dati  
con le servlet: le sessioni .....250
- 6 JDBC e MySQL .....255
- 7 Servlet e database MDB  
con parametri .....260

## AREA *digitale*

-  **Esercizi**
-  Schema UML della gerarchia  
delle classi
-  Installazione della libreria  
UCanAccess
-  Installazione di Tomcat su Apache
-  Netbeans, Eclipse  
e il file web.xml
-  Metodi principali classe Cookie
-  Elenco metodi dell'oggetto  
HttpSession
-  Codifica del Session ID nei link
-  Configurazione manuale di JDBC
-  Registrazione database in Windows  
mediante ODBC
-  Installazione librerie UCanAccess

## UNITÀ DI APPRENDIMENTO 5

### Applicazioni lato server in Java: JSP

#### L1 JSP: Java Server Pages

Le Java Server Pages (JSP) .....	266
Tag in un pagina JSP .....	268
Tag scripting-oriented .....	268
XML-oriented tag o JSP Standard Actions .....	276



#### Verifichiamo le conoscenze

.....	278
-------	-----

#### L2 Java Server Pages e Java Bean

Java Bean .....	279
Uso di Java Bean .....	281
Configurazione dell'applicazione .....	283
Passaggio parametri al Bean .....	284
Conclusioni .....	288



#### Verifichiamo le conoscenze

.....	289
-------	-----

#### Verifichiamo le competenze

.....	290
-------	-----

## Esercitazioni di laboratorio

- 1 JSP: primi esempi di Java  
Server Pages .....
- 2 JSP: Java Server Pages con  
parametri da HTML .....
- 3 JSP e database MySQL .....
- 4 JSP e MDB .....
- 5 JSP Bean e database .....

## AREA *digitale*

-  Esercizi

-  Esempio di codice Java generato da  
Tomcat: la classe oggi\_jsp.java

## UNITÀ DI APPRENDIMENTO 6

### Applicazioni lato server in PHP e AJAX

#### L1 I file e l'upload in PHP

L'apertura di un file .....	312
Lettura e scrittura in un file di testo .....	313
L'array associativo \$_FILES .....	314

	.....	316
<b>Verifichiamo le conoscenze</b>	.....	317
<b>Verifichiamo le competenze</b>	.....	318
<b>L2 Gli oggetti in PHP</b>		
Il paradigma a oggetti in PHP .....	319	
Il costruttore .....	322	
L'ereditarietà .....	324	
	.....	324
<b>Verifichiamo le competenze</b>	.....	325
<b>L3 La connessione ai database object oriented</b>		
Premessa .....	326	
Connessione a MySQL con MySQLi .....	327	
	.....	329
<b>Verifichiamo le competenze</b>	.....	330
<b>L4 Le API di Google</b>		
La geolocalizzazione .....	331	
Usare le API di Google Maps .....	333	
Associare eventi agli oggetti Google Maps .....	337	
Il calcolo del percorso .....	341	
Lo Street View panorama .....	344	
	.....	346
<b>Verifichiamo le competenze</b>	.....	347

## Esercitazioni di laboratorio



- 1 Installazione di EasyPHP .....
- 2 Comunicazione client-server con AJAX .....
- 3 Connessione FTP con uno script PHP .....
- 4 Invia una mail con PHP connesso a MySQL .....
- 5 Creare file pdf con PHP .....
- 6 Creare file di Excel e Word con PHP .....
- 7 Le API di Google: interazione tra javascript, AJAX e PHP .....
- 8 Invia un file con i socket in PHP .....



## AREA digitale



- ▶ Esercizi



- ▶ Why in the World Would PHP do this?
- ▶ Come ottenere la API key
- ▶ Vantaggi offerti dal formato PDF

**Come utilizzare il coupon per scaricare  
la versione digitale del libro (eBook+) e i  
contenuti digitali integrativi (risorse online)..... 380**

# 1

# Architettura di rete e formati per lo scambio dei dati

L1 I sistemi distribuiti

L2 Evoluzione dei sistemi distribuiti e dei modelli architetturali

L3 Il modello client-server

L4 Le applicazioni di rete

## Esercitazioni di laboratorio

1 Il linguaggio XML; 2 JSON; 3 Web server Apache; 4 Web server Apache in Linux; 5 Web server IIS su Windows

### Conoscenze

- Conoscere gli stili architetturali fondamentali per sistemi distribuiti
- Comprendere il modello client-server
- Avere chiaro il concetto di elaborazione distribuita
- Conoscere il concetto di middleware
- Le caratteristiche del modello client-server
- L'evoluzione del modello client-server
- Avere chiaro il concetto di applicazione di rete

### Competenze

- Saper riconoscere le diverse tipologie di sistemi distribuiti
- Saper classificare le architetture distribuite
- Individuare i benefici della distribuzione
- Confrontare la distribuzione con l'elaborazione concentrata
- Individuare le diverse applicazioni distribuite
- Saper classificare le applicazioni di rete

### Abilità

- Scegliere i protocolli per le applicazioni di rete
- Definire strutture dati in XML
- Definire strutture dati in JSON
- Saper installare e configurare IIS
- Saper installare e configurare Apache e MySQL

## AREA *digitale*



Esercizi



- ▶ Creiamo cluster HPC con ParallelKnoppix
- ▶ Applicazioni P2P
- ▶ Comparison chart TPC vs UDP
- ▶ Browser e XML
- ▶ Definizione di JSON dal suo creatore Douglas Crockford
- ▶ Select dinamiche con PHP e JSON
- ▶ Esercizi per il recupero e l'approfondimento



Esempi proposti

Consulta il DVD in allegato al volume



Soluzioni

Puoi scaricare il file anche da hoepliScuola.it

# I sistemi distribuiti

In questa lezione impareremo...

- il concetto di elaborazione distribuita
- i benefici della distribuzione
- gli svantaggi rispetto alla elaborazione concentrata

## ■ I sistemi distribuiti

A metà degli anni Quaranta inizia l'era dei *calcolatori elettronici* moderni e a partire dalla metà degli anni Ottanta due sviluppi nella tecnologia hanno iniziato a cambiare la situazione: l'avvento dei **microprocessori** e la realizzazione di **reti di elaborazione** ottenute connettendo singoli computer per realizzare sistemi di calcolo complessi e potenti.

Le architetture dei **sistemi informativi** si sono sviluppate ed evolute nel corso degli anni successivi passando da schemi **centralizzati** a modelli **distribuiti** più aderenti alla necessità di decentralizzazione e cooperazione delle moderne organizzazioni.

Nei **sistemi centralizzati** le applicazioni girano in un singolo processore, o comunque su un solo **host**, che costituisce l'unico componente autonomo nel sistema che è condiviso da vari utenti e tutte le risorse del componente sono sempre accessibili.

Nei **sistemi distribuiti** le applicazioni sono costituite da più processi, cooperanti, eseguiti in parallelo su un insieme di unità di elaborazione autonome: sono quindi sistemi ottenuti dall'aggregazione di singole **CPU**, **sistemi di memorizzazione** e **periferiche**.

Un sistema informatico si dice **distribuito** se almeno una delle seguenti due condizioni è verificata:

- **elaborazione distribuita**: le applicazioni risiedono su più **host** che collaborano tra loro;
- **base di dati distribuita**: il patrimonio informativo è ospitato su più **host**.

Numerose sono state le definizioni di sistema distribuito date a partire dagli anni Settanta ma nessuna di queste è pienamente soddisfacente: ognuna risente delle tecnologie e della visione prevalente (hardware o software) all'epoca in cui è stata formulata.

Riportiamo le più note.



### SISTEMA DISTRIBUITO

1. Un sistema distribuito consiste di un insieme di calcolatori indipendenti che appaiono all'utente del sistema come un singolo calcolatore (**Tanenbaum**).
2. È un **sistema** in cui i componenti hardware o software posizionati in calcolatori collegati in rete comunicano e coordinano le proprie azioni solo tramite lo scambio di messaggi (**Couloris & Dollimore**).
3. È un sistema in cui il fallimento di un calcolatore di cui nemmeno conosci l'esistenza può rendere inutilizzabile il tuo calcolatore (**Lamport**).

In questa sezione adotteremo la seguente definizione più astratta.



### SISTEMA DISTRIBUITO

Un **sistema distribuito** è costituito da un insieme di applicazioni logicamente indipendenti che collaborano per il perseguimento di obiettivi comuni attraverso una infrastruttura di comunicazione hardware e software.

Su ogni componente del **sistema distribuito** viene eseguito un programma che può essere differente sia per il compito che svolge che per il ruolo di elaborazione che, grazie a esso, assume nella rete. Alle **applicazioni** vengono dati quindi nomi diversi proprio in funzione dei diversi ruoli:

- **cliente (client)**: una applicazione assume il ruolo di **cliente** quando è utilizzatore di servizi messi a disposizione da altre applicazioni;
- **servente (server)**: una applicazione assume il ruolo di **servente** quando è fornitore di servizi usati da altre applicazioni;
- **attore (actor)**: una applicazione assume il ruolo di **attore** quando assume in diverse situazioni nel contesto del sistema sia il ruolo di **cliente** che quello di **servente**.

Nella lezione 3 affronteremo lo studio del modello architettonale che prende proprio il nome dai “ruoli dei suoi componenti”, cioè modello **client-server**.

Esiste una “sottile” differenza tra sistemi distribuiti e sistemi paralleli:

- un **sistema distribuito** è un insieme di computer indipendenti che **comunicano** e **possono cooperare** per risolvere problemi;
- un **sistema parallelo** è un insieme di elementi di elaborazione che **comunicano** e **cooperano** per risolvere velocemente problemi di grandi dimensioni.

## ■ Classificazione dei sistemi distribuiti

È possibile classificare i **sistemi distribuiti** in tre grandi famiglie:

- **sistemi di calcolo** distribuiti;
- **sistemi informativi** distribuiti;
- **sistemi distribuiti pervasivi**.

I **sistemi di calcolo** distribuiti sono generalmente configurati per il calcolo ad alte prestazioni; nel seguito ne descriveremo due configurazioni:

- **cluster computing**;
- **grid computing**.

Tra i **sistemi informativi** distribuiti, oltre che al **web** che risulta essere il più grande sistema distribuito, le nuove tecnologie **mobile** hanno fatto da volano nell'evoluzione dei sistemi informativi tradizionali.



◀ I legacy system (**sistemi legacy**) sono i sistemi presenti in azienda, realizzati su architetture basate su mainframe al quale si collegano terminali poco sofisticati, generalmente con interfaccia a caratteri. Si tratta di sistemi obsoleti, ereditati dalle prime fasi di informatizzazione dell'aziende che spesso rivestono ruoli critici e hanno dimensioni notevoli. Sebbene abbiano subito una lunga storia di interventi di manutenzione, non sono mai stati rimpiazzati: hanno la caratteristica di... **funzionare (!)** ▶

nali, che integrando ◀ **sistemi legacy** ▶ con nuove tecnologie di comunicazione, hanno generato i moderni sistemi informativi.

I **sistemi distribuiti pervasivi** sono una nuova generazione di sistemi che hanno tipicamente connessioni di rete wireless e che generalmente sono sottoparti di sistemi più grandi; tra di essi rientrano i **sistemi domestici**, le **personal area network (PAN)**, le **wearable computing** e le **reti di sensori**.

## ■ Benefici della distribuzione

Tutti i grandi sistemi informatici sono oggi dei **sistemi distribuiti** e presentano come ogni sistema un insieme di vantaggi e di svantaggi che illustriamo sommariamente, senza specificare se sono dovuti all'hardware oppure al software.

### L'affidabilità

Il principale vantaggio dei **sistemi distribuiti** è l'**affidabilità**: grazie alla sua ridondanza intrinseca un **SD** è in grado di “sopravvivere” a un guasto di un suo componente; è comunque necessario prevenire queste situazioni e predisporre strumenti in grado di intervenire automaticamente al verificarsi di situazioni indesiderate, per esempio con l'implementazione di algoritmi che permettano alle entità non guaste di “sostituire” quella danneggiata e continuare l'elaborazione.

Non sempre però questa strada è percorribile: la difficoltà di realizzazione cresce con l'aumentare della autonomia delle entità.

### Integrazione

Un secondo vantaggio non meno importante del precedente è la capacità che ha un sistema distribuito di integrare componenti spesso eterogenei tra loro, sia per **tipologia hardware** (da PC a mainframe, da smartphone a tablet), sia per **sistema operativo**.

Ogni componente deve poter interfacciarsi allo stesso modo con il **sottosistema di comunicazione** del **sistema distribuito** ed è proprio l'interfaccia che permette di rendere “trasparenti” i componenti dello “strato inferiore” del sistema all'intera rete.

Di fondamentale importanza è la possibilità di connettere dispositivi di nuova generazione con **legacy system** che, di fatto, utilizzano tecnologie in altro modo incompatibili. È noto a tutti che con l'utilizzo della **tecnologia ethernet** oppure con la **piattaforma Web** sistemi operativi e architetture hardware diverse possono interconnettersi e scambiarsi informazioni.

Sono stati definiti appositi linguaggi, come **XML (eXtensible Markup Language)** e notazioni **JSON (JavaScript Object Notation)** per favorire lo scambio di informazioni nel Web e permettere un'agevole ed efficiente pubblicazione di dati complessi: li descriveremo entrambi in seguito.

### Trasparenza

Come trasparenza si intende il concetto di “vedere” il **sistema distribuito** non come un insieme di componenti ma come un unico sistema di elaborazione: l'utente non deve accorgersi che sta interagendo con un **sistema distribuito** ma deve avere la percezione di utilizzare un **singolo elaboratore**, senza percepire eventuali mutamenti che si verificano nel tempo nel sistema.

L'**ANSA** nell'**ISO 10746, Reference Model of Open Distributed Processing**, identifica otto forme di trasparenza:

- ▶ di **accesso** (*access transparency*): si devono nascondere le differenze nella rappresentazione dei dati e nelle modalità di accesso alle risorse, in modo da permettere di accedere a risorse locali e remote con le stesse operazioni, in modo unico e uniforme;
- ▶ di **locazione** (*location transparency*): si deve nascondere dove è localizzata una risorsa e permettere di accedere a essa senza conoscerne la locazione (per esempio, l'**URL** nasconde l'indirizzo **IP**);
- ▶ di **concorrenza** (*concurrency transparency*): si permette ai processi di operare in maniera concorrente, cioè facendo in modo che l'accesso concorrente a una risorsa condivisa la lasci sempre in uno stato consistente, implementando per esempio meccanismi di **locking**;
- ▶ di **replicazione** (*replication transparency*): le operazioni di duplicazione vengono effettuate all'insaputa degli utenti mantenendo gli stessi nomi per ogni risorsa, in modo da avere copie di risorse che aumentano l'affidabilità e le prestazioni del sistema;
- ▶ ai **guasti** (*failure transparency*): viene mascherato il guasto e il recovery di una risorsa;
- ▶ alla **migrazione** (*mobility transparency*): si nasconde l'eventuale spostamento (logico o fisico) di una risorsa senza interferire sulla sua modalità di accesso, cioè senza influenzare le operazioni degli utenti che la riguardano;
- ▶ alle **prestazioni** (*performance transparency*): vengono nascoste le operazioni necessarie per reconfigurare il sistema al variare del carico per migliorarne le prestazioni;
- ▶ di **scalabilità** (*scaling transparency*): il sistema viene espanso senza interrompere o modificare il funzionamento.

I più importanti sono i primi due, cioè la trasparenza di **accesso** e di **locazione**.

## Economicità

Generalmente i **sistemi distribuiti** offrono spesso un miglior rapporto prezzo/qualità dei sistemi centralizzati basati su mainframe: una rete di PC connessi ha un prezzo di alcuni ordini di grandezza inferiore rispetto a quello di un mainframe e con le tecnologie odierne la capacità computazionale è paragonabile. Inoltre, come già abbiamo detto, introduce la possibilità di connettere con un costo molto basso i **sistemi legacy** in modo da non dover abbandonare una tecnologia su cui è stato realizzato un certo investimento e poter convivere con la tecnologia più recente.

## Apertura

Con la definizione di protocolli standard si favorisce l'apertura all'hardware e software di fornitori diversi in modo da avere:

- ▶ **interoperabilità**: implementazioni diverse su elaboratori di diverso tipo possono coesistere per comporre un unico sistema;
- ▶ **portabilità**: una applicazione sviluppata su un sistema operativo può funzionare su di un altro presentando la medesima interfaccia all'utente in modo da non modificare l'operatività;
- ▶ **ampliabilità**: è relativamente semplice "far crescere" il sistema aggiungendo sia componenti hardware che software.

## Connattività e collaborazione

Nei **sistemi distribuiti** la possibilità di condividere risorse hardware e software comporta vantaggi economici. Per esempio è possibile condividere apparecchiature speciali di costo elevato, quali stampanti particolari, plotter, oppure sistemi di storage e di recovery. Lo scambio e la condivisione di informazioni arricchisce ogni utilizzatore: per esempio vengono creati groupware con specifici interessi.

## Prestazioni e scalabilità

Un **sistema distribuito** può essere composto semplicemente da due workstation e un file server oppure da una **LAN** che può contenere anche molti server, migliaia di workstation, molte stampanti ecc. La crescita di un **sistema distribuito** con l'aggiunta di nuove risorse e, quindi, di nuovi servizi, fornisce a tutti i suoi componenti un miglioramento delle prestazioni e permette di sostenere l'aumento del carico di richieste: questa possibilità prende il nome di **scalabilità orizzontale**. Con la **scalabilità** si superano i problemi visti nel caso di un sistema centralizzato dove spesso alcune risorse condivise sono disponibili in misura limitata: nei **SD** concettualmente non esiste una limitazione nel numero di risorse disponibili, basta che il sistema abbia la “possibilità di espandersi”.

## Tolleranza ai guasti

La possibilità di replicare risorse offre una certa garanzia di tolleranza ai guasti; la presenza di un componente guasto non deve pregiudicare il funzionamento del sistema ma, al limite, introdurre solo inefficienze in termini di tempo di risposta.

In questa situazione si parla di “guasto parziale”, cioè di un guasto che non influenza pesantemente il funzionamento del sistema.

Un primo approccio è quello di duplicare interamente le risorse, però questa è una soluzione spesso molto costosa. Quindi è opportuno optare per un sistema software di recupero, che sia in grado di riportare il sistema in uno stato consistente precedente e di trovare una alternativa al componente guasto.

## ■ Svantaggi legati alla distribuzione

I **sistemi distribuiti** hanno però anche una serie di svantaggi che le nuove tecnologie cercano di ridurre o eliminare in fase di progettazione dei nuovi dispositivi di rete. Descriviamo sinteticamente i principali problemi.

### Produzione di software

I programmatore del “secolo scorso” hanno dovuto modificare il proprio stile di programmazione e aggiornarsi con lo studio dei nuovi linguaggi e dei nuovi strumenti di sviluppo per poter realizzare applicazioni distribuite. Il passaggio è avvenuto in tre tappe:

- la definizione dello standard **TCP/IP** (e quindi della modalità di comunicazione tramite **socket**) è divenuta la base per lo sviluppo di applicazioni di rete;
- lo sviluppo di architetture **Web** e di tutti i linguaggi a esse connesse, sia lato **client** come **HTML**, **CSS**, **Javaseript**, sia lato **server**, come **CGI**, **servlet**, **JSP**, **PHP**, **ASP** ecc. che hanno permesso di far effettuare un salto di qualità alla produzione del software;
- il diffondersi del linguaggio **Java** che, grazie alla sua tecnologia a *macchina virtuale*, permette ai programmi di essere eseguiti su macchine anche completamente differenti tra loro.

### Complessità

Proprio per la struttura hardware i **sistemi distribuiti** sono più complessi di quelli centralizzati: richiedono strumenti per l'interconnessione degli host e tecniche per l'instradamento corretto dei messaggi e dei dati. È anche più complesso valutare le performance di un sistema distribuito piuttosto che di un unico elaboratore.

### Sicurezza

Con la connessione di più host tra loro si crea la possibilità di accedere a dati e risorse anche a chi non ne ha il diritto: nascono nuove problematiche connesse alla **sicurezza** che nel caso di sistemi centralizzati erano inesistenti. Infatti, nei vecchi sistemi per lo più bastava proteggere il sistema

dall'accesso fisico delle persone ai locali dove erano presenti i dispositivi da proteggere (hard disk e supporti di memorizzazione). Oggi l'accesso avviene via etere e via cavo e anche le trasmissioni sono soggette a rischio di intercettazione (**sniffing**) e quindi richiedono l'applicazione di appositi accorgimenti per tutelare tutti gli utenti e garantire sicurezza e riservatezza nei dati, sia memorizzati sui proprio computer personali, sia trasmessi per transazioni commerciali o semplicemente personali (**email**).

## Comunicazione

Il trasferimento a distanza delle informazioni richiede nuove tipologie di sistemi di telecomunicazione, sia **cablati** che **wireless**, e l'aumento esponenziale degli utenti fa sì che giornalmente aumenti la richiesta di bande trasmissive, anche per migliorare la qualità del servizio offerto e offrire nuove tipologie di applicazioni sempre più performanti (alta velocità, alta definizione, video streaming ecc.). La difficoltà maggiore sta nel fatto che si è sempre di fronte a nuove problematiche delle quali non si ha "ben chiaro" né il dominio di definizione ne l'ambito delle soluzioni: la mancanza di **prevedibilità** delle richieste genera situazioni "casuali" di carico e quindi di risposta del sistema che possono essere molto diverse anche a breve distanza di tempo. A volte l'utilizzo di tecniche "sperimentali" porta all'aumento di complessità in modo accidentale anche a causa del fatto che le tecniche di analisi e progettazione sono spesso basate su vecchi metodi di sviluppo di applicazioni monoprocesso, non adeguate alle situazioni di sistema distribuito.

A **distributed system** consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility.



	<b>Centralised System</b>	<b>Distributed System</b>
<b>Characteristics</b>	One component with non-autonomous parts Component shared by users all the time All resources accessible Software runs in a single process Single Point of control Single Point of failure	Multiple autonomous components Components are not shared by all users Resources may not be accessible Software runs in concurrent processes on different processors Multiple Points of control Multiple Points of failure

Certain common characteristics can be used to assess distributed systems : Resource Sharing, Openness, Concurrency, Scalability, Fault Tolerance, Transparency.

### Resource Sharing

Ability to use any hardware, software or data anywhere in the system.

Resource manager controls access, provides naming scheme and controls concurrency.

Resource sharing model (e.g. client/server or object-based) describing how resources are provided, they are used and provider and user interact with each other.

### Openness

Openness is concerned with extensions and improvements of distributed systems, detailed interfaces of components need to be published, new components have to be integrated with existing components. Differences in data representation of interface types on different processors (of different vendors) have to be resolved.

### Concurrency

Components in distributed systems are executed in

concurrent processes. Components access and update shared resources (e.g. variables, databases, device drivers). Integrity of the system may be violated if concurrent updates are not coordinated. Adaption of distributed systems to accomodate more users respond faster (this is the hard one).

Usually done by adding more and/or faster processors: components should not need to be changed when scale of a system increases. Design components to be scalable!

### Fault Tolerance

Hardware, software and networks fail! Distributed systems must maintain availability even at low levels of hardware/software/network reliability. Fault tolerance is achieved by recovery/redundancy.

**Transparency** has different dimensions that were identified by ANSA: Access, Location, Concurrency, Replication, Failure, Migration, Performance, Scaling.

## Verifichiamo le conoscenze



### 1. Risposta multipla

**1 In un sistema informatico distribuito almeno una delle seguenti condizioni è verificata:**

- a. elaborazione distribuita
- b. calcolatori distribuiti
- c. base di dati distribuita
- d. informazioni distribuite

**2 Quale delle seguenti affermazioni per i sistemi legacy sono false?**

- a. sono quelli presenti in azienda realizzati generalmente su architetture basate su mainframe
- b. a essi si collegano terminali poco sofisticati, generalmente con interfaccia a caratteri
- c. spesso rivestono ruoli critici e hanno dimensioni notevoli
- d. devono essere rimpiazzati perché iniziano a dare problemi di funzionamento

**3 Quali tra le seguenti forme di trasparenza non sono previste dall'ANSA nella ISO 10746?**

- a. di accesso (access transparency)
- b. di locazione (location transparency)
- c. di concorrenza (concurrency transparency)
- d. di prezzo (price transparency)
- e. di replicazione (replication transparency)
- f. ai guasti (failure transparency)
- g. di sicurezza (security transparency)
- h. alla migrazione (mobility transparency)
- i. alle prestazioni (performance transparency)
- j. di scalabilità (scaling transparency)

**4 Le forme di trasparenza più importanti sono:**

- a. accesso e locazione
- b. locazione e concorrenza
- c. accesso e concorrenza
- d. locazione e scalabilità
- e. accesso e scalabilità

**5 I principali svantaggi legati alla distribuzione sono:**

- a. produzione di software
- b. complessità
- c. affidabilità
- d. sicurezza
- e. costo



### 2. Vero o falso

**1 In un sistema informatico centralizzato dati e applicazioni risiedono in un unico nodo.** V F

**2 Un sistema distribuito è anche un sistema parallelo.** V F

**3 Il principale vantaggio dei sistemi distribuiti è l'affidabilità dovuta alla sua ridondanza intrinseca.** V F

**4 Con sistemi legacy si intendono i sistemi informativi obsoleti presenti in azienda.** V F

**5 Generalmente i sistemi distribuiti offrono un miglior rapporto prezzo/qualità che i sistemi centralizzati.** V F

**6 Nei vecchi sistemi non era sufficiente proteggere il sistema dall'accesso fisico delle persone ai locali dove erano presenti i dispositivi da proteggere.** V F

**7 L'utilizzo di tecniche di sviluppo "sperimentali" porta all'aumento di complessità in modo accidentale.** V F



# Evoluzione dei sistemi distribuiti e dei modelli architetturali

In questa lezione impareremo...

- ▶ la classificazione delle architetture distribuite hardware e software
- ▶ il concetto di middleware

## ■ Premessa

Un **sistema distribuito** è ottenuto dalla connessione di più **personal computer** che per molti anni hanno avuto una crescita tecnologica legata alla **legge di Moore** ma che nel prossimo futuro sarà disattesa in quanto non sarà più possibile **raddoppiare la potenza degli elaboratori ogni due anni**.

Il problema maggiore è quello legato alla **miniaturizzazione** dei microprocessori e alla crescita delle frequenze di lavoro!

Attualmente i microprocessori più potenti hanno circuiti larghi 14 nanometri (un miliardesimo di metro) e si sta lavorando per farne di ancora più piccoli, ma ci sono limiti fisici con i quali si dovrà fare i conti quando si arriverà nell'ordine dei 2-3 nanometri dato che a quelle dimensioni i componenti sarebbero larghi quanto **una decina di atomi**!

Le leggi della fisica pongono dei limiti anche a partire dal fatto che non è possibile superare la velocità della luce e quindi, dato che questa nel rame è di circa 200.000 km/s, si ha:

Se vogliamo avere frequenze di lavoro dell'ordine dei GHZ non possiamo superare la distanza di **20 cm** in trasmissione altrimenti verrebbe **generato un ritardo** e se si restringono i circuiti avremmo problemi di **instabilità e di surriscaldamento** in quanto il calore prodotto viene difficilmente dissipato in una architettura così densa.

$$v = \frac{200000 \cdot 10^3}{10^9} = 0,2 \text{ [m/ns]}$$

Questo significa che nel tempo di **1 ns** vengono percorsi **20 cm** di conduttore.

L'unica soluzione è quelle di andare quindi verso **architetture di elaborazione** di diverso tipo, sia dal punto di vista costruttivo (hardware) che del punto di vista logico (software).

## ■ Architetture distribuite hardware: dalle SISD al cluster di PC

Dal punto di vista hardware si è passati alla realizzazione di macchine e sistemi dotati di più di una **CPU** in modo da avere più potenza di calcolo senza “esasperare” i limiti di velocità di ogni singola **CPU**: stiamo parlando di macchine parallele o macchine ad architettura parallela.

Esistono diverse possibilità per classificare le architetture hardware a seconda dei fattori che si prendono come riferimento. Noi ricordiamo quella di **Flynn (1972)** che si basa sui due flussi di informazioni normalmente presenti nei calcolatori:

- flusso delle istruzioni;
- flusso dei dati.

A seconda di come si “combinano” flusso di dati e di istruzioni abbiamo quattro possibili situazioni:

	<b>DATI SINGOLI</b>	<b>DATI MULTIPLI</b>
Istruzioni singole	SISD	SIMD
Istruzioni multiple	MISD	MIMD

- **macchine SISD** (*Single Instruction Single Data*): flusso di istruzioni unico e flusso di dati unico;
- **macchine SIMD** (*Single Instruction Multiple Data*): flusso di istruzioni unico e flusso di dati multiplo;
- **macchine MISD** (*Multiple Instruction Single Data*): flusso istruzioni multiplo e flusso dati unico;
- **macchine MIMD** (*Multiple Instruction Multiple Data*): flusso istruzioni multiplo e flusso dati multiplo.

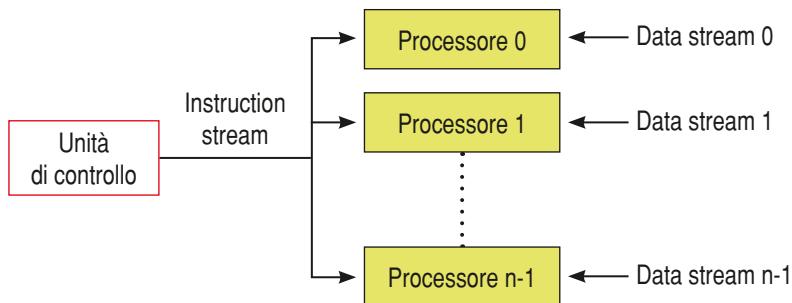
### SISD

Un elaboratore come la macchina di **Von Neumann** che ha un solo flusso dati e un solo flusso istruzioni è una macchina **Single Instruction stream – Single Data stream (SISD)**: tutte le macchine che hanno una singola **CPU** come i Personal Computer, le workstation e i mainframe appartengono a questa categoria.

Nelle macchine a singola **CPU** il flusso di istruzioni è unico e quindi viene eseguito un solo programma alla volta: dopo l'esecuzione della prima istruzione si passa alla seconda e così via fino al termine del programma, le istruzioni sono quindi eseguite in modalità **sequenziale**.

### SIMD

Nelle macchine **SIMD** l'elaborazione avviene su più flussi dati in contemporanea ma con un singolo flusso di istruzioni: generalmente sono presenti più processori (macchine a **vector processor** o ad **array processor**) che eseguono la stessa istruzione su flussi di dati diversi (si sfrutta il parallelismo a livello di dati).



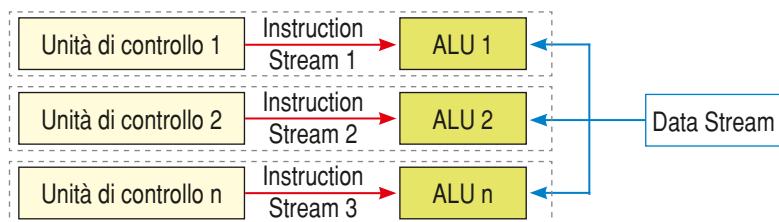
Un elaboratore **SIMD** è particolarmente adatto per realizzare calcoli vettoriali e matriciali soprattutto in ambiente grafico.

## MISD

A questa categoria appartengono gli elaboratori che eseguono più istruzioni sullo stesso flusso dati, cioè **Multiple Instruction stream – Single Data stream**: nel modello di calcolo MISD esistono quindi più processori, ognuno con una propria memoria (registri), la quale a sua volta avrà un proprio flusso di istruzioni che verranno eseguite sullo stesso flusso di dati.

Con questa tipologia di architettura a oggi non sono ancora state costruite macchine da commercializzare, anche se il modello ricorda la modalità di esecuzione delle istruzioni in pipeline.

Avere a disposizione una macchina con queste caratteristiche potrebbe essere utile nell'ambito della crittografia, dove vengono richieste elaborazioni simili da eseguirsi sugli stessi dati in parallelo.



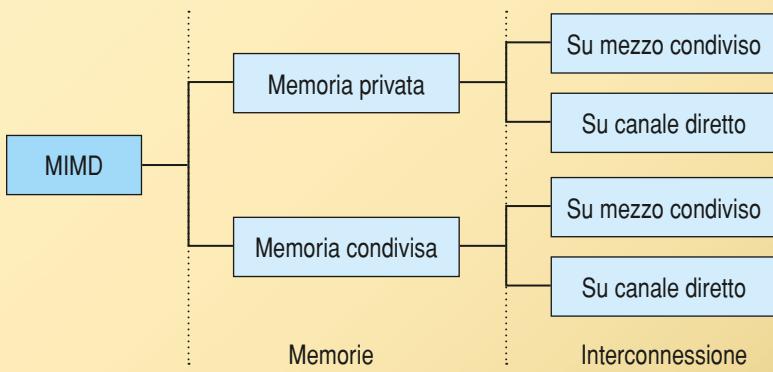
## MIMD

L'architettura **Multiple Instruction stream – Multiple Data stream** comprende tutte le tipologie di elaboratori composti da più unità centrali di elaborazione indipendenti che possono lavorare su stream di dati anch'essi indipendenti. Viene effettuata una ulteriore classificazione delle macchine **MIMD** in riferimento a come è suddivisa la memoria fisica, e cioè possiamo avere:

- macchine **MIMD a memoria fisica condivisa**;
- macchine **MIMD a memoria privata**.

Le prime sono anche conosciute con il nome di **multiprocessor**, mentre le seconde con quello di **multicomputer**.

È possibile introdurre una successiva classificazione dei **multicomputer** e **multiprocessor** a seconda di come sono interconnesse le unità di elaborazione, cioè se utilizzano un unico mezzo fisico condiviso (per esempio a bus oppure ad anello) o attraverso un canale diretto fisico tra coppie di **unità centrali**: la classificazione completa è riportata nella successiva figura.

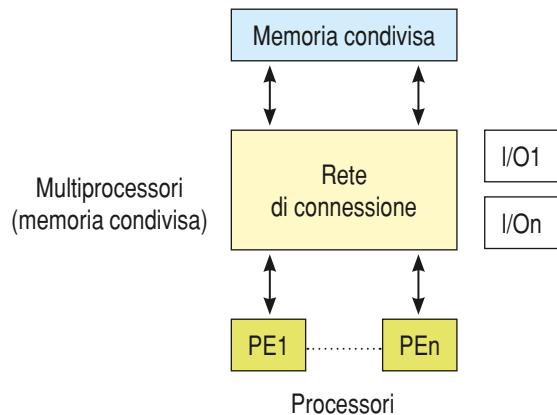


In queste macchine sia il flusso delle istruzioni che dei dati è multiplo: ogni processore legge le proprie istruzioni e opera sui propri dati e quindi sono presenti contemporaneamente più processi in esecuzione. Il parallelismo è fisico ed è molto flessibile dato che ogni singola CPU evolve in modo autonomo.

## MIMD: multiprocessori

I **sistemi a multiprocessori** sono architetture MIMD a memoria condivisa (**shared memory**): la memoria è in comune, e quindi esiste un unico spazio di indirizzamento condiviso tra tutti i processori.

Dato che la comunicazione tra processi avviene mediante variabili condivise è necessario implementare gli opportuni meccanismi di **sincronizzazione** per regolare gli accessi alla memoria in modo da coordinare i diversi processi per gestire la competizione alle risorse comuni.



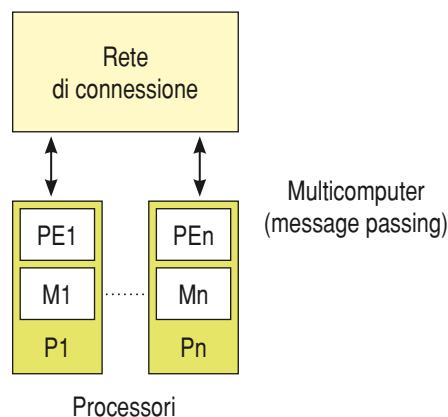
Non necessariamente i processori devono avere una unica memoria in comune (**centralizzata**): possono avere ciascuno una propria memoria e condividerne parte con gli altri processori, in modo da realizzare una memoria condivisa **distribuita**.

## MIMD: multicomputer

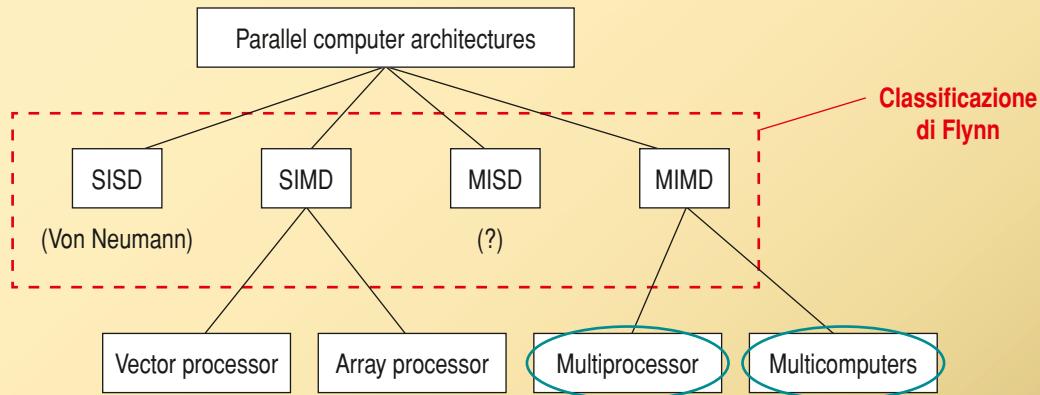
Nei **multicomputer** non è presente una memoria condivisa a livello di architettura e quindi la comunicazione avviene mediante lo **scambio di messaggi** (**message passing**) esplicito, effettuato mediante apposite procedure (**send** e **receive**). Ogni computer possiede una propria area di memoria privata, non indirizzabile da parte dei processori remoti.

### ESEMPIO

Le **LAN** di Personal Computer sono da considerarsi sistemi MIMD dato che hanno **una memoria privata** e sono tra loro interconnessi con **mezzo condiviso**.



Possiamo ora estendere la classificazione di **Flynn** inserendo anche le sottoclassificazioni:



## Cluster computing

Con **cluster computing** si intende un sistema distribuito costituito da un insieme di nodi ad alte prestazioni interconnessi tramite una rete locale ad alta velocità: devono essere **omogenei**, cioè i singoli **nodi** hanno lo stesso sistema operativo, hardware molto simile, e sono connessi attraverso la stessa rete.

Teoricamente un **cluster di PC** ha una potenza di calcolo pari alla somma di quelle dei singoli computer che lo costituiscono, e differisce da una **rete di PC** principalmente:

- ▷ per la **potenza di elaborazione** ad alte prestazioni (**HPC** o **High Performance Computing**);
- ▷ per la **velocità del trasferimento dati** (oltre 1 Gbit/s);
- ▷ per la **centralizzazione fisica** delle macchine, tutti i PC sono montati sullo stesso rack;
- ▷ per la presenza di una **applicazione di management**, residente su un singolo PC, che permette di lanciare processi su altri PC, monitorare il loro comportamento ecc.

Abbiamo due tipiche possibili architetture:

- ▷ organizzazione **gerarchica** con singolo nodo principale: ad esempio **Beowulf**, dove spesso sono usate librerie di **message passing** per il calcolo parallelo (**MPI**);
- ▷ organizzazione **Single System Image**: ad esempio **MOSIX**, che effettua un bilanciamento automatico del carico effettuando una eventuale migrazione di processi.

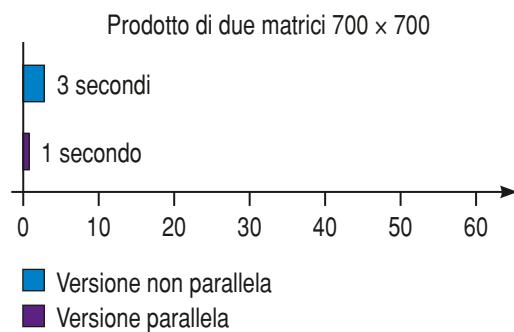
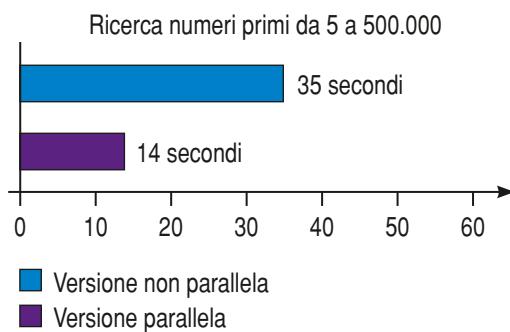
In base alla definizione e considerando le unità centrali come entità, un sistema **cluster di PC** corrisponde all'insieme delle macchine **MIMD a memoria privata**.

L'enorme vantaggio dei **cluster di PC** è quello di affrontare calcoli particolarmente onerosi che sarebbero molto lunghi o impossibili con un solo computer.

### ESEMPIO

Vediamo per esempio il confronto di prestazioni ottenuto con il calcolo parallelo effettuato su un cluster di 4 PC in rispetto all'elaborazione sequenziale in due classiche situazioni di calcolo matematico: *ricerca di numeri primi* e *il prodotto di matrici*.

Come è possibile osservare dai grafici riportati di seguito, nella ricerca dei numeri primi si ha una riduzione del tempo di elaborazione da 35 secondi a 14 secondi, mentre per il prodotto di due matrici di ordine 700 il tempo di computazione scende da 3 secondi a 1 secondo!



(Grafici pubblicati su Rottanuova)

**AREA** digitale



Creiamo cluster HPC con ParallelKnoppix

## Grid computing

Con **grid computing** si intende un sistema distribuito di calcolo altamente decentralizzato, composto da un gran numero di nodi disposti a griglia (**grid**) e caratterizzati da un grado elevato di eterogeneità sia per hardware, per il software, la tecnologia di rete, le politiche di sicurezza ecc.

Il termine “**griglia**” è stato coniato intorno alla metà degli anni Novanta: il vero problema alla base del concetto di **griglia** è la condivisione coordinata di risorse all’interno di una dinamica e multi-istituzionale organizzazione virtuale (**Virtual Organization**, brevemente indicata con **VO**).

La condivisione non è limitata solo allo scambio dei file, ma si estende all’accesso diretto a computer, al software, in generale a tutto l’hardware necessario alla risoluzione di un problema scientifico, ingegneristico o industriale.

Si è passati da un’architettura a 4 livelli di specificità alla sua evoluzione **service-oriented** che ha portato alla definizione della **Open Grid Service Architecture (OGSA)** dove risorse computazionali, di storage, reti, programmi, database ecc., sono tutti dei servizi (**grid services**).

## Sistemi distribuiti pervasivi

Questa è una nuova generazione di SD i cui nodi sono piccoli, mobili, con connessioni di rete wireless e spesso facenti parte di un sistema più grande:

- ▷ sistemi domestici, sistemi elettronici per l’assistenza sanitaria;
- ▷ reti di sensori.

Alcuni requisiti per sistemi pervasivi:

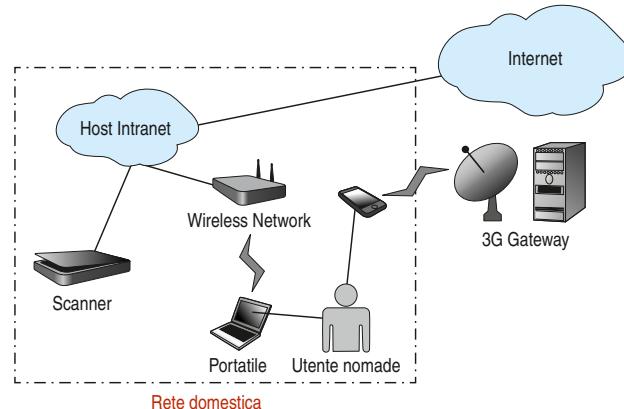
- ▷ **cambi di contesto**: il sistema è parte di un ambiente che può cambiare in ogni momento;
- ▷ **composizione ad hoc**: ogni nodo può essere usato in modi molto diversi da utenti differenti;
- ▷ richiesta la facilità di configurazione;
- ▷ **condivisione come default**: i nodi vanno e vengono, fornendo informazioni e servizi da condividere.

Pervasività e trasparenza della distribuzione non sono facilmente coordinabili: è preferibile esporre la distribuzione piuttosto che cercare di nasconderla.

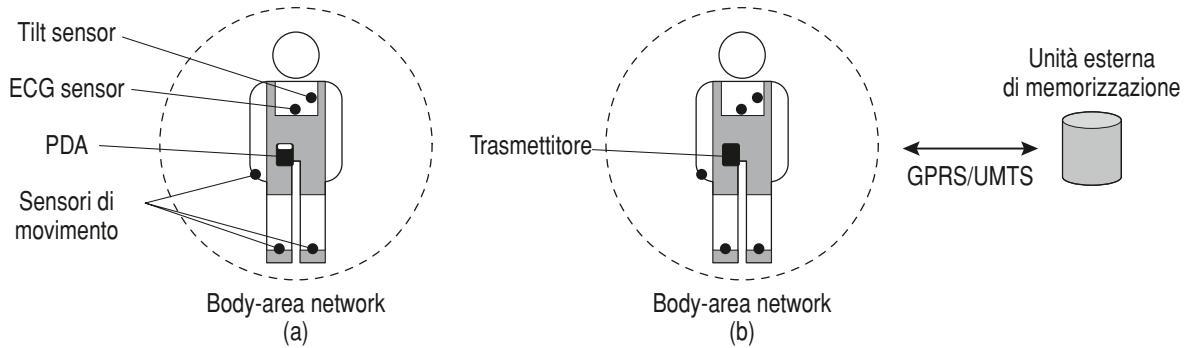
### ESEMPIO

Le **reti domestiche** sono caratterizzate dall’assenza di un amministratore di sistema e generalmente sono sistemi completamente auto-configuranti e autogestiti in quanto generalmente gli utenti non hanno conoscenze specifiche di connettività.

La soluzione più semplice è quella di una home box centralizzata alla quale si connettono wireless tutte le periferiche ed eventuali dispositivi “ospiti”.

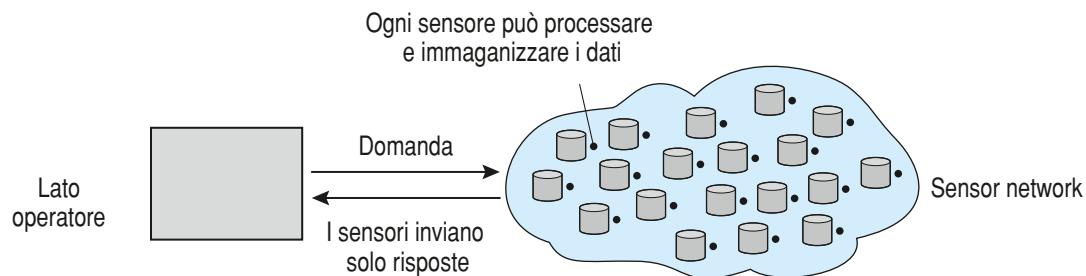


Oggi i sistemi **wearable computing** sono essenzialmente utilizzati per l'assistenza sanitaria ed effettuano il monitoraggio di parametri biologici o memorizzando i dati in un computer palmare (**PDA Personal Digital Assistant**) come nell'esempio riportato nel disegno a) oppure trasmettendo i dati a un sistema di archiviazione remoto, come nel caso b).



I principali problemi che vanno affrontati in questi sistemi, oltre alla memorizzazione dei dati è la prevenzione contro la perdita di dati cruciali, la gestione della sicurezza e le modalità di generazione e propagazione di eventuali allarmi.

Le **reti di sensori** sono composte da molteplici semplici sensori (da 10 a 1000) e ogni sensore corrisponde a un nodo della rete: generalmente i dati raccolti vengono elaborati e memorizzati in una base di dati centralizzata ma in alcuni casi i dati vengono proprio memorizzati in essi stessi, realizzando di fatto anche una **base di dati distribuita**.



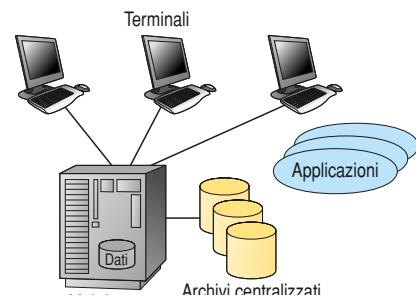
## ■ Architetture distribuite software: dai terminali remoti ai sistemi completamente distribuiti

Come per l'hardware, anche per il software abbiamo avuto una evoluzione nelle architetture distribuite che spesso hanno anticipato i cambiamenti delle architetture hardware e dei loro sistemi operativi. Ricordiamo brevemente le tappe fondamentali.

### Architettura a terminali remoti

La prima tipologia di elaboratori ha questa architettura dove tutte le operazioni vengono effettuate da un'unica entità centrale alla quale sono collegati terminali privi di capacità di elaborazione che si limitano a visualizzare e inviare e ricevere le informazioni alla entità centrale.

In questi sistemi si hanno **terminali omogenei** e quindi il



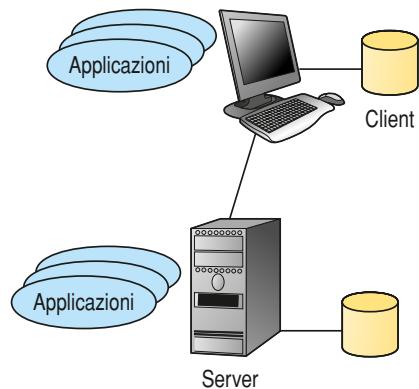
software di gestione svolge dei compiti relativamente semplici, replicando per ciascuno di essi una area di memoria riservata e facendo evolvere singolarmente i singoli task avviati dalle diverse unità remote.

## Architettura client-server

A differenza dell'architettura precedente i **client** non sono "macchine stupide" ma hanno una loro capacità di elaborazione: questi richiedono un servizio a un **server** "di competenza", che è in grado di esaudirne la richiesta, la elabora e invia la risposta al **client**. Come vedremo in seguito, possono essere presenti più **server** che offrono più servizi contemporaneamente così come più **client** possono richiedere lo stesso servizio o servizi differenti sia a **server** diversi che allo stesso **server**.

Inoltre un **server** può essere contemporaneamente anche **client**: quindi può essere **server** per uno (o più) **servizi** e **client** per altri richiedendo i servizi che necessita ad altri server.

**Client** e **server** possono essere tecnologicamente diversi, sia come hardware che come sistema operativo: questa architettura è quella che meglio si presta a far comunicare e cooperare entità non omogenee.



## Architettura WEB-centric

L'evoluzione che negli ultimi anni ha caratterizzato i **sistemi distribuiti** riguarda il **web**: la diffusione capillare delle piattaforme mobili ha, di fatto, provocato uno spostamento delle applicazioni sul **server** facendo "in qualche modo" regredire gli **host** quasi "a terminali stupidi del secolo passato", utilizzati solo per l'interfaccia grafica verso l'utente.

Molte applicazioni gestionali sono state "convertite al web" rendendo il **web server** come centro del sistema distribuito (**web-centric**) al quale i **client** vi accedono per ottenere dei servizi: tutta la computazione avviene sui **server** che restituisce ai **client** il risultato.

I componenti fondamentali di un'applicazione web sono analoghi per certi versi a quelli di una tradizionale applicazione client/server: infatti il web è un sistema distribuito composto da **client** e **server** che accedono a documenti collegati. I server gestiscono insiemi di documenti, mentre i client forniscono agli utenti una interfaccia facile da usare per accedere e presentare questi documenti.

Le architetture possono essere classificate in:

- architetture **web tradizionali**;
- architetture **web multilivello**.

## Architettura cooperativa

L'evoluzione dell'architettura **client-server** è l'architettura **cooperativa** che si basa su entità autonome che esportano e richiedono servizi secondo il modello di sviluppo a **componenti** per la programmazione lato **server** proprio della **programmazione a oggetti**.

Di fatto è stato preso uno dei principi chiave della programmazione orientata agli oggetti, l'**incapsulamento**, applicandolo all'abbattimento delle differenze tra i prodotti usati nell'infrastruttura hardware, software, di programmazione e di rete di un sistema distribuito.

Per rendere efficace l'utilizzo di questi meccanismi è necessario introdurre una standardizzazione sulle modalità con le quali i servizi vengono offerti e come i client possano richiederli: **OdP (Open Distributed Processes)** e **CORBA (Common Object Request Broker Architecture)** sono due esempi di standardizzazione.

## Architettura completamente distribuita

In opposizione alla architettura **web-centric** troviamo l'architettura completamente distribuita, tipica di sistemi dove è necessaria la cooperazione di gruppi di entità paritetiche, come nei sistemi **groupware**, che dialogano tra loro offrendo ciascuno i propri servizi e richiedendo servizi che spesso risultano essere duplicati per garantire l'immunità ai guasti (vengono introdotte ridondanze anche nei dati).

Le tecnologie più importanti sono:

- ▶ **OMG (Object Management Group)**: è un consorzio creato nel 1989 da varie aziende tra cui **Microsoft**, HP, NCR, SUN, allo scopo di creare sistemi di gestione di architetture distribuite;
- ▶ **RMI (Remote Method Invocation)**: è una tecnologia che consente a processi **Java** distribuiti di comunicare attraverso una rete; è specifica appunto del mondo **Java**;
- ▶ **DCOM (Distributed Component Object Model)**: è una tecnologia informatica introdotta nel 1996 da **Microsoft** ed è basato sull'estensione in rete della precedente tecnologia **COM**.

## ■ Architettura a livelli

Per alleggerire il carico elaborativo dei **serventi** sono state introdotte le **applicazioni multi-livello**, nelle quali avviene la separazione delle funzionalità logiche del software in più livelli.

Si introducono gli **strumenti di middleware**, cioè uno strato software “in mezzo” che si colloca sopra al **sistema operativo** ma sotto i **programmi applicativi**, e sono l'evoluzione dei **sistemi operativi distribuiti**.

Con **middleware** si intende una classe di tecnologie software sviluppate per aiutare gli sviluppatori nella gestione della complessità e della eterogeneità presenti nei sistemi distribuiti [D.E. Bakken]. Il **middleware** ha lo scopo di realizzare la comunicazione e le interazioni tra i diversi componenti software di un **sistema distribuito**.

Questo software si interpone quindi negli elaboratori locali tra le applicazioni e il sistema operativo locale creando un'**architettura a tre livelli**, può quindi essere diverso da host a host, e sarà descritta nella prossima lezione.

Lo scopo principale di **middleware** è di permettere e garantire l'interoperabilità delle applicazioni sui diversi sistemi operativi.

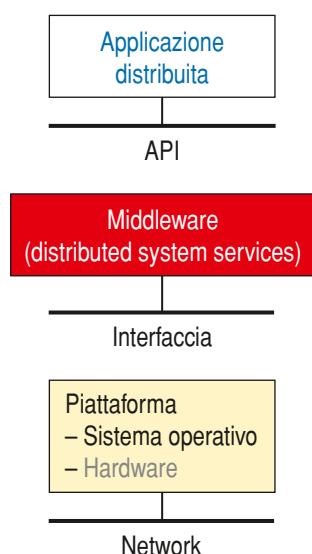
Inoltre permette la connettività tra servizi che devono interagire e collaborare su piattaforme distribuite sulla base di meccanismi di programmazione e **API** relativamente semplici.

Di fatto la presenza di questo strato rende facilmente programmabili i sistemi distribuiti offrendo una specifica modalità di interazione, che può essere per esempio un paradigma di interazione basato sulla chiamata di procedure remote (**RPC, Remote Procedure Call**) oppure un paradigma di programmazione basata sullo scambio di messaggi.

Schematicamente possiamo collocare il **middleware** come mostrato a fianco: ►

Tra le funzionalità del **middleware** ricordiamo:

- ▶ i servizi di astrazione e cooperazione;
- ▶ i servizi per le applicazioni;
- ▶ i servizi di amministrazione del sistema;
- ▶ il servizio di comunicazione;
- ▶ l'ambiente di sviluppo applicativo.



Un **servizio di middleware** è un servizio general-purpose che si colloca tra piattaforme e applicazioni [Bernstein].

Nonostante i molti benefici offerti dal **middleware**, questo però non è la risposta a tutti i problemi legati ai sistemi distribuiti: i problemi connessi alle decisioni “affrettate” o alla “progettazione approssimativa” e/o spesso non completamente competente degli sviluppatori.

Per esempio, chi sviluppa applicazioni distribuite spesso non conosce la differenza nella realizzazione di una chiamata di procedura remota e di una chiamata locale!

A volte il **middleware** è focalizzato solo sulla comunicazione tra componenti e quindi non ha “nessuna responsabilità” sui problemi derivati da errori e malfunzionamenti della applicazione e non è in grado di gestire fallimenti della rete e guasti nei server.

## ■ Conclusioni

Nella seguente tabella ricapitoliamo le caratteristiche essenziali delle diverse architetture distribuite descritte sinora:

	Architetture MIMD		OS Network	Middleware
	Multiprocessori	Multicomputer		
Gradi di trasparenza	Molto alto	Alto	Basso	Alto
Unico SO su tutti i nodi	Sì	Sì	NO	NO
NR. di copie del SO	1	N	N	N
Forma di comunicazione	Memoria condivisa	Messaggi/ memoria condivisa	Files	Modelli specifici
Gestione delle risorse	Globale centralizzata	Globale distribuita	Per nodo	Per nodo
Scalabilità	NO	Modesta	Sì	Variabile
Apertura ai diversi SO	CHIUSO	CHIUSO	APERTO	APERTO

### ODP

The Reference Model for **Open Distributed Processing (ODP)** is an international standard created to give a solid basis for describing and building widely distributed systems and applications in a systematic way.

### Client-server architectures

Distributed services which are called on by clients. Servers that provide services are treated differently from clients that use services.

### Distributed object architectures

No distinction between clients and servers. Any object on the system may provide and use services from other objects.

### Groupware

Software that supports multiple users working on related tasks in local and remote networks. Also called “collaborative software,” groupware is an evolving concept that is more than just multiuser software which allows access to the same data. Groupware provides a mech-

anism that helps users coordinate and keep track of ongoing projects together.



### Grid computing

Everything is represented by a Service: a network enabled entity that provides some capabilities through the exchange of messages.

### Grid service

A web service that conforms to a set of conventions and supports standard interfaces, where each interface defines a set of operations that are invoked by exchanging a defined sequence of messages.

### Middleware

Software that manages and supports the different components of a distributed system. In essence, it sits in the distributed system. In essence, it sits in the middle of the system.

Middleware is usually off-the-shelf rather than specially written software.

## Verifichiamo le conoscenze

### 1. Risposta multipla

**1** La classificazione di Flynn si basa sui due flussi normalmente presenti nei calcolatori:

- a. flusso delle istruzioni
- b. flusso di controllo
- c. flusso dei dati

**2** Alla categoria SISD appartengono i seguenti calcolatori:

- a. personal computer
- b. video terminali
- c. workstation
- d. mainframe

**3** Nella macchine MIMD viene effettuata una ulteriore classificazione:

- a. macchine a memoria fisica condivisa
- b. macchine ad accesso parallelo
- c. macchine a memoria privata
- d. macchine a controllo numerico
- e. macchine a canale condiviso
- f. macchine a mezzo diretto

**4** Un cluster di PC differisce da una rete di PC principalmente perché:

- a. ha una potenza di calcolo pari alla somma di quelle dei singoli computer che lo costituiscono
- b. ha una velocità del trasferimento dati di oltre 1 Gbit/s
- c. ha una centralizzazione fisica delle macchine
- d. esiste una applicazione di management, residente su un singolo PC

**5** Nei sistemi wearable computing possiamo avere:

- a. sensori di movimento
- b. PDA
- c. connessioni cablate
- d. ECG sensor
- e. architetture MISD

**6** Le architetture WEB-centric possono essere:

- a. web tradizionali
- b. web avanzate
- c. web remote
- d. web multilivello.

**7** Le tecnologie con architettura completamente distribuita più importanti sono:

- a. OMG
- b. OdP
- c. CORBA
- d. RMI
- e. DCOM

**8** L'acronimo CORBA significa:

- a. Comunication Object Request Basic Architecture
- b. Common Object Request Basic Architecture
- c. Common Object Request Broker Architecture
- d. Comunication Object Request Broker Architecture

**9** Tra le funzionalità del middleware ricordiamo (indica quella non presente):

- a. i servizi di astrazione e cooperazione
- b. i servizi per le applicazioni
- c. i meccanismi di sincronizzazione
- d. i servizi di amministrazione del sistema
- e. il servizio di comunicazione
- f. l'ambiente di sviluppo applicativo

**10** Lo scopo principale di middleware è quello di (indica quello inesatto):

- a. aiutare gli sviluppatori nella gestione della complessità
- b. permettere di garantire l'interoperabilità
- c. ridurre la quantità di comunicazione
- d. permettere la connettività tra servizi che devono interagire su piattaforme distribuite



## 2. Vero o falso

- 1** Il limite inferiore ottenibile con l'ottimizzazione dell'hardware è legato alla velocità della luce. V F
- 2** La velocità della luce nel rame è di circa 200.000 km/s. V F
- 3** Con frequenze di lavoro dell'ordine dei Megahertz non possiamo superare la distanza di 20 cm senza introdurre ritardi. V F
- 4** Nelle macchine a singola CPU il flusso di istruzioni è unico. V F
- 5** Un elaboratore SIMD non ha trovato a oggi applicazioni commerciali. V F
- 6** Un elaboratore MISD è particolarmente adatto per realizzare calcoli vettoriali e matriciali. V F
- 7** Le macchine MIMD sono anche chiamate multicomputer. V F
- 8** Le macchine MIMD sono anche chiamate multiprocessor. V F
- 9** I sistemi multicomputer sono architetture MIMD a memoria condivisa (shared memory). V F
- 10** Lo scambio di messaggi esplicativi viene effettuato mediante apposite procedure (send e receive). V F
- 11** Nei multiprocessori ogni computer possiede una propria area di memoria privata, non indirizzabile da parte dei processori remoti. V F
- 12** Le LAN di PC sono da considerarsi sistemi MIMD. V F
- 13** Con i cluster di PC è possibile affrontare calcoli particolarmente onerosi che sarebbero molto lunghi o impossibili con un solo computer. V F
- 14** Nelle architetture client-server due client possono collaborare tra loro unicamente attraverso uno o più server che permettono la coordinazione e la condivisione dei dati. V F

## 3. Completamento

- 1** Completa la seguente tabella.

	Architetture MIMD		OS Network	Middleware
	Multiprocessori	Multicomputer		
Gradi di trasparenza				
Unico SO su tutti i nodi				
NR. di copie del SO				
Forma di comunicazione				
Gestione delle risorse				
Scalabilità				
Apertura ai diversi SO				

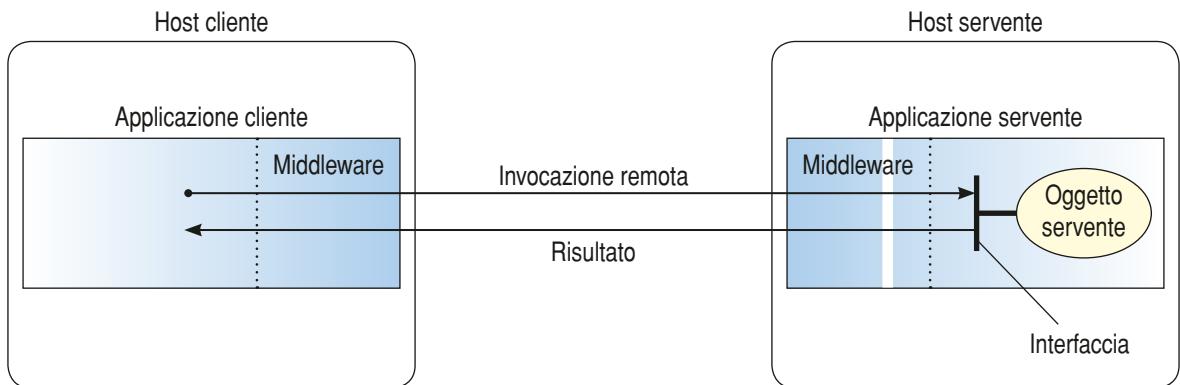
# Il modello client-server

**In questa lezione impareremo...**

- ▶ le caratteristiche del modello client-server
- ▶ l'evoluzione del modello client-server

## ■ Il modello client-server

Il modello **client-server** è costituito da un insieme di **host** che gestiscono una (o più) risorse, i **serventi o server**, e da un insieme di **clienti** (o **client**) che richiedono l'accesso ad alcune risorse distribuite gestite dai **server**. Inoltre ogni processo server può a sua volta diventare **client** per richiedere accesso ad altre risorse gestite da altri (processi) **server**.



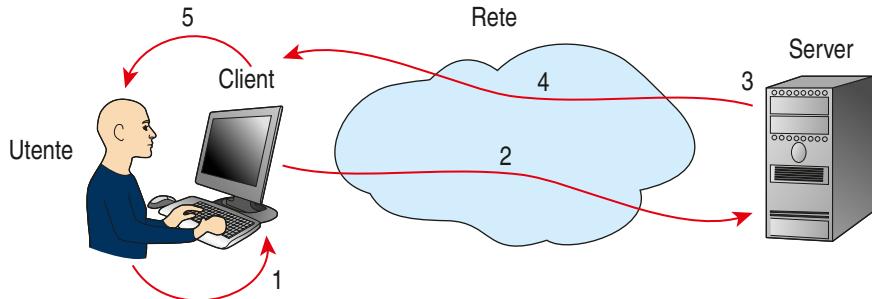
Per essere precisi, non sono gli **host** a essere **server** o **client** ma i **processi che sono in esecuzione** su di essi, dove come **processo** si intende un programma in esecuzione: dato che su un **host** possono essere in esecuzione più processi, un **host** può essere contemporaneamente sia **client** che **server**.

Non bisogna confondere **servizio** e **server**: un **servizio** è un'entità astratta che viene fornito da uno o più **server** che lavorano su macchine spesso differenti e che cooperano via rete.

Descriviamo il modello **client server** anche analizzando la sua evoluzione che oggi gli permette di gestire applicazioni molto complesse (*applicazioni* di tipo *enterprise*) che hanno le seguenti caratteristiche:

- ▷ molti utenti concorrenti che richiedono i servizi;
- ▷ una logica applicativa complessa;
- ▷ archivi di grandi dimensioni con organizzazione di dati complessa e distribuita;
- ▷ notevoli requisiti di sicurezza;
- ▷ sistemi transazionali.

Lo schema di funzionamento di un modello **client-server** è sintetizzato nella seguente figura:



- 1 il **client** manda una richiesta al **server**;
- 2 il **server** (in attesa) riceve la richiesta;
- 3 il **server** esegue il servizio richiesto (generando un thread concorrente);
- 4 il **server** manda una **risposta** ed eventualmente dei dati;
- 5 il **client** riceve la risposta ed eventualmente i dati.

Questo meccanismo è utilizzato sia che i processi fisicamente si trovino in esecuzione su due calcolatori diversi connessi in rete (come nell'esempio descritto) sia che risiedano sul medesimo calcolatore.

Tutte queste attività avvengono in modo trasparente a entrambi i processi.

Di seguito sono riportati alcuni servizi tipici delle architetture **client-server**:

- ▷ **Telnet**: mediante un tale programma (programma **client**) è possibile operare su un computer remoto come si opera su un computer locale; questo è possibile se sulla macchina remota è presente un programma server in grado di esaudire le richieste del client telnet;
- ▷ **HTTP**: il browser è un client http (Web), che richiede pagine Web ai computer su cui è installato un Web server, il quale esaudirà le richieste spedendo la pagina desiderata;
- ▷ **FTP**: tramite un client **FTP** è possibile copiare e cancellare file su un computer remoto, purché qui sia presente un server **FTP**;
- ▷ altri servizi di questo tipo sono **SMTP**, **IMAP4**, **NFS**, **NIS** e così via.

La **programmazione di rete** assume sempre più importanza in quanto la maggior parte delle applicazioni ha bisogno di reperire o scambiare dati presenti (o recuperabili) su altri PC e quindi la necessità prima è quella della connessione reciproca.

## ■ Distinzione tra server e client



► Un **socket** è formato dalla coppia **<indirizzo IP: numero della porta>** che permette di individuare univocamente il gestore di un servizio: verrà descritto nella prossima lezione. ►

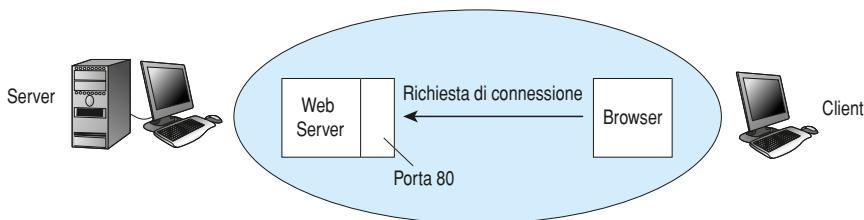
Un programma chiamato **client** richiede dei servizi a un altro programma chiamato **server**. Quest'ultimo è ospitato su un computer chiamato **host** ed è **in ascolto** tramite un **socket** su una determinata **porta**, in attesa che un **client** richieda la connessione: il **client** invia la richiesta al **server** tentando la connessione proprio tramite tale porta, quella cioè su cui il **server** è in ascolto.

Un **client**, quindi, per comunicare con un **server** usando il protocollo **TCP/IP** deve, per prima cosa, "connettersi" al **socket** dell'host dove il **server** è in esecuzione specificando l'**indirizzo IP della macchina** e il **numero di porta** sulla quale il **server** è in ascolto.

Naturalmente su uno stesso computer possono essere in esecuzione **server** diversi, in ascolto su porte diverse (per semplificare con un'analogia, si può pensare al fatto che più persone abitano allo stesso indirizzo, ma a numeri di interno diversi: i numeri di interno rappresentano le porte).

Un **server** "rimane in ascolto" su una determinata porta finché un **client** non crea una comunicazione con il **socket** specificando la porta sulla quale è disponibile il servizio; quindi esegue le richieste del **client** con le risorse che ha a disposizione e rispedisce, se richiesto, i risultati al **client**.

La figura seguente riporta un esempio di servizio **http**, dove il **Web server** rimane in attesa della connessione dei browser dei **client** sulla porta 80.



## Comunicazione unicast e multicast

Ora che è stato introdotto il concetto **client-server**, possiamo passare a distinguere due tipi di comunicazione:

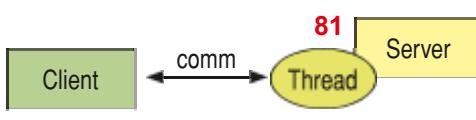
- **unicast**: il **server** comunica con un solo **client** alla volta accettando una richiesta di connessione solo se nessun altro client è già connesso;
- **multicast**: al **server** possono essere connessi più **client** contemporaneamente.

Nel caso di trasmissione **multicast** se la richiesta di connessione tra **client** e **server** va a buon fine il **server**, prima di stabilire il canale di connessione con il **client**, sposta la richiesta dalla porta nella quale è stata effettuata, **port address**, su una nuova porta, così lascia libera la prima in attesa di altre connessioni, e manda in esecuzione un **thread** che soddisfa la richiesta.

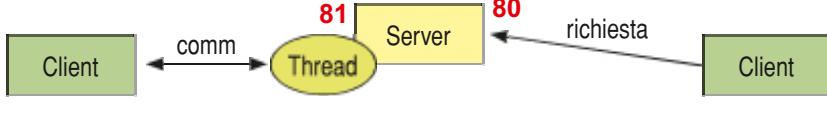
1. Il server riceve una richiesta



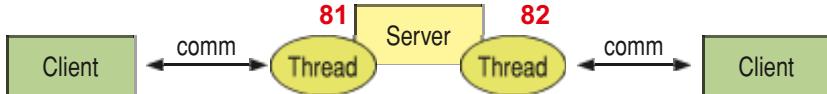
2. Genera un thread per soddisfarla su una nuova porta per potersi mettere in attesa di altre richieste



3. Alla ricezione di una nuova richiesta



4. Genera un nuovo thread... e via di seguito



**ESEMPIO**

Un classico esempio di applicazione **client-server** che implementa trasmissioni **multicast** sono **browser** e **Web server**: infatti il **Web server** deve garantire gli stessi servizi a più utenti e deve sempre restare in ascolto sulla **porta 80**, in attesa di nuove richieste di connessione da parte di nuovi utenti.

## ■ Livelli e strati

Le architetture **client-server** sono normalmente organizzate a “**livelli**” (*tier*) dove ogni **livello** corrisponde a un nodo o gruppo di nodi di calcolo su cui è distribuito il sistema: ciascun livello funziona da **server** per i suoi **client** nel livello precedente e da **client** per il livello successivo ed è organizzato in base al tipo di servizio che fornisce.

Spesso il modello **client-server a livelli** è combinato con quello a **strati** dove ogni strato viene definito dal punto di vista funzionale come un “livello di astrazione”.

Ricapitolando, un sistema adotta un’architettura a livelli e il software in ciascun livello è spesso organizzato internamente a strati (punto di vista di **deployment**).

In generale nelle applicazioni possiamo individuare tre tipi principali di funzionalità che corrispondono a una struttura in tre strati o livelli (modello **three-tier**):

- **front-end** o **presentation tier**: è l’interfaccia verso l’utente;
- logica applicativa o **middle tier**;
- **back-end** con l’accesso alle risorse/ai dati, anche detto **data tier**.

La nomenclatura indicata è quella tipica delle applicazioni Web, ma la suddivisione in tre strati può essere effettuata per ogni tipo di applicazione informatica, con la seguente terminologia:

**Presentation Layer (PL)**: è composta dall’insieme delle procedure o moduli dedicate all’acquisizione e alla presentazione dei dati all’utente (maschere di input, organizzazione di tabelle e tabulati video/cartacei).

Per esempio, nei sistemi Web che visualizzano pagine **HTML**, il **Presentation Layer** è costituito dai moduli del web server che concorrono a creare i documenti **HTML**, come le **Java Servlet**, gli script **PHP** e **ASP**, mentre il **client** può essere identificato con il **browser**.

**Resource Management Layer**: è composto dall’insieme delle procedure che gestiscono i dati, cioè memorizzano e recuperano le informazioni persistenti dagli archivi di massa delle basi di dati.

Nel caso in cui esso è implementato tramite un **DBMS**, è detto semplicemente **Data Access Layer (DAL)**.

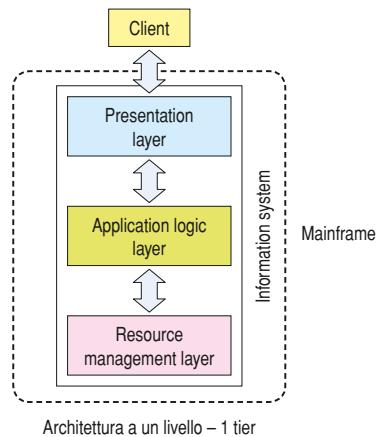
**Business Logic Layer (BLL) o Resource Management Layer (RML)**: è il “corpo centrale” della applicazione che comprende la logica della elaborazione e le definizione delle relazioni esistenti tra le diverse entità.

Per esempio, comprende l’algoritmo che implementa le operazioni legate a un prelievo su un conto corrente bancario, o la sequenza di passi da compiere per effettuare un acquisto on-line.

Nelle moderne architetture **client-server** è possibile individuare una corrispondenza tra livelli e strati e nella nostra trattazione, dopo aver ripercorso storicamente la sua evoluzione, la prenderemo come modello di riferimento.

## Architettura a un livello – 1 tier

Storicamente, a partire dagli anni Settanta, le architetture si riducevano a un solo mainframe al quale erano collegati i terminali “stupidi”: quindi tutta l’elaborazione era effettuata dall’elaboratore centrale e i terminali servivano solo per le fasi di I/O. Questa architettura non rientra nella tipologia client-server e può essere classificata come architettura a un solo livello (**1 tier**) ed è la situazione che si presentava prima dell’avvento dei sistemi distribuiti: ►



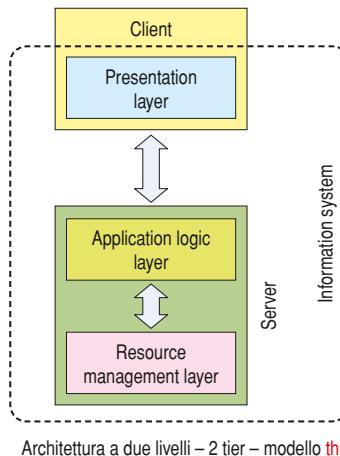
## Architettura a due livelli – 2 tier

Con l'avvento delle reti locali, a partire dagli anni Ottanta, sono nate le architetture **client-server** dove le funzionalità e le responsabilità erano suddivise su due livelli:

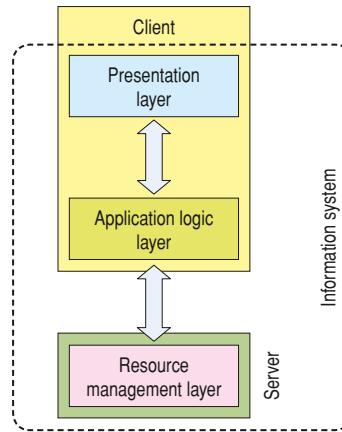
- un livello **server**;
- un livello **client**.

Possiamo individuare due sottocategorie di architetture a due livelli:

- il modello **thin-client**, dove il **server** è responsabile della logica applicativa e gestione dei dati e il **client** è responsabile dell'esecuzione del software di presentazione.
- il modello **thick-client** (o fat-client), dove il **server** è responsabile della gestione dei dati mentre il **client** è responsabile di presentazione e logica applicativa.



Architettura a due livelli – 2 tier – modello thin



Architettura a due livelli – 2 tier – modello thick

Il passaggio dal modello **thin-client** al modello **thick-client** è avvenuto verso la fine degli anni Ottanta con l'aumentare della potenza di calcolo dei PC: mentre il modello **thin-client** è stato il primo passo per effettuare il passaggio dai sistemi mainframe alle architetture distribuite, con il **thick-client** si è spostata parte della applicazione sul **client** favorendo la connessione di host di tipo diverso che era di fatto praticamente impossibile nei modelli precedenti.

Il limite delle architetture **client-server** a due livelli è che sono poco scalabili dato che il **server** deve gestire la connessione e lo stato della sessione di ciascun **client**: questo carico di elaborazione porta alla limitazione del numero limitato di **client** che possono essere gestiti contemporaneamente.

Il passaggio al **thick-client** è stato alla base delle **architetture distribuite moderne**.

## Architettura a tre livelli – 3 tier

A partire dagli anni Novanta l'architettura **client-server** è a tre livelli e a ogni livello corrisponde uno strato architettonico, come precedentemente descritto:

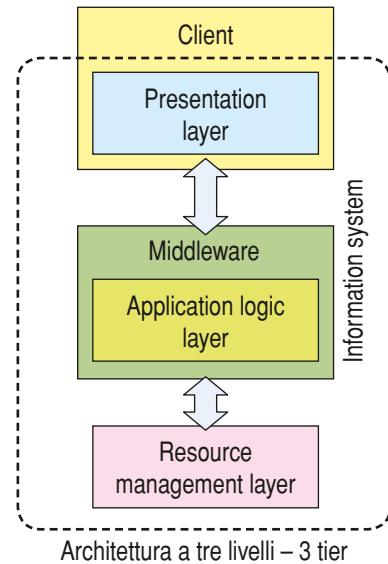
- ▷ **front-end** o **presentation tier**: è l'interfaccia verso l'utente;
- ▷ logica applicativa o **middle tier** (business-tier);
- ▷ **back-end** con l'accesso alle risorse/ai dati, anche detto **data tier** (o resource-tier).

I vantaggi dell'introduzione del **middleware** sono notevoli, soprattutto in termini di prestazioni, in quanto in questo modo si favorisce la distribuzione della quantità di elaborazione a scapito, però, dei tempi di comunicazione. Inoltre il sistema è facilmente scalabile in quanto all'aumentare delle richieste di un servizio è possibile aggiungere qualche server in grado di compensare il carico di lavoro ed è inoltre più tollerante ai guasti.

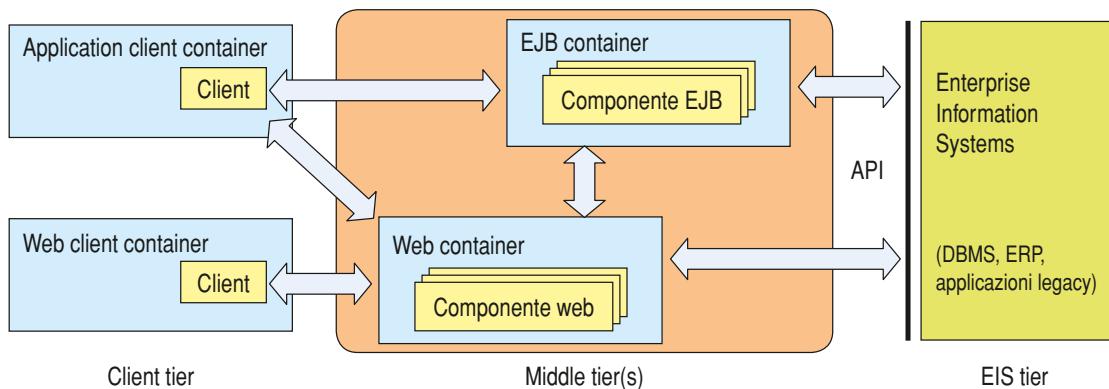
Anche in termini di sicurezza il modello a tre livelli porta notevoli vantaggi in quanto rende possibile l'introduzione di sicurezza a livello di servizio e quindi più facilmente gestibile.

Nei sistemi **3 tier** è però più difficile la loro progettazione, lo sviluppo e l'amministrazione.

Vediamo per esempio l'architettura **3 tier** a componenti della piattaforma **Java EE** (**Enterprise Edition**), rappresentata nel seguente schema:



Architettura a tre livelli – 3 tier



È anche necessario osservare che in questo caso le corrispondenze **tra strati e livelli** non sono sempre così nette, come si può vedere dai due esempi di seguito riportati:

- 1 supponiamo per esempio di avere una applicazione **client** che consiste in una **applet** realizzata in **Java Swing**, che interagisce col **server** mediante l'invocazione di **JSP** (**Java Server Page**): la **presentazione** viene eseguita dal lato **client** solamente **dopo che è stata scaricata** completamente dal lato **server** e solo allora può interagire col **server** stesso;
- 2 anche una semplice applicazione web ha il software del **client** (che in questo caso è il browser **HTML**) che risiede sul **server** oppure le pagine vengono **generate da esso combinando** codice **HTML** con script dinamici sia **Javascript** (**lato client**) che **ASP** o **PHP** (**lato server**).

## Architettura a **n** tier

Le architetture **client-server** a **N livelli** sono una generalizzazione del modello **client-server** a tre livelli dove vengono scomposti e introdotti un numero qualunque di livelli e **server** intermedi.

Questa scomposizione viene effettuata per suddividere ulteriormente i compiti dei vari strati: prende anche il nome di applicazione **multi-tier**.



### Client-server

The **client-server** model is a computing model that acts as a distributed application with partitions tasks or workloads between the providers of a resource or service, called **servers**, and service requesters, called **clients**.

### Enterprise applications

"Enterprise applications are about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data."

### Benefits of the Client-Server Model

Divides Application Processing across multiple machines.

Optimizes Client Workstations for data input and presentation (e.g., graphics and mouse support).

Optimizes the Server for data processing and storage (e.g., large amount of memory and disk space).

**Scales Horizontally:** multiple servers, each server having capabilities and processing power, can be added to distribute processing load.

**Scales Vertically:** can be moved to more powerful machines, such as minicomputer or a mainframe to take advantage of the larger system's performance.

**Reduces Data Replication:** data stored on the servers instead of each client, reducing the amount of data replication for the application.

### 2 – Tier Pros and Cons

Advantages	Disadvantages
<b>Development Issues:</b> <ul style="list-style-type: none"> <li>Simple structure;</li> <li>Easy to setup and maintain.</li> </ul>	<b>Development Issues:</b> <ul style="list-style-type: none"> <li>Complex application rules difficult to implement in database server – requires more code for the client ;</li> <li>Complex application rules difficult to implement in client and have poor performance;</li> <li>Changes to business logic not automatically enforced by a server – changes require new client side software to be distributed and installed;</li> <li>Not portable.</li> </ul>
<b>Performance:</b> <ul style="list-style-type: none"> <li>Adequate performance for low to medium volume environments;</li> <li>Business logic and database are physically close.</li> </ul>	<b>Performance:</b> <ul style="list-style-type: none"> <li>Inadequate performance for medium to high volume environments, since database server is required to perform business logic. This slows down database operations on database server.</li> </ul>

(Prepared By Channu Kambalyal)

### 3 – Tier Pros and Cons

Advantages	Disadvantages
<b>Development Issues:</b> <ul style="list-style-type: none"> <li>Complex application rules easy to implement in application server;</li> <li>Business logic off-loaded from database server and client, which improves performance;</li> <li>Changes to business logic automatically enforced by server – changes require only new application server software to be installed;</li> <li>Application server logic is portable to other database server platforms by virtue of the application software.</li> </ul>	<b>Development Issues:</b> <ul style="list-style-type: none"> <li>More complex structure;</li> <li>More difficult to setup and maintain.</li> </ul>
<b>Performance:</b> <ul style="list-style-type: none"> <li>Superior performance for medium to high volume environments.</li> </ul>	<b>Performance:</b> <ul style="list-style-type: none"> <li>The physical separation of application servers containing business logic functions and database servers containing databases may moderately affect performance.</li> </ul>

(Prepared By Channu Kambalyal)

## Verifichiamo le conoscenze



### 1. Risposta multipla

**1 Indicare quali di queste affermazioni sono false per un modello client-server:**

- a. una macchina server non può essere anche client
- b. non sono gli host a essere server o client ma i processi
- c. un host può essere contemporaneamente sia client che server
- d. ogni processo server può a sua volta diventare client
- e. un processo client non può diventare server
- f. un servizio è un'entità astratta che viene fornito da uno o più server

**2 Quali caratteristiche non sono caratteristiche delle applicazioni di tipo enterprise:**

- a. molti utenti concorrenti che richiedono i servizi
- b. una logica applicativa poco complessa
- c. archivi di grandi dimensioni con organizzazione di dati complessa e distribuita
- d. richiedono modesti requisiti di sicurezza;
- e. gli utenti possono riferirsi a tutte le risorse in un modo esclusivo

**3 Le applicazioni client-server di tipo enterprise hanno (indicare la risposta errata):**

- a. molti utenti concorrenti che richiedono i servizi
- b. una logica applicativa complessa
- c. archivi di grandi dimensioni con organizzazione di dati centralizzata
- d. notevoli requisiti di sicurezza
- e. sistemi transazionali

**4 Quale tra le seguenti non è una tipica applicazione delle architetture client-server?**

- a. Telnet
- b. HTTP
- c. FTP
- d. DNS
- e. SMTP

**5 Nelle applicazioni possiamo individuare tre tipi principali di funzionalità che corrispondono a una struttura in tre strati o livelli (modello three-tier):**

- |  |   |
|--|---|
| <ul style="list-style-type: none"> <li>a. front-end, presentation tier, back-end</li> <li>b. middle tier, front-end, presentation tier</li> <li>c. back-end, data tier, front-end</li> </ul> | <ul style="list-style-type: none"> <li>d. presentation tier, middle tier, data tier</li> <li>e. front-end, logica applicativa, middle tier</li> </ul> |
|--|---|



### 2. Vero o falso

**1 Un processo server può richiedere accesso ad altre risorse a un altro server.**

V F

**2 Un'applicazione si dice distribuita se è installata su diversi PC.**

V F

**3 Un socket è formato dalla coppia <indirizzo IP: numero della porta>.**

V F

**4 In una comunicazione unicast un solo client si può connettere a un port.**

V F

**5 Una applicazione client-server che implementa trasmissioni multicast è il server web.**

V F

**6 Nel modello thick-client il server è responsabile della gestione dei dati e della logica applicativa.**

V F

**7 Il livello back-end con l'accesso alle risorse/ai dati è anche detto data tier.**

V F

**8 Un'applicazione Java si colloca nell'application layer.**

V F

# Le applicazioni di rete

**In questa lezione impareremo...**

- ▷ il concetto di applicazione di rete
- ▷ le tipologie di applicazione
- ▷ a scegliere i protocolli per le applicazioni di rete

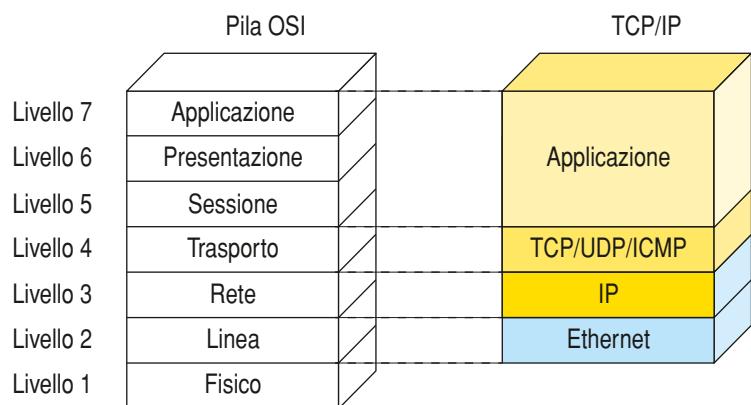
## ■ Il modello ISO/OSI e le applicazioni

Nel modello ISO/OSI e TCP il **livello delle applicazioni** si occupa di implementare le **applicazioni di rete** che vengono utilizzate dall'utente finale, in cui il programmatore non si deve preoccupare dei livelli inferiori ma soltanto utilizzare le primitive di comunicazione messe a disposizione dai diversi protocolli degli altri strati: a tale livello possiamo individuare **Internet** come la struttura più complessa esistente.

Il **modello a pila** (stack) ISO/OSI

può essere messo in relazione con altri stack protocolari, in modo da poterli confrontare: la figura a fianco mostra la pila **TCP/IP**, formata da 5 livelli messa a confronto con la pila **ISO/OSI**.

Possiamo notare che la pila **TCP/IP** è analoga alla pila **OSI**, a eccezione del livello applicazione di Internet che racchiude i livelli 5, 6 e 7.



Il principale scopo delle reti, sia in locale che in remoto, è proprio quello di **condividere dati** mediante **applicazioni**.

Il **livello applicazione** implementa i vari **protocolli**, tra cui:

- ▷ **SNMP**: Simple Network Management Protocol;
- ▷ **SMTP**: Simple Mail Transfer Protocol;
- ▷ **POP3**: Post Office Protocol;
- ▷ **FTP**: File Transfer Protocol;
- ▷ **HTTP**: HyperText Transfer Protocol;
- ▷ **DNS**: Domain Name System.

Questi **protocolli** vengono utilizzati da tutte le **applicazioni di rete generali** come la posta elettronica, il Web, la condivisione di file **P2P**, i giochi multiutente via rete, la messaggistica istantanea, la telefonia via Internet, la videoconferenza in tempo reale, lo streaming di video-clip memorizzati, le autenticazioni in un calcolatore remoto (**Telnet** e **SSH**) ecc.

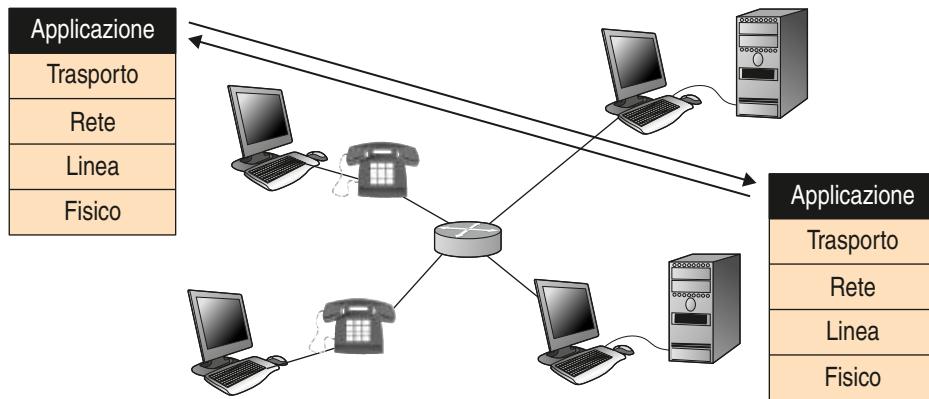
Oltre alle applicazioni generali offerte al pubblico di Internet, sulla rete “gira” un infinito numero di **applicazioni proprietarie**, cioè sviluppate all’interno di un’organizzazione per la gestione privata, come per esempio il collegamento tra filiali remote di una società commerciale, la connessione in tempo reale degli agenti col magazzino centrale ecc.

Non va quindi confusa l’applicazione con il **livello di applicazione**: il livello di applicazione è lo strato protocollare che mette a disposizione i protocolli mediante i quali le applicazioni possono comunicare tra host remoti presenti sulla rete.

## ■ Applicazioni di rete

In generale una **applicazione di rete** è costituita da un insieme di programmi che vengono eseguiti su due o più computer **contemporaneamente**: questi operano interagendo tra loro utilizzando delle risorse comuni, accedendo cioè **concorrentemente** agli archivi (database), mediante la rete di comunicazione che li connette.

L’**applicazione di rete** prende anche il nome di **applicazione distribuita** dato che non viene eseguita su di un solo elaboratore (concentrata).

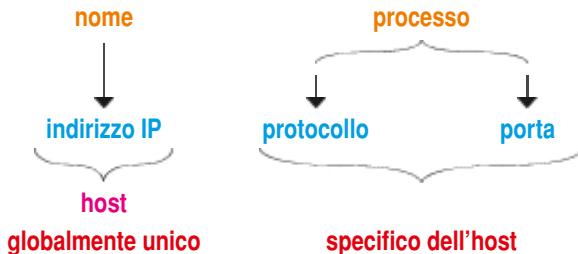


I **processi** hanno la necessità di scambiare messaggi con gli altri **processi** della medesima **applicazione**, sia che essi appartengano alla stessa rete **locale** oppure che siano **remoti** e quindi dislocati dall’altra parte del globo: per comunicare tra loro questi processi devono mettersi in “contatto” tramite i loro indirizzi e utilizzando i servizi offerti dal **livello di applicazione**.

## Identificazione mediante socket

Affinché un processo, presente su un determinato host, invii un messaggio a un qualsiasi altro host, il processo mittente deve identificare il processo destinatario in modo univoco. L'identificazione non può avvenire soltanto mediante l'**indirizzo IP** del destinatario, in quanto quest'ultimo individua soltanto l'host ma non il processo specifico. Pertanto l'identificazione deve tenere conto di due informazioni, l'**indirizzo IP** e il **processo** appartenente a quel determinato host, ovvero si ha:

- ▷ un'**identificazione del nodo** su cui opera il processo con cui si desidera comunicare;
- ▷ un'**identificazione del** particolare **processo** all'interno di quel nodo.



### ESEMPIO

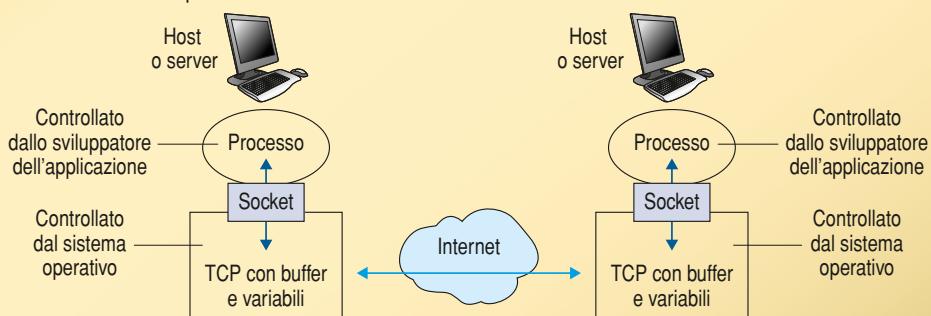
Un caso tipico è quello rappresentato da un server sul quale sono disponibili più servizi, tra i quali:

- ▷ **email**: viene inviata usando il protocollo applicativo **SMTP**, quindi occorre inviare un messaggio opportunamente codificato alla porta **TCP 25** del server;
- ▷ **sito Web**: per richiedere una pagina Web si usa il protocollo applicativo **HTTP**, che invia un opportuno messaggio di richiesta alla porta **TCP 80** del server;
- ▷ per trasformare un nome di un calcolatore in un indirizzo **IP** si invia una opportuna richiesta alla porta **UTP 53** del server che offre il servizio **DNS**.

Per richiedere il **servizio** a questo **server** è quindi necessario specificare l'**indirizzo dell'host** e il **tipo di servizio desiderato**.

L'identificazione univoca avviene conoscendo sia l'**indirizzo IP** che il **numero di porta** associato al processo in esecuzione su un host: questo meccanismo è già introdotto nelle lezioni precedenti e prende il nome di meccanismo dei **socket**.

Come abbiamo visto in precedenza un **socket** è formato dalla coppia <**indirizzo IP:numero della porta**>; si tratta di un identificatore analogo a una porta, cioè a un punto di accesso/uscita: un processo che vuole inviare un messaggio lo fa uscire dalla propria "interfaccia" (**socket** del mittente) sapendo che un'infrastruttura esterna lo trasporterà attraverso la rete fino alla "interfaccia" del processo di destinazione (**socket** del destinatario).



Un **socket** consente quindi di comunicare attraverso la rete utilizzando la pila **TCP/IP** ed è quindi parte integrante del protocollo: le **API** mettono a disposizione del programmatore gli strumenti necessari a codificare la connessione e l'utilizzo del **protocollo di comunicazione**.



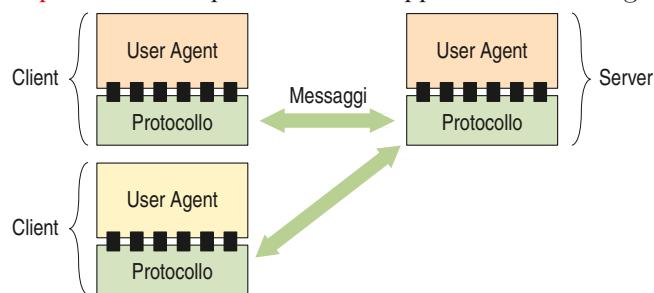
## Zoom su...

### API – APPLICATION PROGRAMMING INTERFACE

Le **Application Programming Interface** rappresentano un insieme di procedure, utilizzabili dal programmatore, utili alla stesura di applicazioni di rete. Le **API** forniscono l'interfaccia tra l'applicazione e lo strato di trasporto realizzando quella che si chiama **astrazione tra hardware e programmatore**, svincolando in tal modo gli sviluppatori dalle problematiche di comunicazione e trasferimento dati che sono i compiti degli strati inferiori.

L'applicazione di rete può essere vista come composta da **due parti**:

- una **user agent**, che funge da **interfaccia** tra l'utilizzatore dell'applicazione e gli aspetti comunicativi;
- l'implementazione dei **protocolli** che permettono all'applicazione di integrarsi con la rete.



### ESEMPIO

In un **browser Web** possiamo individuare queste **due componenti**:

- l'**interfaccia utente** che serve a visualizzare i documenti ricevuti dai **client**, a permettere la loro navigazione e a richiedere nuovi documenti specificando la loro **URL**;
- il **motore del browser** che è la parte che si preoccupa di inviare le richieste ai vari **server** e di ricevere le risposte.

## ■ Scelta dell'architettura per l'applicazione di rete

Il primo passo che il programmatore deve effettuare per progettare una applicazione di rete è la **scelta della architettura** dell'applicazione; sintetizziamo le principali caratteristiche delle architetture attualmente utilizzate, e cioè:

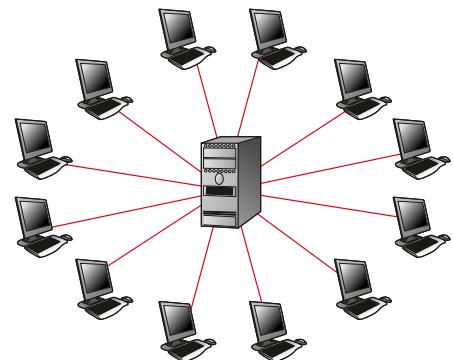
- **client-server**;
- **peer-to-peer (P2P)**;
- architetture **ibride** (dove convivono **client-server** e P2P).

### Architettura client-server

Nella **architettura client-server** la caratteristica principale è che deve sempre esserci un **server attivo** che offre un servizio, restando in attesa che **uno o più client** si connettano a esso per poter rispondere alle richieste che gli vengono effettuate.

Un tipico esempio di questa architettura è il **WWW**, dove molteplici **server** (al limite uno per ogni sito pubblicato) possiedono le pagine (statiche o dinamiche) che saranno inviate ai client che ne fanno richiesta tramite i browser.

Il **server** deve sempre essere attivo e deve possedere un indirizzo **IP fisso** dove può essere raggiunto dagli host **client**: quindi l'indirizzo **IP** deve essere **statico**, contrariamente a quello dei **client** che generalmente è **dinamico**.



Un **client** non è in grado di comunicare con gli altri **client** ma solo con il **server**: più **client** possono invece comunicare contemporaneamente con lo stesso **server**.

Se un **server** viene consultato contemporaneamente da molti **client** potrebbe non essere in grado di soddisfare tutte le richieste e potrebbe entrare in stato di **congestione**: è necessario virtualizzare la risorsa realizzando una **server farm**. Questa non è altro che un **server** con un unico **hostname** ma con più **indirizzi IP**, trasparenti rispetto al **client**, sui quali vengono dirottate le richieste di connessione (viene utilizzato per esempio da **Google**, **Amazon**, **Facebook** e tutti i siti che hanno una elevata affluenza di visite).

## Architettura peer-to-peer (P2P)

Nelle architetture **peer-to-peer** (P2P) abbiamo coppie di **host** chiamati **peer** (letteralmente “*persona di pari grado, coetaneo*”) che dialogano direttamente tra loro.

Un sistema **P2P** è formato da un insieme di entità autonome (**peers**), capaci di auto organizzarsi, che **condividono** un insieme di **risorse distribuite** presenti all'interno di una rete. Il sistema utilizza tali risorse per fornire una determinata funzionalità in modo completamente o parzialmente **decentralizzato**.

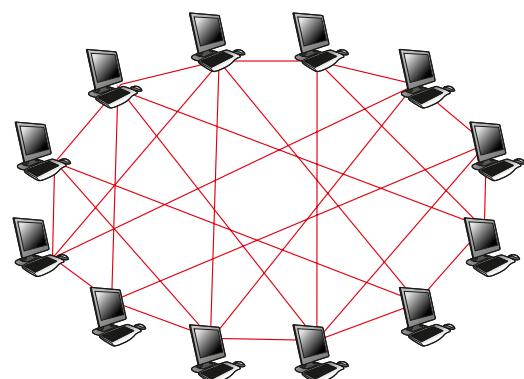
Nei sistemi **P2P** gli host possono essere visti come una comunità che collabora con il binomio dare e ricevere: ogni **peer** fornisce una risorsa e ottiene in cambio altre risorse. Gli esempi più noti sono rappresentati dalle **P2P** in ambito di condivisione di file, come **Emule** e **Gnutella**.

Un **peer** può anche decidere di offrire gratuitamente risorse, magari per la partecipazione a iniziative caritatevoli oppure di ricerca, come per esempio alla ricerca sul cancro oppure agli aiuti ai terremotati mediante donazioni.

## P2P decentralizzato

Nella architettura completamente decentralizzata un **peer** ha sia funzionalità di **client** che di **server** (hanno funzionalità simmetrica e sono anche chiamati **servent**), ed è impossibile localizzare una risorsa mediante un indirizzo **IP** statico: vengono effettuati nuovi meccanismi di indirizzamento, definiti a livello superiore rispetto al livello **IP**.

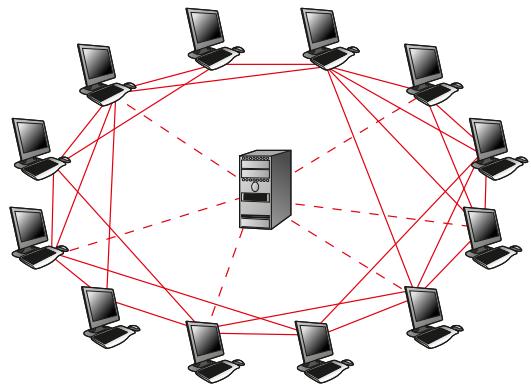
Le risorse che i **peer** condividono sono i dati, la memoria, la banda ecc.: il sistema **P2P** è capace di adattarsi a un continuo cambiamento dei nodi partecipanti (**churn**) mantenendo connettività e prestazioni accettabili senza richiedere l'intervento di alcuna entità centralizzata (come un **server**).



## P2P centralizzato

Il P2P centralizzato è un compromesso tra il determinismo del modello **client-server** e la scalabilità del sistema puro: ha un server centrale (directory server) che conserva informazioni sui **peer** (index, cioè il mapping resorse-peer) e risponde alle richieste su quelle informazioni effettuando quindi la ricerca in modalità centralizzata.

I **peer** sono responsabili di conservare i dati e le informazioni (il **server** centrale non memorizza file), di informare il server del contenuto dei file che intendono condividere e di permettere ai **peer** che lo richiedono di scaricare le risorse condivise.

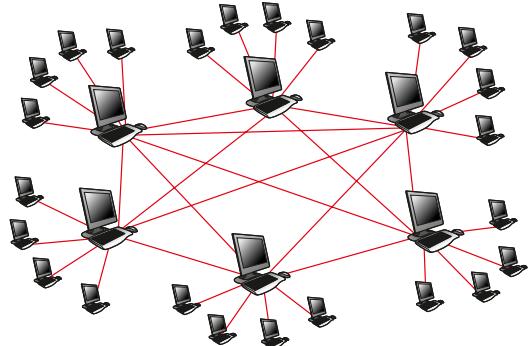


### ESEMPIO

L'implementazione più famosa del P2P centralizzato è **Napster**, dove gli utenti si connettono a un server centrale nel quale pubblicano i nomi delle risorse che condividono.

## P2P ibrido (o parzialmente centralizzato)

Il **P2P ibrido** è un **P2P parzialmente centralizzato** dove sono presenti alcuni **peer** (detti **supernodi** o **super-peer** o **ultra-peer**) determinati dinamicamente (tramite un algoritmo di elezione) che hanno anche la funzione di indicizzazione: gli altri nodi sono anche chiamati **leaf peer**.



### AREA digitale



Applicazioni P2P

## ■ Servizi offerti dallo strato di trasporto alle applicazioni

Le applicazioni richiedono allo strato di trasporto un insieme di servizi specifici oltre ai protocolli necessari per realizzarli, che possono essere standard o realizzati ad hoc.

Tutti i protocolli, sia standard che specifici, hanno in comune una particolarità: **trasferire dei messaggi** da un punto a un altro della rete.

Ogni applicazione deve scegliere tra i protocolli di trasporto quale deve adottare per realizzare un protocollo applicativo in base ai servizi che sono necessari alle specifiche esigenze della applicazione, che possono essere riassunte in:

- ▷ **trasferimento dati affidabile;**
- ▷ **ampiezza di banda;**
- ▷ **temporizzazione;**
- ▷ **sicurezza.**

### Trasferimento dati affidabile

Con trasferimento di dati affidabile intendiamo un servizio che garantisce la consegna completa e corretta dei dati: da una parte sappiamo che alcune applicazioni, come per esempio quelle di audio/video possono tollerare qualche perdita di dati senza compromettere lo scopo dell'applicazione mentre altre, come per esempio il trasferimento di file, richiedono un trasferimento dati affidabile al 100%.

A tale scopo il livello di trasporto mette a disposizione due protocolli:

- **UDP User Datagram Protocol**: il protocollo di trasporto senza connessione da utilizzarsi quando la perdita di dati è un fatto accettabile in quanto non è affidabile, non offre il controllo di flusso, il controllo della congestione, del ritardo e una banda minima;
- **TCP Transmission Control Protocol**: il protocollo **orientato alla connessione** da utilizzarsi quando la perdita di dati è un evento inaccettabile, ovvero quando il trasferimento deve essere affidabile; dà la garanzia di un trasporto senza errori o perdita di informazioni, effettua il controllo di flusso in quanto se il ricevente è più lento del mittente esso rallenterà per non sommergere il ricevente, esegue anche il controllo della congestione limitando il mittente se la rete è sovraccarica, ma non dà garanzie di banda minima.

Riportiamo una tabella dove sono indicati i protocolli utilizzati da alcune applicazioni:

Applicazione	Protocollo a livello applicazione	Protocollo di trasporto sottostante
Posta elettronica	SMTP (RFC 2821)	TCP
Accesso a terminali remoti	Telnet (RFC 854)	TCP
Web	HTTP (RFC 2616)	TCP
Trasferimento file	FTP (RFC 959)	TCP
Multimedia in streaming	Proprietario (RealNetworks)	TCP o UDP
Telefonia Internet	Proprietario (Vonage, Dialpad)	Tipicamente UDP

## Aampiezza di banda (Bandwidth) o Throughput

Alcune applicazioni, come per esempio quelle multimediali, per poter “funzionare” hanno bisogno di avere una garanzia sulla larghezza di banda minima disponibile, cioè possono richiedere un **throughput** garantito di **r bsp**: si pensi alla trasmissione di un evento in diretta in una **Web-TV**.

Altre applicazioni, invece, non hanno questo bisogno come prioritario, ma utilizzano in modo elastico l'ampiezza di banda che si rende disponibile: tipici esempi sono la posta elettronica o i sistemi **FTP**.

Internet ha una natura eterogenea e quindi i protocolli di trasporto non sono in grado di garantire la presenza di una certa quantità di banda per tutta la durata necessaria per trasmettere un messaggio: vedremo che mette a disposizione la possibilità di implementare un protocollo applicativo flessibile che realizza un **circuito virtuale** che si adatta se la banda è insufficiente.

## Temporizzazione

Alcune applicazioni, come la telefonia **VoIP**, i giochi interattivi, gli ambienti virtuali, per essere “realistiche” ammettono solo piccoli ritardi per essere efficaci: lo strato di trasporto non è in grado di garantire i tempi di risposta perché le temporizzazioni presenti assicurano un certo ritardo **end-to-end** tra le applicazioni.

Il protocollo **TCP** garantisce la consegna del pacchetto, ma non il tempo che ci impiega e neppure il protocollo **UDP**, nonostante sia più veloce del **TCP**, è temporalmente affidabile.

La soluzione, come vedremo, è quella di utilizzare un protocollo di trasporto in tempo reale, come **RTP (Real Time Protocol)**, che è in grado di studiare i ritardi di rete e calibrare gli apparati e i collegamenti per garantire di restare nei limiti di tempo prefissati, scegliendo alternativamente quando utilizzare **UDP** e quando **TCP**.

## Sicurezza

Una applicazione può richiedere allo strato di trasporto la **cifratura** di tutti i dati trasmessi in modo tale che anche se questi venissero intercettati da malintenzionati non si perda la riservatezza.

È quindi possibile che vengano richiesti dei **servizi di sicurezza** da applicare per garantire l'integrità dei dati e l'autenticazione **end-to-end**.

Il **problema della sicurezza** nelle reti riveste una grande importanza dato che le reti per loro natura non sono sicure: molteplici sono le minacce e i pericoli per i dati che sono presenti nei diversi **host** e che circolano sulla rete. Garantire la sicurezza di un sistema informativo significa impedire a potenziali soggetti attaccanti l'accesso o l'uso non autorizzato di informazioni e risorse.

## ■ Conclusioni

Riportiamo in una tabella i requisiti richiesti al **servizio di trasporto** da parte di alcune applicazioni:

Applicazioni	Tolleranza alla perdita dei dati	Aampiezza di banda	Sensibilità dal tempo
Trasferimento file	No	Variabile	No
Posta elettronica	No	Variabile	No
Documenti Web	No	Variabile	No
Audio/Video in tempo reale	Sì	Audio: da 5 Kbps a 1 Mbps Video: da 10 Kbps a 5 Mbps	Sì, centinaia di ms
Audio/Video memorizzati	Sì		Sì, pochi secondi
Giochi interattivi	Sì	Fino a pochi K	Sì, centinaia di ms
Messaggistica istantanea	No	Variabile	Sì e no

**Server farm** is a group of servers that are housed in one facility. A server farm comprises dozens, hundreds or even thousands of rack-mounted servers, typically running the same operating system and applications. Using load balancing, the workload is distributed among all machines.



**Servent P2P** knows of only one role, the servent, where the term "servent" is the combination of server and client ("SERVer + cliENT = SERVENT). Every servent performs, or at least is able to perform the same tasks (symmetric roles), which usually consist of searching its local or locally registered resources, forwarding search requests and serving an offered resource.

There are two types of **Internet Protocol (IP)** traffic. They are **TCP or Transmission Control Protocol** and **UDP or User Datagram Protocol**. **TCP** is connection oriented – once a connection is established, data can be sent bidirectional. **UDP** is a simpler, connectionless Internet protocol. Multiple messages are sent as packets in chunks using **UDP**.

Criteria to choose whether to use **TCP** or **UDP** for your game:

- ▶ Use **HTTP** over **TCP** for making occasional, client-initiated stateless queries when it's OK to have an occasional delay.
- ▶ Use persistent plain **TCP** sockets if both client and server independently send packets but an occasional delay is OK (e.g. Online Poker, many MMOs).
- ▶ Use **UDP** if both client and server may independently send packets and occasional lag is not OK.



## Verifichiamo le conoscenze



### 1. Risposta multipla

**1** Quale tra i seguenti non è un protocollo applicativo?

- |         |         |         |
|---------|---------|---------|
| a. HTTP | c. DNS  | e. SMTP |
| b. POP3 | d. SMNP | f. FTP  |

**2** Quale tra le seguenti non è una applicazione di rete?

- |                             |                                   |
|-----------------------------|-----------------------------------|
| a. Posta elettronica        | e. Videoconferenza in tempo reale |
| b. Condivisione di file P2P | f. TV in streaming                |
| c. Scheduler dei processi   | g. Telnet                         |
| d. Telefonia via Internet   |                                   |

**3** Che cosa significa SMTP?

- |                                     |                                     |
|-------------------------------------|-------------------------------------|
| a. Simple Message Transfer Protocol | c. Simple Mail Transfer Protocol    |
| b. System Mail Transfer Protocol    | d. System Message Transfer Protocol |

**4** Qual è il significato di API?

- |                                   |                                      |
|-----------------------------------|--------------------------------------|
| a. Application Protocol Internet  | c. Application Programming Interface |
| b. Application Protocol Interface | d. Application Programming Internet  |

**5** Quali tra le seguenti non sono architetture di una applicazione di rete?

- |                  |        |        |        |
|------------------|--------|--------|--------|
| a. client-server | b. B2B | c. B2C | d. P2P |
|------------------|--------|--------|--------|

**6** Quale tra i seguenti servizi offerti non è garantito dallo strato di trasporto alle applicazioni?

- a. trasferimento dati affidabile
- b. ampiezza di banda
- c. velocità di comunicazione
- d. temporizzazione sicurezza
- e. sicurezza



### 2. Vero o falso

**1** Il protocollo HTTP è a livello di trasporto.

V F

**2** Il protocollo TCP è a livello di applicazione.

V F

**3** L'applicazione di rete prende anche il nome di applicazione distribuita.

V F

**4** L'applicazione di rete prende anche il nome di applicazione multiutente.

V F

**5** Sullo stesso host possono essere in esecuzione molti processi.

V F

**6** Sullo stesso host può essere in esecuzione una sola applicazione.

V F

**7** Un protocollo è una parte integrante di una applicazione ed è sviluppato all'interno di essa.

V F

**8** In un'architettura client-server gli indirizzi IP devono essere statici.

V F

**9** Il termine servent è ottenuto dalla contrazione server-client.

V F

**10** Nel P2P decentralizzato i servent hanno indirizzo IP statico.

V F



# ESERCITAZIONI DI LABORATORIO 1

## IL LINGUAGGIO XML

### ■ Generalità

L'**eXtensible Markup Language (XML)** non è un linguaggio di markup né l'evoluzione **HTML**: è un **meta-linguaggio** di markup, cioè un linguaggio che permette di definire altri linguaggi di markup. **XML** non ha tag predefiniti e non serve né per programmare né per definire pagine **web**, è costituito da un **insieme standard di regole sintattiche** per modellare la struttura di documenti e dati.

Le specifiche ufficiali sono state definite dal **W3C (World Wide Web Consortium)** e sono consultabili all'indirizzo <http://www.w3.org/XML>.

Gli obiettivi iniziali che diedero vita a **XML** erano rivolti alla soluzione di un problema di standard per il Web ma, una volta definito, grazie alla caratteristica di essere abbastanza generale, venne subito utilizzato nei più disparati contesti, dalla definizione della struttura di documenti allo scambio di informazioni tra sistemi diversi, dalla rappresentazione di immagini alla definizione di formati di dati.

### ESEMPIO

Nella programmazione **lato server**, come vedremo, viene utilizzato per la definizione del **deployment descriptor**, cioè del file **web.xml** che viene utilizzato per informare il server in cui è installata un'applicazione **J2EE** riguardo alla configurazione dell'applicazione stessa.

Inoltre in esso possono essere definiti parametri di inizializzazione, la mappatura dei percorsi e, in generale, tutto ciò che contiene informazioni importanti per la corretta esecuzione dell'applicazione.

### ■ XML

Il linguaggio **XML** non possiede tag predefiniti: la principale differenza tra l'**XML** e l'**HTML** è che l'**HTML** si occupa di visualizzare i dati, mentre l'**XML** si occupa di descrivere la natura dei dati che compongono le informazioni.

Possiamo definire anche l'**XML** come un **metalinguaggio** che ha lo scopo di rappresentare contenuti testuali organizzati in modo gerarchico.

La definizione dell'**XML** prescinde dal suo campo di applicazione e quindi anche dalla tecnologia e dall'ambiente di lavoro in cui viene utilizzato: l'**XML** non è un linguaggio per il web, così come non è un meccanismo per la sola rappresentazione di contenuti provenienti da **basi di dati**.

La caratteristica principale dell'**XML** è quella che permette di rappresentare contenuti in un **formato compatibile con qualsiasi sistema** hardware e software in quanto si tratta di un semplice file in formato testo.

## File XML

I file **XML** possono essere trasmessi tra aziende e utenti diversi, senza alcun vincolo, dato che sono **file di testo**: hanno come suffisso .XML.

Quindi i file in **formato XML**:

- ▶ consentono di separare i dati dalla rappresentazione grafica;
- ▶ possiedono un insieme unico di regole di sintassi che consentono di leggere e scrivere, nello stesso modo e con strumenti analoghi, i contenuti rappresentati;
- ▶ analogamente a **HTML** sono basati su tag del tipo **<nome></nome>**; i tag permettono di leggere le informazioni in essi contenute, in base al loro nome e non necessariamente alla loro posizione;
- ▶ a ogni documento **XML** può essere associata la definizione della struttura utilizzata per renderlo facilmente leggibile e per rendere rapido il processo di verifica della struttura del documento.

## ■ Utilizzo dell'XML

Quando si usa l'**HTML** per visualizzare dati, questi ultimi sono memorizzati all'interno del codice **HTML**; con l'**XML**, invece, i dati possono essere memorizzati in un file **XML** separato dal documento **HTML**.

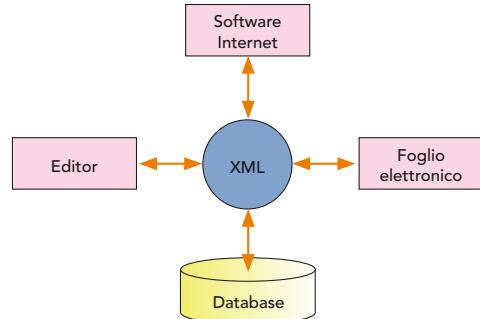
In questo modo possiamo dedicarci all'uso dell'**HTML** per il layout e la presentazione dei dati e avere la garanzia che i dati "sottostanti" non richiedano al file **HTML** alcun cambiamento.

Con l'**XML** i dati strutturati possono essere memorizzati all'interno delle pagine **HTML**: si definiscono così le cosiddette aree **data islands** (isole dei dati).

## Scambio di dati con l'XML

I sistemi di elaborazione e i **database** contengono dati spesso **incompatibili tra loro**: questo è stato uno dei maggiori problemi per gli sviluppatori, che impiegavano una notevole quantità di tempo per scambiare dati via Internet tra computer diversi. La conversione dei dati in formato **XML** può ridurre notevolmente la complessità della comunicazione di informazioni e può creare dati leggibili da diversi sistemi di elaborazione. Il formato **XML** è infatti divenuto compatibile con un numero sempre più ampio di applicazioni.

L'**XML** è ormai uno standard per quanto concerne lo scambio di informazioni di tipo commerciale via Internet.



## Condivisione dei dati

Poiché i dati **XML** sono memorizzati in formato di **testo puro**, l'**XML** fornisce un modo di condividere i dati slegato sia dal software sia dall'hardware. Ciò rende molto più facile creare dati con cui possono lavorare applicazioni diverse.

## Memorizzazione dei dati

L'**XML** può essere utilizzato per memorizzare dati in un file o in un **database**; consente di scrivere

applicazioni per archiviare e recuperare informazioni dagli archivi, mentre applicazioni di tipo generico possono essere usate per visualizzare i dati. L'**XML** può rendere dunque più accessibili gli archivi di dati.

## ■ La sintassi XML

L'**XML** è leggibile su qualsiasi **piattaforma**, ma per ottenere questo risultato occorre prestare un minimo di attenzione nella costruzione dei documenti.

Ogni documento deve presentare un **prologo** costituito dalla **dichiarazione XML**, che identifica la **versione** della specifica **XML** a cui è conforme il documento. Questa dichiarazione può contenere anche informazioni di codifica (**encoding**) dei caratteri e l'indicazione del fatto che il documento è autonomo (**stand-alone**) oppure richiede una descrizione della sua struttura, rappresentata in un file esterno. Sempre nel prologo possiamo trovare la descrizione esplicita della struttura del documento (**DTD, Document Type Definition**).

Anche l'**XML**, come l'**HTML**, prevede l'inserimento di commenti. Questi, se presenti, si devono rappresentare con la stessa simbologia utilizzata nell'**HTML** (<!-- **commento** -->)

Un esempio di prologo è il seguente:

```
<?xml version=" Versione" encoding=" Codifica" standalone=" yes/no"?>
<!-- commenti vari -->
<!DOCTYPE ....>
```

La prima riga (<?...?>) viene chiamata **processing instruction**. È quella che serve per indicare la versione del documento **XML** (per ora 1.0) ed eventuali informazioni di encoding, oltre alla presenza di una definizione di struttura del documento. Il contenuto del documento deve corrispondere a una **gerarchia di tag** non sovrapposti, quindi i tag che si aprono per ultimi devono chiudersi per primi e **ogni tag deve avere una chiusura**.

```
<pagina>
<titolo>esempio di documento in stile XML</titolo>
<texto>
  <paragrafo>testo testo testo</paragrafo>
  <paragrafo>testo testo testo</paragrafo>
</texto>
</pagina>
```

Deve sempre esserci un tag che racchiuda tutti gli altri, che viene chiamato **root** (radice). Inoltre l'**XML** è **case sensitive**, pertanto un tag maiuscolo è diverso dallo stesso tag scritto in minuscolo.

Un documento **XML** si dice **ben formato (well formed)** se possiede i nomi dei tag che non iniziano con il carattere underscore (\_), con un numero e non contengono spazi al loro interno.



### ESEMPIO **Documento XML per definire una rubrica**

Supponiamo che si vogliano scambiare informazioni di database su Internet utilizzando un browser e inviare al server le informazioni, per esempio un questionario compilato dagli utenti. Questo processo richiede un formato che può essere personalizzato per un utilizzo specifico. L'**XML** è la soluzione per questo tipo di problema: infatti la rappresentazione visiva di un documento **XML** risulta

elementare, come mostra l'esempio che segue.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <rubrica>
  - <persona>
    <nome>Paolo</nome>
    <cognome>Camagni</cognome>
    <telefono_casa>0288888888</telefono_casa>
    <telefono_lavoro>0299999999</telefono_lavoro>
    <indirizzo_email>paolo@camagni.it</indirizzo_email>
  </persona>
  - <persona>
    <nome>Riccardo</nome>
    <cognome>Nikolassy</cognome>
    <telefono_casa>0277777777</telefono_casa>
    <telefono_lavoro>0299999999</telefono_lavoro>
    <indirizzo_email>riccardo@nikolassy.it</indirizzo_email>
  </persona>
</rubrica>
```

La prima linea del codice indica la versione dell'**XML** e il set di caratteri utilizzato: caratteri ISO-8859-1 (Latin-1/West European).

La riga successiva descrive l'elemento radice del documento (**rubrica**). L'elemento rubrica possiede come elemento figlio **persona** che a sua volta si compone di altri elementi figli (**nome**, **cognome**, **telefono** ecc.).

## ■ Elementi dell'XML

Gli elementi dell'**XML** sono modificabili ed espandibili (possono essere definiti, e si potrebbe dire "inventati", da chi crea il documento **XML** senza alcuna limitazione) e devono seguire regole molto semplici per specificare i loro nomi.

### ESEMPIO **Messaggio tra due utenti**

Esempio di documento **XML** che indica un messaggio tra due utenti:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <messaggio>
  <mittente>Mario</mittente>
  <destinatario>Maria</destinatario>
  <oggetto>promemoria</oggetto>
  <texto> ricordati che siamo d'accordo di vederci venerdì sera </texto>
</messaggio>
```

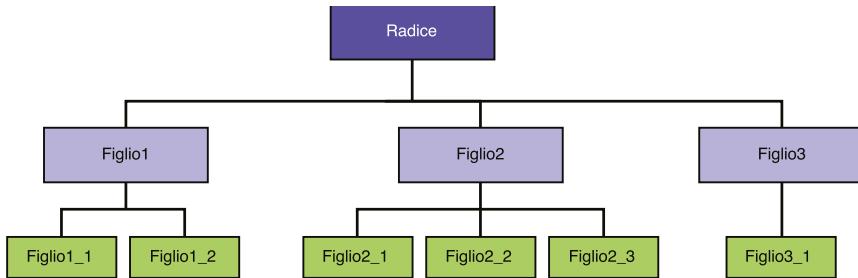
Se l'autore del documento **XML** aggiunge alcune informazioni ulteriori, per esempio la data:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <messaggio>
  <data>2017-07-12</data>
  <mittente>Mario</mittente>
  <destinatario>Maria</destinatario>
  <oggetto>promemoria</oggetto>
  <texto> ricordati che siamo d'accordo di vederci venerdì sera </texto>
</messaggio>
```

l'applicazione è ancora in grado di effettuare l'estrazione degli elementi **<messaggio>**, **<mittente>**, **<destinatario>**, **<texto>** per produrre il medesimo output senza difficoltà.

## Gerarchia degli elementi

Gli elementi di un documento XML sono da considerarsi come nodi appartenenti alla struttura di un albero.



### ESEMPIO

Si consideri la descrizione del contenuto di un libro che parla di XML:

```

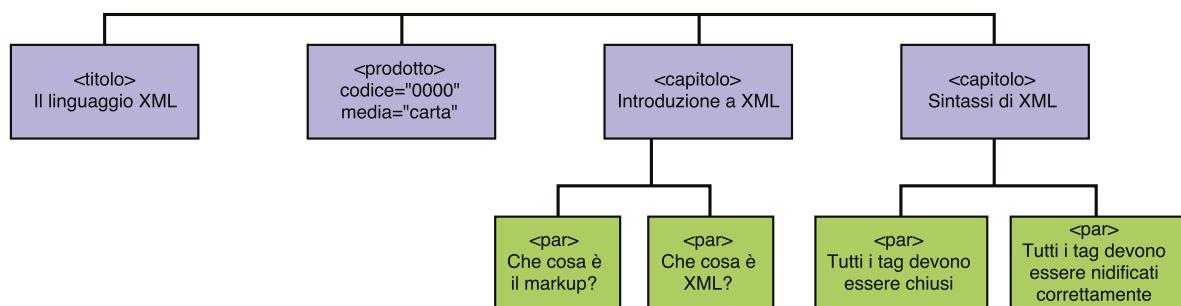
il linguaggio XML
Introduzione a XML
    Che cosa è il markup
    Che cosa è XML
Sintassi di XML
    Tutti i tag devono essere chiusi
    Tutti i tag devono essere nidificati correttamente
  
```

La precedente struttura di dati può essere codificata in XML mediante il codice che segue:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
- <libro>
  <titolo>Il linguaggio XML</titolo>
  <prodotto media="carta" codice="1234"/>
  - <capitolo>
    Introduzione a XML
    <par>Che cosa è il markup</par>
    <par>Che cosa è XML</par>
  </capitolo>
  - <capitolo>
    Sintassi di XML
    <par>Tutti i tag devono essere chiusi</par>
    <par>Tutti i tag devono essere nidificati correttamente</par>
  </capitolo>
</libro>
  
```

Dal punto di vista grafico, il documento XML può essere rappresentato con un albero:



Nell'esempio precedente, `<libro>` è l'**elemento XML** radice; `<titolo>`, `<prodotto>` e `<capitolo>` sono **elementi** figlio di `<libro>` e sono **fratelli** perché hanno lo stesso **padre**.

Un **elemento** è tutto ciò che va dal tag iniziale al tag finale e può avere diversi tipi di contenuto: può essere formato da altri **elementi**, può avere un contenuto **misto** oppure **semplice** e può anche essere vuoto.

Nell'esempio precedente:

- ▷ `<libro>` ha un contenuto formato da altri elementi figli;
- ▷ `<capitolo>` ha un contenuto di tipo misto perché contiene sia testo sia altri elementi;
- ▷ `<par>` ha un contenuto di tipo semplice perché contiene solo testo;

Solo `<prodotto>` ha attributi:

- ▷ l'attributo di nome `<media>` ha valore "carta";
- ▷ l'attributo di nome `<codice>` ha valore 1234.

## Differenza tra attributi ed elementi

I dati possono essere rappresentati attraverso gli **attributi** oppure attraverso gli **elementi**: è da ricordare che in **XML** la regola sintattica per gli attributi prevede sempre che siano racchiusi tra virgolette.

Vediamo un esempio dove riportiamo a confronto le due possibili rappresentazioni:

```
<persone>
  <!-- caso 1: sesso come attributo-->
  <persona sesso="f">
    <nome>Anna</nome>
    <cognome>Rossi</cognome>
    <nazione>Italia</nazione>
  </persona>
  <!-- caso 2: sesso come elemento-->
  <persona>
    <sesso>f</sesso>
    <nome>Anna</nome>
    <cognome>Rossi</cognome>
    <nazione>Italia</nazione>
  </persona>
</persone>
```

Nel primo caso, `<sesso>` è un **attributo**.

Nel secondo caso `<sesso>` è un **elemento figlio** della radice `<persona>`.

Entrambe le codifiche forniscono la medesima informazione e non esistono regole precise per decidere quando è meglio usare **attributi** e quando è meglio scegliere **elementi figlio**.

In generale, si può dire che gli attributi sono opportuni in **HTML**, ma è meglio evitarne l'uso con l'**XML**; infatti utilizzando gli attributi potrebbero nascere alcuni problemi dato che:

- ▷ gli attributi **non possono contenere valori multipli** (gli elementi figlio sì);
- ▷ gli attributi **non descrivono la struttura** (gli elementi figlio sì);
- ▷ gli attributi **sono più difficili da elaborare** da parte di un codice di programma.

## ■ Esercitazione: il magazzino

- 1 Apri il file [magazzino](#).
  - ▶ Modifica il file XML aggiungendo il prologo corretto.
  - ▶ Aggiungi due articoli a piacere.
  - ▶ Modifica l'intero file sostituendo gli elementi visualizzati con gli attributi.
- 2 Crea lo schema ad albero per la struttura dei dati che descrivono le caratteristiche di un'automobile (nome, marca, cilindrata, prezzo, dimensioni, tipo motore e così via).  
Crea il file XML corrispondente allo schema ad albero appena realizzato.
- 3 Crea uno schema ad albero per la struttura dei dati presenti in una fattura (si tenga conto che una fattura è composta da tre parti: l'intestazione che contiene i dati anagrafici del destinatario e del mittente, la parte tabellare che contiene la lista degli articoli venduti e il piede che presenta il totale della fattura e i totali per l'IVA).  
Crea il file XML corrispondente allo schema ad albero appena realizzato.
- 4 Data la seguente tabella [Categorie](#):

ID	Nome categoria	Descrizione
1	Beverages	Soft drinks, coffees, teas, beers and ales
2	Condiments	Sweet and savory sauces, relishes, spreads and seasonings
3	Confections	Desserts, candies and sweet breads
4	Dairy Products	Cheeses
5	Grains/Cereals	Breads, crackers, pasta and cereal
6	Meat/Poultry	Prepared meats
7	Produce	Dried fruit and bean curd

costruisci il file [XML](#) corrispondente e associa a esso il file CSS opportuno per mostrare i dati nel modo seguente:

Nome categoria	Descrizione
Beverages	Soft drinks, coffees, teas, beers, and ales
Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
Confections	Desserts, candies, and sweet breads
Dairy Products	Cheeses
Grains/Cereals	Breads, crackers, pasta, and cereal
Meat/Poultry	Prepared meats
Produce	Dried fruit and bean curd

- 5 Crea un documento XML che descriva la struttura di un documento contenente:  
 Da  
 A  
 Data  
 Ora  
 Priorità  
 Oggetto  
 Corpo

# ESERCITAZIONI DI LABORATORIO 2

## JSON

### ■ Cos'è JSON

JSON è l'acronimo di **JavaScript Object Notation** ed è un formato standard, aperto, adatto a immagazzinare varie tipologie di informazioni e a scambiare queste informazioni tra applicazioni, sia **stand alone** che nel **web**.

La facilità di scrittura e di analisi della notazione **JSON** ne ha fatto uno dei suoi punti di forza e ne ha diffuso l'utilizzo tra gli sviluppatori.

Già dal 2005 Yahoo! supporta **JSON** come alternativa dei suoi servizi web e oggi è supportato praticamente da tutte le applicazioni web.

JSON si basa su un sottoinsieme del linguaggio di programmazione **JavaScript**, Standard ECMA-262 Terza Edizione (1999).



### ■ XML o JSON?

Sappiamo che **XML** è uno strumento **versatile dotato di grande flessibilità in grado di adattarsi a ogni situazione e di crescere insieme a** qualsiasi progetto, dato che con esso è sempre possibile creare nuovi tag e nidificarli in base alle esigenze nelle strutture **XML** già definite.

Inoltre **XML** è molto diffuso, conosciuto dalla maggior parte degli sviluppatori, molto facile da utilizzare e manipolare: nei casi molto articolati, però, a volte risulta poco comprensibile la “vista d’insieme del documento” e richiede una accurata analisi per comprenderne l’organizzazione dei dati.

Come vedremo in questa lezione **JSON** possiede una struttura ancora più semplice di **XML** che consente di capire quasi immediatamente il significato di ciò che descrive: questa particolarità è comoda specialmente quando si lavora a progetti molto grandi e, soprattutto, quando al progetto collaborano più programmati.

Un’ulteriore caratteristica di **JSON** è quella di utilizzare **JavaScript** come linguaggio e, quindi, di poter essere semplicemente integrato nelle applicazioni **Web** poiché tutte le informazioni che vengono ricavate da **JSON** possono essere utilizzate in **JavaScript** utilizzando la funzione **eval()**.

Naturalmente la conoscenza di **JavaScript** è un prerequisito necessario per l’utilizzo di **JSON** mentre **XML** non richiede nessuna conoscenza pregressa.

Non possiamo stabilire se uno dei due sia migliore dell'altro: sono due strumenti tra loro alternativi che però ogni programmatore deve conoscere, anche per il fatto che sono altamente utilizzati e, quindi, sicuramente sono “strumenti del mestiere” indispensabili.

## ■ Formato di JSON

JSON è costituito da due sole strutture:

- un **insieme di coppie** (nome, valore);
- una **lista ordinata** di valori.

Il vantaggio di avere solo queste due strutture lo rende facilmente utilizzabile dato che praticamente tutti i linguaggi possono rappresentare le **coppie** o mediante **array** (nel web abbiamo i “comodi” array associativi) oppure come **record**, e ammettono la definizione di strutture a lista.

Nel web JSON è il formato maggiormente utilizzato per i messaggi di risposta dei servizi web anche per il fatto di utilizzare **JavaScript**: ne vediamo un primo esempio utilizzando la sintassi degli **oggetti letterali** in **JavaScript**.

Un **oggetto letterale** può essere definito con un insieme di coppie proprietà/valori separate dalla virgola: l'intero oggetto viene racchiuso tra parentesi graffe.

```
var JSON = {
    proprietA1: 'valore',
    proprietA2: 'valore',
    ...
    proprietAN: 'valore'
}
```

Come differenza rispetto a **JavaScript** sottolineiamo che **JSON** ammette solo valori semplici e atomici (**stringhe**, **numeri**, **array**, **oggetti letterali**, **true**, **false**, **null**) e, quindi, non può contenere funzioni.

### ESEMPIO

Vediamo un esempio con i dati: utilizziamo **JSON** per memorizzare le informazioni relative a libri (con alcuni attributi):

```
{
  "libro": [
    {
      "id": "01",
      "linguaggio": "Java",
      "edizione": "seconda",
      "autore": "Riccardo Nikolassy"
    },
    {
      "id": "04",
      "linguaggio": "PHP",
      "edizione": "terza",
      "autore": "Paolo Camagni"
    }
  ]
}
```

Vediamo un secondo esempio dove inseriamo del codice JSON all'interno del codice JavaScript di una pagina HTML.

## ESEMPIO

Salviamo il codice seguente nel file **JSON1.htm**:

```

1 <html>
2 <head>
3 <title>Primo esempio con JSON</title>
4 <script language="javascript" >
5 var oggetto1= {
6   "linguaggio" : "Java",
7   "autore" : "Riccardo Nikolassy"
8 };
9 document.write("<h1>Esempio con JSON e JavaScript</h1>");
10 document.write("<br>");
11 document.write("<h3> Linguaggio = " + oggetto1.linguaggio + "</h3>"); 
12 document.write("<h3>Autore = " + oggetto1.autore + "</h3>"); 
13 var oggetto2= {
14   "linguaggio": "PHP",
15   "autore": "Paolo Camagni"
16 };
17 document.write("<br>");
18 document.write("<h3> Linguaggio = " + oggetto2.linguaggio + "</h3>"); 
19 document.write("<h3>Autore = " + oggetto2.autore + "</h3>"); 
20
21 document.write("<hr />"); 
22 document.write(oggetto2.linguaggio + " puoi studiarlo sul libro scritto da " + oggetto2.autore); 
23 document.write("<hr />"); 
24 </script>
25 </head>
26 <body>
27 </body>
28 </html>
```

Ora proviamo ad aprire **JSON1.htm** utilizzando IE o qualsiasi altro browser abilitato JavaScript, ottenendo il seguente risultato:

### Esempio con JSON e JavaScript

Linguaggio = Java

Autore = Riccardo Nikolassy

Linguaggio = PHP

Autore = Paolo Camagni

PHP puoi studiarlo sul libro scritto da Paolo Camagni

## ■ Tipo dei dati

JSON supporta i tipi di dati analizzati di seguito.

### Numeri

Viene utilizzato un unico formato per tutti i numeri ed è quello in virgola mobile a doppia precisione utilizzato da **JavaScript**: non sono previsti l'**ottale** e l'**esadecimale**.

Anche il **NaN** e l'**Infinity** non sono contemplati.

```
var giorni = {marzo: 31}
```

### String

Si tratta di una sequenza di zero o più **caratteri Unicode** delimitati da virgolette e terminante con backslash, dove ogni **carattere** è una stringa di caratteri singola cioè una **stringa** di lunghezza 1.

```
var alunno = {nome: 'Andrea'}
```

### Boolean

Può assumere i due valori **true** oppure **false**.

```
var alunno = {nome: 'Andrea', promosso: true }
```

### Array

È un insieme ordinato di valori racchiuso tra parentesi quadre: i valori sono separati da **,** (virgola) con l'indicizzazione dell'array che può iniziare sia con indice 0 oppure 1.

```
{
  "books": [
    { "linguaggio": "Java" , "edizione": "seconda" },
    { "linguaggio": "Pyton" , "edizione": "prima" },
    { "linguaggio": "C++" , "edizione": "seconda" }
  ]
}
```

### Object

È un insieme **non ordinato** di coppie nome/valore racchiusi tra parentesi graffe: ogni nome è seguito da **:** (due punti) e le coppie nome / valore sono separati da **,** (virgola).

Le chiavi devono essere stringhe diverse tra di loro.

```
{
  "id": "01669",
  "linguaggio": "JAVA",
  "prezzo": 15.50
}
```

### Whitespace

Lo spazio bianco può essere inserito tra una qualsiasi coppia di token per migliorare la leggibilità del codice: confrontiamo le due dichiarazioni seguenti:

```
var x = " ascissa";
var y = " ordinata ";
```

**null**

Utilizzato per l'inizializzazione a vuoto delle variabili.

```
var x = null;
if(x==1)
{
    document.write("<h1>il valore di x è 1</h1>");
}
else
{
    document.write("<h1>il valore di x è null</h1>");
}
```

Riassumiamo i tipi di dati in una tabella:

TIPO	DESCRIZIONE
Number	formato in virgola mobile a doppia precisione formato in JavaScript
String	doppio byte – Unicode con backslash finale
Boolean	vero o falso
Array	una sequenza ordinata di valori
Value	può essere una stringa, un numero, vero o falso, null ecc.
Object	una collezione non ordinata di coppie <b>chiave: valore</b>
Whitespace	lo spazio bianco può essere utilizzato tra una qualsiasi coppia di token
null	vuoto

## ■ Creare oggetti in JSON

Ci sono tre modalità per creare un oggetto JSON utilizzando **JavaScript**:

- 1 creare un oggetto vuoto:

```
var oggetto1 = {};
```

- 2 creare un nuovo oggetto:

```
var oggetto2 = new Object();
```

- 3 creare un nuovo oggetto assegnando i valori:

```
var oggetto3 = { "titolo": "TPSIT vol.3", "prezzo": 23.50 };
```

Vediamo un semplice esempio dove definiamo e utilizziamo un oggetto:

```
1 <html>
2 <head>
3 <title>Creazione di oggetti JSON con JavaScript</title>
4 <script language="javascript" >
5 var oggettoJSON = { "nome": "www.hoepliscuole.it", "anno": 2017 };
6 document.write("<h1>Esempio di JSON con JavaScript</h1>");
7 document.write("<br>");
8 document.write("<h3>Indirizzo sito Web = "+oggettoJSON.nome+"</h3>"); 
9 document.write("<h3>Anno = "+oggettoJSON.anno+"</h3>"); 
10 </script>
11 </head>
12 <body>
```

Aprendolo in un browser otteniamo:

## Esempio di JSON con JavaScript

Indirizzo sito Web = [www.hoepliscuola.it](http://www.hoepliscuola.it)

Anno=2017

L'esempio seguente mostra la creazione di un oggetto array in **JavaScript** utilizzando **JSON**:

```

6 var libri = [
7   "informatica" : [
8     { "linguaggio" : "C++", "prezzo" : 15.50 },
9     { "linguaggio" : "JavaScript", "prezzo" : 17.50 },
10    { "linguaggio" : "Java", "prezzo" : 12.50 },
11  ],
12  "cucina" : [
13    { "portata" : "antipasti", "prezzo" : 25.50 },
14    { "portata" : "dolci", "prezzo" : 13.00 }
15  ]
16 ]

```



### Prova adesso!

- Creazione di oggetti in JSON
- Utilizzo JSON in Javascript

Utilizzando l'oggetto appena definito, completa lo script in modo che sullo schermo venga visualizzata la seguente videata:

### Array di oggetti JSON

linguaggio	C++	linguaggio	JavaScript	linguaggio	Java	portata	antipasti	portata	dolci
prezzo	15.5	prezzo	17.5	prezzo	12.5	prezzo	25.5	prezzo	13

Confronta la tua soluzione con quella presente nel file [JSON\\_libri.htm](#).

È anche possibile realizzare oggetti nidificati, come nel seguente esempio:

```

var alunni = {
  "pino" : {
    "nome" : "Giuseppe",
    "eta" : "21"
  },
  "mario" : {
    "nome" : "Mariolino",
    "eta" : "24"
  },
  "gianni" : {
    "nome" : "Gianni",
    "eta" : "20"
  }
}

```

In questo caso, l'oggetto persone contiene tre “proprietà”: pino, mario, gianni; quest'ultime, a loro volta, sono degli oggetti che contengono una serie di nomi/valori.

## Notazioni a confronto

Riportiamo un esempio con tre diverse modalità di definizione di un oggetto per la codifica dei colori RGB in esadecimale in modo da poterne confrontare la sintassi:

<pre>{   "colorsArray": [     {       "colorName": "red",       "hexValue": "#f00"     },     {       "colorName": "green",       "hexValue": "#0f0"     },     {       "colorName": "blue",       "hexValue": "#00f"     },     {       "colorName": "black",       "hexValue": "#000"     }   ] }</pre>	<pre>{   "colorsArray": [     {       "red": "#f00",       "green": "#0f0",       "blue": "#00f",       "black": "#000"     }   ] }</pre>	<pre>{   "red": "#f00",   "green": "#0f0",   "blue": "#00f",   "black": "#000" }</pre>
---	---	--

Ricordiamoci di separare una proprietà dall'altra con una virgola `,` mentre dopo l'ultima proprietà questa non è richiesta.

## ■ JSON e PHP

A partire da [PHP 5.2](#) sono integrate nativamente delle funzioni per la codifica e la decodifica dello stream **JSON**; esse sono **json\_encode()** e **json\_decode()**:

- **json\_encode( mixed value )** si occupa di trasformare un valore **PHP** nella stringa **JSON** che lo rappresenta;
- **json\_decode( string json [, bool assoc] )** effettua l'operazione contraria trasformando una stringa **JSON** in un valore **PHP** valido.

### La funzione **json\_encode()**

Vediamo un primo esempio dove convertiamo un array **PHP** in formato **JSON**:

```

1 <?php
2 // array per la codifica
3 $vettore = array("sequenza", "selezione", "iterazione");
4 // codifica JSON
5 $stringa = json_encode($vettore);
6 // stampa del risultato
7 echo $stringa;
8 ?>

```

L'esecuzione produce il seguente output:

```
[{"sequenza": "selezione", "iterazione": "iterazione"}]
```

Effettuiamo ora sempre la conversione di un array associativo:

```
1 <?php
2 $vettore = array('qui' => 1, 'quo' => 2, 'qua' => 3);
3 // codifica JSON
4 $stringa = json_encode($vettore);
5 // stampa del risultato
6 echo $stringa;
7 ?>
```



L'esecuzione produce il seguente output:

```
{"qui":1, "quo":2, "qua":3}
```

Descriviamo il funzionamento dello script: la funzione `json_encode()` svolge il compito di restituire una stringa contenente la rappresentazione di un valore partendo da una semplice array: il **server** invia una richiesta di stampa a video dell'operazione di codifica dei dati operata tramite `json_encode()` e il **client** effettuerà una decodifica per restituirli nuovamente sotto forma di vettore: i valori rappresentati nella stringa saranno gli stessi che compongono l'array.

Nel secondo esempio definiamo un oggetto con PHP e lo convertiamo in formato JSON:

```
1 <?php
2 class Atleta {
3     public $nome = "";
4     public $nazione = "";
5     public $sport = "";
6     public $dataNascita = "";
7 }
8 $e = new Atleta();
9 $e->nome = "Pele";
10 $e->nazione = "Brasile";
11 $e->sport = "calcio";
12 $e->dataNascita = date('m-d-Y h:i:s a', strtotime("23-10-1940 12:10:03"));
13 echo json_encode($e);
14 ?>
```



L'esecuzione produce il seguente output:

```
{"nome": "Pele", "nazione": "Brasile", "sport": "calcio", "dataNascita": "10-23-1940 12:10:03 pm"}
```

Vediamo un ulteriore esempio dove aggiungiamo la definizione **MIME** per il formato di ritorno (può essere sia **JSON** che **JavaScript**): questo script ritorna al programma chiamante un vettore che contiene il parametro che ha ricevuto al momento della sua chiamata:



```

1 <?php
2 // definizione MIME per JSON
3 header('Content-type: application/json');
4 // leggiamo il parametro passato
5 $id = $_GET['id'];
6 // creiamo un array e lo ritorniamo al chiamante
7 $data = array("ciao", $id);
8 // Send the data.
9 echo json_encode($data);
10 ?>

```

Chiamiamo la funzione passando direttamente la variabile:



Possiamo visualizzare l'array che viene generato aprendo il risultato col **Blocco Note**:

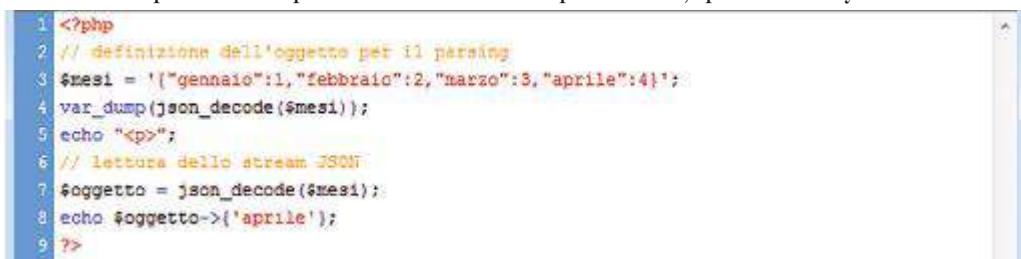


## La funzione `json_decode()`

La funzione `json_decode()` viene utilizzata per trasformare dati **JSON** in **PHP** e, a seconda del suo parametro, ritorna o meno un array associativo (se tale parametro viene omesso equivale a `false`).

```
mixed value json_decode ($obj_json [, $associativo = false ])
```

Vediamo un primo esempio dove omettiamo il parametro, quindi l'array non sarà associativo:

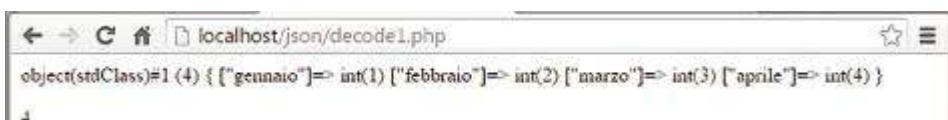


```

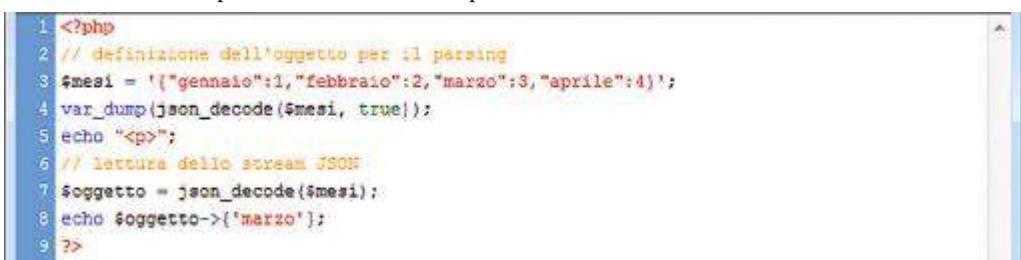
1 <?php
2 // definizione dell'oggetto per il parsing
3 $mesi = '["gennaio":1,"febbraio":2,"marzo":3,"aprile":4]';
4 var_dump(json_decode($mesi));
5 echo "<p>";
6 // lettura dello stream JSON
7 $oggetto = json_decode($mesi);
8 echo $oggetto->('aprile');
9 ?>

```

L'esecuzione produce il seguente output:



Nel secondo esempio dove mettiamo il parametro a `true`:



```

1 <?php
2 // definizione dell'oggetto per il parsing
3 $mesi = '["gennaio":1,"febbraio":2,"marzo":3,"aprile":4]';
4 var_dump(json_decode($mesi, true));
5 echo "<p>";
6 // lettura dello stream JSON
7 $oggetto = json_decode($mesi);
8 echo $oggetto->('marzo');
9 ?>

```

l'esecuzione produce il seguente output:

 int(1) ["febbraio"]=> int(2) ["marzo"]=> int(3) ["aprile"]=> int(4) }."/>

```
array(4) { ["gennaio"]=> int(1) ["febbraio"]=> int(2) ["marzo"]=> int(3) ["aprile"]=> int(4) }
```

In entrambi gli esempi abbiamo estratto il valore di un elemento tramite una chiamata al primo elemento della coppia: nella seconda parte il nostro script dopo l'esecuzione della `json_decode()` visualizza solamente il numero corrispondente al mese indicato nella istruzione 8.



## Prova adesso!

- La funzione `json_encode()`
- La funzione `json_decode()`

- 1** Scrivi un segmento di codice in cui il client deve inviare al server una lista di oggetti selezionati da un utente all'interno di una tabella e i rispettivi costi calcolati dal codice JavaScript. Ad esempio si seleziona il prodotto in un form e si indica in una casella di testo la quantità; lo script deve generare qualcosa simile a:

```
{
  "frutti" : [
    {
      "nome": "Banane", "quantita" : 5
    },
    {
      "nome": "Pomodori", "quantita": 3
    }
  ],
  "costi_unitari" : [ 1.50, 2.50 ]
}
```

Realizza uno script che effettua la codifica e uno che ne esegue la decodifica.

- 2** Scrivi un script dove l'utente seleziona una squadra di calcio e il tipo di competizione (campionato, Champion League ecc.): dopo aver inviato questi dati a una pagina **PHP** visualizza in una tabella le vittorie che tale squadra ha ottenuto in ciascuna competizione. Implementa due soluzioni, una delle quali senza l'utilizzo dei database.

## ■ Conclusioni

Abbiamo visto che **JSON** offre la possibilità di effettuare codifiche e decodifiche praticamente “al volo”, cosa impossibile con **XML** dato che, con esso, è necessario generare un documento intermedio che penalizza le prestazioni degli script: infatti non deve effettuare il parsing di file che, con l'aumento del numero dei dati da gestire così come del traffico generato dai client, può diventare anche estremamente “pesante”.

### AREA digitale

**Lab.3** Web server Apache

### AREA digitale

**Lab.4** Web server Apache in Linux

### AREA digitale

**Lab.5** Web server IIS su Windows

### AREA digitale

Select dinamiche con PHP e JSON

# 2

# Android e i dispositivi mobili

L1 Dispositivi e reti mobili

L2 Android: un sistema operativo per applicazioni mobili

## Esercitazioni di laboratorio

- 1 Android Studio: installazione e configurazione; 2 L'interfaccia grafica di Android Studio;
- 3 Utilizziamo i widget nelle app Android;
- 4 Un'app completa: la calcolatrice; 5 Utilizziamo i sensori;
- 6 La connessione a database locali in Android

### Conoscenze

- Comprendere il ruolo del Sistema Operativo Android
- Conoscere il ciclo di vita di una Activity
- Riconoscere il campo di applicazione di una Activity e di una Service
- Riconoscere i diversi widget utilizzabili nell'interfaccia grafica Android
- Riconoscere il ruolo dell'annotazione @Override

### Competenze

- Riconoscere gli elementi di una applicazione Android
- Realizzare una applicazione di prova
- Utilizzare Android Monitor
- Saper collocare breakpoint
- Modificare le proprietà dei widget in ambiente grafico e nel file activity\_main.xml
- Mostrare a video messaggi a tempo (toast)

### Abilità

- Scaricare, installare e configurare Android Studio
- Installare Android SDK e AVD Manager
- Effettuare il debug con emulatore Android e collegando il dispositivo mediante USB
- Collocare i widget disponibili nel layout
- Utilizzare l'evento OnCreate e OnClick

## AREA *digitale*



Esercizi



Orizzonte 2020

I sistemi operativi Palm OS, Symbian e Linux

Set che compongono Xcode

Versioni di Android

I diversi tipi di tocco su display touch

Il layout degli elementi grafici

Esercizi per il recupero e l'approfondimento



Esempi proposti

Consulta il DVD in allegato al volume



Soluzioni

Puoi scaricare il file anche da [hoepliscuola.it](#)

# Dispositivi e reti mobili

In questa lezione impareremo...

- i tipi di reti mobili
- i dispositivi mobili
- il software per dispositivi mobili

## ■ Premessa

È uso comune parlare indifferentemente di **reti mobili** e **reti wireless** associandole per il fatto che in entrambi i casi i dispositivi si scambiano informazioni senza fili, ma mentre le ultime sono spesso la soluzione di **reti locali aziendali** realizzate senza dover cablare gli edifici, composte da dispositivi per lo più con postazioni fisse, le prime sono caratterizzate da **terminali che possono muoversi** continuando a mantenere attiva la connessione, con la innegabile comodità di essere rintracciati anche quando sono in movimento a chilometri di distanza.

La **mobilità** è la caratteristica fondamentale di queste reti.

Oggi ogni utente ha sempre con sé uno (o più) dispositivi in grado di connettersi con gli altri: telefoni cellulari, smartphone e tablet hanno processori con potenze di calcolo che non hanno nulla da invidiare alle postazioni fisse.

Per le loro caratteristiche fisiche, soprattutto le dimensioni dello schermo, questi necessitano di meccanismi di interfacciamento diversi rispetto alle postazioni fisse, primo fra tutti il meccanismo **touch** che raramente viene implementato sui PC.

Possiamo quindi individuare due elementi essenziali caratterizzanti i dispositivi mobili:

- a) meccanismi che permettono la **connettività mobile**;
- b) **software specifico** per ogni singolo tipo di dispositivo.

## ■ Reti mobili

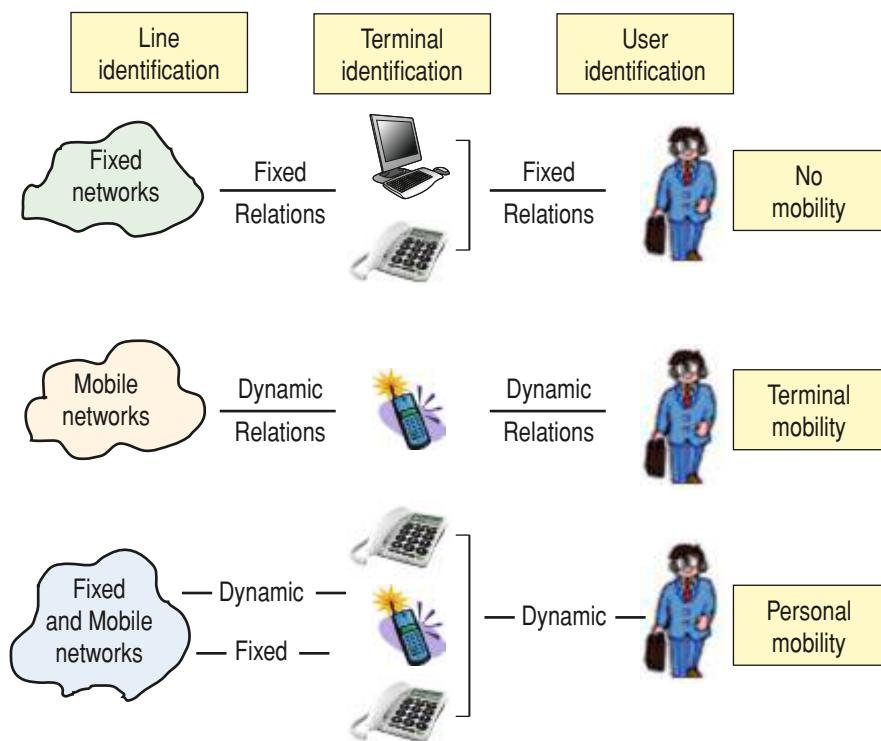
La connettività tra dispositivi mobili viene realizzata con le **reti mobili** che possono essere classificate sia in base ai tipi di mobilità che offrono, sia in base alle tecnologie utilizzate.

## Classificazione delle reti mobili in base ai tipi di mobilità

In base ai tipi di mobilità possiamo distinguere quattro diverse classi:

1. **Access mobility** – Sono i sistemi tipicamente indoor con raggio di azione inferiore ai 500 m, come i sistemi cordless telefonici che tramite un access point connettono alla **rete fissa** di telecomunicazione.
2. **Terminal mobility** – Sono i sistemi nei quali il terminale è in movimento e può accedere, rimanere connesso e identificato in una rete di telecomunicazione.
3. **Service profile portability** – In questo sistema un terminale può connettersi non solo alla propria rete ma anche ad altre reti di telecomunicazione per poter usufruire dei servizi associati al proprio profilo; esiste quindi la portabilità del proprio profilo su provider diversi, in modo trasparente per l'utente.
4. **Personal mobility** – In questo sistema è l'utente che può accedere ai servizi di telecomunicazione tramite qualsiasi terminale; la rete riconosce l'utente e non il terminale; questo permette un'offerta completa di servizi ed in questo caso si parla di **Personal Communications Services (PCS)**.

Possiamo distinguere meglio le diverse situazioni utilizzando la figura riportata di seguito:



- nel caso di **Access mobility** esiste una relazione fissa tra l'utente e il proprio terminale e tra il terminale e la rete fissa e, quindi, siamo praticamente in assenza di mobilità (*no mobility*);
- nel caso di **Terminal mobility** il terminale, tramite la sua SIM card, viene identificato e connesso alla rete; inoltre può connettersi a reti diverse in situazioni di **Service profile portability**;
- nelle situazioni di **Personal mobility** abbiamo la completa dinamicità in quanto l'utente può diventare dinamico, poiché può essere contemporaneamente presente su più terminali fissi o mobili.

## Classificazione delle reti mobili in base alla tecnologia utilizzata

La classificazione delle reti in base alla tecnologia utilizzata introduce il concetto di **generazione**, in particolare di quattro generazioni, con la “quinta alle porte”:

**I generazione** – La **prima generazione** è quella che nei “lontani” anni Ottanta venne identificata dalla sigla **1G**: i sistemi di trasmissione sono di **tipo analogico**, in grado di gestire solo il **traffico voce**. Non esisteva l’interoperabilità a livello europeo e tra i telefoni cellulari di prima generazione ricordiamo il sistema standardizzato **TACS (Total Access Communication System)**, con apparecchiature decisamente voluminose, scarsa qualità audio con frequenti interruzioni, limiti legati alla tipologia di segnale.

**II generazione** – Nel 2006 nacque la prima rete di trasmissione digitale standardizzata e interoperante a livello europeo: si introsse lo standard **GSM (Global System for Mobile communications)**. Con l’avvento del digitale si iniziarono ad avere i primi servizi di trasmissione dati, sotto forma di messaggi di testo (**SMS, Short Message Service**), messaggi multimediali (**MMS, Multimedia Message Service**).

In questa generazione venne sviluppato il protocollo **WAP (Wireless Application Protocol)** in grado di permettere l’accesso da parte dei telefoni a particolari contenuti Internet, realizzando con la tecnologia **GPRS** quella che è conosciuta come generazione **2.5 G**, una via di mezzo fra la seconda e la terza generazione.

La tecnologia **GPRS**, acronimo di **General Packet Radio Service**, consiste in un tipo di trasferimento dei dati detto commutazione a pacchetto, così chiamato perché i dati vengono divisi per essere spediti separatamente per poi essere ricongiunti a destinazione.

A questa tecnologia ha avuto seguito la tecnologia **EDGE (Enhanced Data rates for Global Evolution)** portando un notevole incremento della velocità di connessione.

**III generazione** – Con l’inizio del nuovo secolo iniziarono a diffondersi sistemi multimediali, globali, che integrarono più servizi, accessibili da ogni terminale a disposizione dell’utente, così da realizzare la **Personal Mobility**: lo standard **UMTS (Universal Mobile Telecommunications System)**, tutt’ora attuale e il più utilizzato in Europa, è un’evoluzione del **GSM**, e si impone come standard internazionale di telefonia mobile **3G**.

L’**UMTS** sfrutta il protocollo **W-CDMA (Wideband Code Division Multiple Access)**, una particolare tecnologia di accesso multiplo al canale radio a divisione di codice **CDMA (Code Division Multiple Access)**, che permette di raggiungere fino a **2 Mbps** (teorici) di velocità di trasmissione.

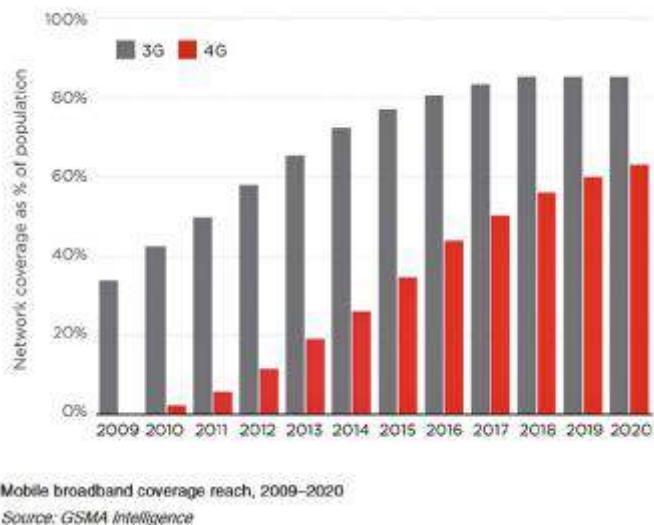
Con il successivo protocollo **HSPA (High Speed Packet Access)** l’offerta di servizi multimediali venne completata con, ad esempio, le videochiamate “finalmente” fluide e senza “scatti” e la navigazione in Internet da terminali mobili caricando con buona velocità anche le pagine ricche di contenuti, script e immagini.

**IV generazione** – La sigla **4G** identifica la quarta, e attuale, generazione dei servizi di telefonia mobile. Il protocollo **HSPA** del **3G** evolve nell’**HSDPA**, un protocollo di trasmissione che migliora ulteriormente le prestazioni consentendo di raggiungere la velocità teorica di **14.4 Mbps** in download e **2 Mbps** in upload, cioè prestazioni simili alle connessioni cablate **ADSL**, e si pone come passaggio intermedio tra l’**UMTS** delle reti **3G** e le reti **LTE (Long Term Evolution)** della quarta generazione.

Con la tecnologia **LTE** e la sua più recente evoluzione **LTE Advanced (LTE-A)** si inviano e ricevono dati a una ottima velocità di connessione, quasi la stessa delle connessioni domestiche, e quindi si può navigare e accedere a cloud, servizi di streaming e video conferenza in alta definizione senza rallentamenti o interruzioni.

In Italia la rete **4G** ha una velocità di connessione di **15 Mbps** che ci colloca tra i primi paesi al mondo come qualità del servizio ma, purtroppo, non abbiamo ancora una buona copertura anche se tutti i nuovi dispositivi, sia smartphone che tablet, sono in grado di utilizzarla.

L'obiettivo dell'**Agenda digitale per l'Italia** prevede la copertura universale a **30 Mbit/s** per la banda larga dell'UE entro il **2020**; il seguente grafico riporta le previsioni di crescita della rete **4G** rispetto alla **3G** per il decennio corrente.



Oltre alle tecnologie **LTE** e **LTE A**, nel 2010 l'**Unione Internazionale delle Telecomunicazioni (ITU)** ha autorizzato anche la tecnologia **Wimax** (*Worldwide Interoperability for Microwave Access*) che consente l'accesso di tipo wireless a reti di telecomunicazioni a banda larga **BWA** (*Broadband Wireless Access*).

## 5G: la nuova frontiera della tecnologia mobile

Ma gli operatori del settore telecomunicazioni sono agguerriti e stanno già studiando il “dopo **4G**”, e l'**ITU** sta parallelamente delineando gli standard del **5G** con la definizione delle linee guida **IMT-2020**, che porteranno la velocità massima teorica di navigazione con smartphone e tablet a **20 Gbps**, che sarebbe 20 volte più veloce della velocità massima teorizzata per il **4G** che è di 1 Gbps!

Con la generazione **5G** si potranno connettere un altissimo numero di dispositivi per kmq, mantenere la connessione anche viaggiando ad altissime velocità e offrire tempi di latenza ridottissimi.

La sfida è quella di connettere sempre più dispositivi e nuove tipologie di dispositivi, da quelli a bordo delle autovetture e dei mezzi di trasporto a quelle **wearable** delle **body network**, per scopi ludici (gaming pervasivo), sanitari o di realtà aumentata, dove anche gli oggetti avranno “dispositivi identificativi”, cioè una saranno dotati di un’identità elettronica e potranno dialogare in rete ed essere controllabili a distanza.

L'**Unione Europea** ha stanziato quasi 80 miliardi di euro di fondi disponibili in 7 anni (dal 2014 al 2020) nel programma di ricerca e innovazione **Orizzonte 2020**: ci si aspettano scoperte, innovazioni e primati mondiali tra i quali il raggiungimento dell’obiettivo **5G** con un insieme di progetti e applicazioni elencati all’indirizzo <https://ec.europa.eu/programmes/horizon2020>.

## ■ Software per dispositivi mobili

La capillare distribuzione di dispositivi mobili, dovuta anche al loro vertiginoso calo dei costi ed alla crescita della capacità di elaborazione, impone agli sviluppatori una profonda riflessione in quanto sia le aziende che i privati utilizzano questi strumenti oltre che per attività ricreative anche per svolgere le normali funzioni lavorative.

### ESEMPIO

È abbastanza normale vedere agenti e rappresentanti inserire ordini e/o consultare i dati dei listini o del magazzino direttamente seduti al tavolo del bar oppure nello scompartimento di un treno: con questi strumenti i tempi di lavoro vengono drasticamente ridotti aumentando quindi potenzialmente il volume d'affari.

È necessario scrivere quindi software adeguato a questi dispositivi che presentano meccanismi di interfacciamento diversi dai normali PC dotati di keyboard e monitor: come primo passo individuiamo le caratteristiche di questi dispositivi.

### I dispositivi mobili

Esistono sostanzialmente quattro tipologie di dispositivi mobili, ognuna dotata di caratteristiche particolari da tenere in considerazione quando progettiamo le applicazioni a loro destinate:

- **Pocket PC**: sono i cosiddetti **palmari**, con display di tipo touch screen, molto compatti ma con una discreta possibilità di espansione e una discreta potenza di calcolo; possiamo individuare due sottofamiglie:
  - **Standard**: non hanno la possibilità di effettuare chiamate vocali;
  - **Phone Edition**: hanno in aggiunta le funzionalità telefoniche.
- **Smartphone**: sono dispositivi prettamente telefonici, meno potenti dei Pocket PC e non tutti dotati di display touch screen, ma in continua crescita di dimensioni e funzionalità;
- **Tablet PC**: sono dispositivi molto simili ai normali Notebook ma dotati di touch screen e di funzionalità di riconoscimento della scrittura;
- **UMPC (*Ultra Mobile PC*)**: sono una via di mezzo tra il Pocket PC e il Notebook, più orientati verso il mercato consumer che business.

### Sistema operativo e applicazioni

Come in tutti i calcolatori, anche nei dispositivi mobili abbiamo due categorie di software:

- software di base, cioè il **sistema operativo**;
- applicazioni, anche chiamate **apps**.

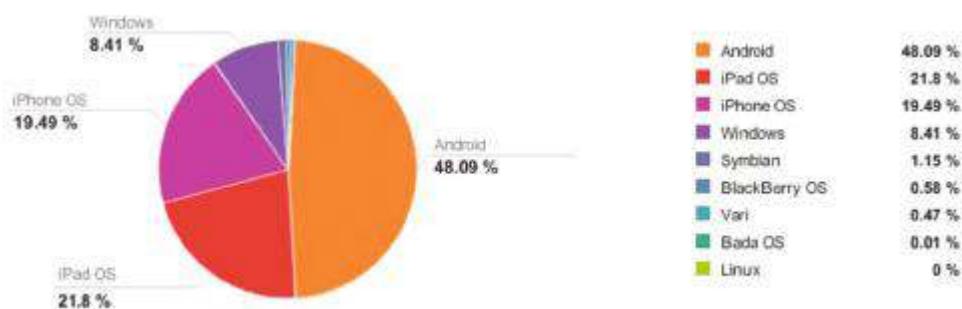
Nel seguito della trattazione faremo una “panoramica” sui principali sistemi operativi oggi “a bordo” degli attuali **smartphone** e per ciascuno di essi descriveremo un ambiente di sviluppo specifico per la realizzazione delle applicazioni.

Essendo un mercato giovane e in grande fermento, sia sistemi operativi che ambienti di sviluppo sono in continua evoluzione e nuove soluzioni vengono proposte “pressoché in continuazione”: per questo motivo non entreremo nel dettaglio di ogni prodotto (ne descriveremo solamente le principali caratteristiche) in quanto potrebbe essere obsoleto già mentre il volume è in stampa; invitiamo quindi il lettore a mantenersi in continuo aggiornamento tramite la rete web.

## ■ Sistemi operativi per dispositivi mobili

Nei dispositivi portatili, a differenza dei sistemi fissi, non è l'utente che decide il **sistema operativo** che vuole utilizzare, ma il produttore dell'hardware, che lo fornisce "imponendolo" assieme al dispositivo. A volte la scelta dell'utente cade su dispositivi che "hanno a bordo" un **sistema operativo** già conosciuto, cioè il medesimo sistema operativo che egli utilizza su altre tipologie di hardware, ma nel settore dei dispositivi portatili questo aspetto si traduce nell'esistenza di poche principali alternative.

Sono molti i **SO** per dispositivi mobili e noi descriveremo sinteticamente solo alcuni di essi, tra i più importanti, individuati mediante un "semplice" criterio operativo, cioè analizzando le visite effettuate ad un sito di primario e-commerce e riportando la classificazione percentuale dei diversi **SO** utilizzati dai navigatori con **dispositivi mobili**.



Individuiamo tre sistemi operativi che coprono circa il 90% delle visite: **Android**, **OS** e **Window**; per ciascuno di essi illustriamo quali sono i punti di forza e gli "immancabili difetti".

### OS per iPhone e iPad

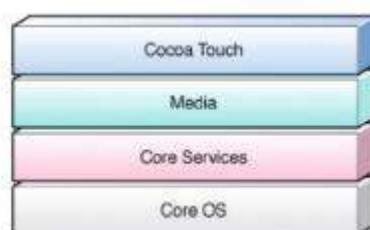
**iOS** (fino a giugno 2010 **iPhone OS**) è il sistema operativo sviluppato da **Apple**, rilasciato per la prima volta nel mercato degli Stati Uniti nel giugno 2007. **iOS** è un sistema operativo proprietario di **Apple** e può essere eseguito solo nei dispositivi **Apple** (**iPad**, **iPhone** e **iPod Touch**), è stato originalmente sviluppato per **iPhone** e ora è utilizzato anche negli **iPod**, **iPad** e **Apple TV**.

L'ottima qualità dei prodotti della casa di Cupertino è anche dovuta al fatto che hanno una perfetta integrazione hardware-software dato che chi programma il sistema operativo è lo stesso che ha progettato la componente elettronica.

Il sistema operativo **iOS** occupa meno di mezzo gigabyte della memoria interna del dispositivo ed è strutturato in quattro livelli di astrazione:

- ▷ il **Core OS** layer;
- ▷ il **Core Services** layer;
- ▷ il **Media** layer;
- ▷ il **Cocoa Touch** layer.

Gli strati più in alto forniscono le astrazioni dei servizi dei livelli più in basso facilitando lo sviluppo delle applicazioni: come in tutti i sistemi organizzati gerarchicamente, ogni livello definisce e implementa nuove funzionalità beneficiando dei servizi offerti dal livello inferiore, indipendentemente dai dettagli implementativi di quest'ultimo.



- Il **Core Services** e il **Core OS** si occupano di offrire i servizi fondamentali del sistema operativo:
- ▶ nel livello più basso (**Core OS**) troviamo lo stack, le funzioni per l'IO, il threading, la gestione processi ecc. (libreria **LibSystem**);
  - ▶ a partire dal livello **Core Services** in poi troviamo tecnologie sviluppate da Apple, in particolare in questo livello sono presenti i framework di base su cui sono sviluppate molte delle **API** di livello più alto (framework **Open GLES**, **Quartz Core**, **Core Animation** ecc.);
  - ▶ il livello **Media** fornisce principalmente strumenti per lo sviluppo di grafica, per la gestione dell'audio e per la gestione del video;
  - ▶ al livello più alto troviamo lo strato **Cocoa Touch** che si compone dei framework **UIKit** e **foundation** che rendono disponibili gli strumenti necessari per la realizzazione della grafica, la visualizzazione e l'interazione con l'utente, comprese le classi legate alla gestione dei dati.

L'accesso da parte degli sviluppatori ai livelli inferiori resta comunque garantito.

Tra i punti deboli dei dispositivi **Apple**, oltre al prezzo decisamente elevato, c'è il fatto che è un "sistema chiuso": questo punto, però, è anche un fattore di forza, dato che permette al dispositivo di essere molto performante, pur limitando la libertà dell'utente che non riesce neppure a cambiare le icone oppure il tema degli sfondi.

## Android

**Android** è un sistema operativo distribuito gratuitamente da **Google** nel suo codice sorgente: chiunque può prenderlo, modificarlo e metterlo sul suo dispositivo da proporre al mercato.

La filosofia di **Android** è diametralmente opposta a quella di **Apple**, decisamente aperta: venne realizzato con l'obiettivo di essere il primo sistema operativo completo di tipo aperto, creato specificamente per i dispositivi mobili e distribuito gratuitamente.

Inizialmente sviluppato da una start-up, **Android Inc.**, venne acquistato da Google nel 2005, si formò un Consorzio di Aziende chiamato **Open Handset Alliance** e nel settembre del 2008 venne rilasciata la sua prima versione, **Android 1.0**.



**Zoom su...**

### OPEN HANDSET ALLIANCE

Oggi il Consorzio è costituito da molte decine di membri tra i quali troviamo:

- ▶ operatori telefonici come Vodafone, T-mobile, Telecom Italia;
- ▶ produttori di dispositivi mobili come Motorola, Samsung;
- ▶ produttori di semiconduttori come Intel, Texas Instruments;
- ▶ oltre che a compagnie di sviluppo e di commercializzazione software.

Per poter essere completamente **open** fu scelto come **kernel** del sistema operativo il **kernel** di **Linux**, sistema operativo **open** per antonomasia, e proprio per questa caratteristica di apertura consente la portabilità di **Android** su differenti piattaforme hardware e si presta ad essere modificato e personalizzato in modo da avere una maggiore integrazione con l'hardware.

Queste sono alcune delle motivazioni che hanno spinto grandi produttori di elettronica come **LG**, **Sony**, **Samsung**, **Huawei** ecc. a sceglierlo per i loro smartphone.

Anche l'architettura di **Android** è strutturata a quattro livelli e viene descritta nella prossima lezione, dato che in questa unità didattica viene anche descritto un ambiente di sviluppo per realizzare applicazioni per questo sistema operativo.

Abbiamo scelto di sviluppare app per **Android** dato che copre circa il 50% del mercato.

## Windows Phone

I terminali **Windows** detengono il terzo posto nel numero di cellulari venduti; il **SO** mobile di **Microsoft** è un sistema chiuso, a pagamento, che però può essere installato su diverse marche di cellulari.

L'aspetto più interessante sta comunque nel fatto che i nuovi sistemi operativi di **Microsoft** sono simili (a partire da **Windows 8**) tra dispositivi fissi e mobili e, quindi, un programma sviluppato per desktop funziona anche su cellulare.

Ad esempio, nella versione 10 è stata aggiunta la funzione **Continuum** che permette di usare il telefono su un secondo schermo, come la TV o un monitor: è possibile collegare un mouse e una tastiera per usare il telefono come il PC, mentre lo schermo del telefono rimane libero per essere usato come telefono.

Lo stesso discorso vale per gli applicativi **Office**: gli stessi strumenti di produttività individuale offerti sul **PC** sono disponibili nel **telefono** e nel **tablet**.

Il vero grande problema di **Windows** su cellulari è la scarsa disponibilità di applicazioni dato che la diffusione è ad oggi ancora modesta e anche le più comuni applicazioni (**Twitter**, **Facebook**, **Whatsapp**) vengono aggiornate pochissimo.

## AREA digitale



I sistemi operativi Palm OS, Symbian e Linux

## ■ Ambienti di sviluppo per dispositivi mobili

Per i sistemi operativi sopra descritti riportiamo di seguito le principali caratteristiche dei più utilizzati ambienti di sviluppo per il software applicativo.

### SDK per iOS

Il primo **Software Development Kit** (SDK), messo a disposizione da **Apple** per gli sviluppatori di terze parti, fu annunciato da **Steve Jobs** in persona il 17 ottobre 2007 ed è stato rilasciato il 6 marzo 2008 col nome di **Xcode**: permette agli sviluppatori di creare applicazioni per **iPhone** e **iPod touch** e testarle in un simulatore di **iPhone** (**iPhone Simulator**).

Unico inconveniente è che per poter caricare una applicazione nei dispositivi è necessario iscriversi (a pagamento) all'**iPhone Developer Program**: il prezzo delle applicazioni è stabilito dal programmatore ma l'unico canale di vendita è **App Store**, e **Apple** si trattiene per questo servizio il 30% del prezzo richiesto. Tuttavia i programmatori possono anche optare per rilasciare l'applicazione gratis: in tale caso non è previsto nessun costo di rilascio o distribuzione.

Un vantaggio dell'ambiente di sviluppo **Xcode** è che consente lo sviluppo di applicazioni per **Mac OS X** e come linguaggi di programmazione supporta l'**Objective C**, l'**AppleScript**, il **C++**, l'**Objective C++** e **Java**.

Forse la caratteristica più significativa di **Xcode** è che supporta la distribuzione in rete del lavoro di compilazione tramite **Bonjour** e **Xgrid**: è possibile compilare un progetto su più computer riducendo i tempi e inoltre la compilazione è di tipo incrementale, cioè il codice viene compilato **mentre viene scritto**, ottimizzando i tempi di compilazione.

L'ambiente **Cocoa**, con la sua suite integrata di componenti software orientate agli oggetti che permette di creare rapidamente software ricchi di caratteristiche, può anche essere utilizzato con altri ambienti di sviluppo che supportano linguaggi, come **Perl**, **Python** (mediante il bridge **PyObjC**) e **Ruby** (tramite **RubyCocoa**).

Le classi **Cocoa** sono ricche di funzionalità, coprono praticamente tutte le necessità di ogni sviluppatore e permettono a terze parti di ottenere uniformità tra le applicazioni sviluppate, soprattutto per quanto riguarda l'approccio alle interfacce grafiche.

## AREA digitale



Set che compongono Xcode

## Java per Android

**Android** mette a disposizione degli sviluppatori il **Software Development Kit (SDK)**, che è un insieme di tool di programmazione che contiene al suo interno le librerie relative alle varie versioni della piattaforma **Android**; inoltre ha una propria macchina virtuale, in modo da eseguire il codice indipendentemente dalle altre applicazioni che sono in esecuzione sulla macchina dello sviluppatore.

Le applicazioni per i dispositivi che utilizzano come sistema operativo **Android** sono scritte in **Java** e, sostanzialmente, vengono realizzate in due ambienti di sviluppo:

- 1 **Eclipse**: è stato il primo ed è il più diffuso prodotto molto utilizzato per il mondo **Java** ed è caratterizzato dal fatto di avere a disposizione molteplici plug-in, cioè componenti software ideati per specifici scopi che rendono possibile l'ampliamento e la personalizzazione di un generico programma da parte di terzi (ricordiamo **ADT Android Development Tools** che contiene tutte le classi e le librerie necessarie per poter sviluppare applicazioni **Android**).
- 2 **Android Studio** è il più recente ambiente di sviluppo integrato (IDE) che sta diventando una soluzione sempre più popolare in quanto offre diversi vantaggi rispetto a **Eclipse**, ad esempio quello di essere più semplice, ed inoltre il fatto che a breve diverrà l'unico ambiente di sviluppo ufficialmente supportato da parte di **Google**.

Dopo averli provati entrambi la nostra scelta è stata quella di **Android Studio**, sia per motivi di performance, in quanto riduce drasticamente i tempi di progettazione e di testing, sia per portabilità, dato che è compatibile con tutte le piattaforme più utilizzate, e cioè **Windows**, **Mac OS X**, **Linux**.

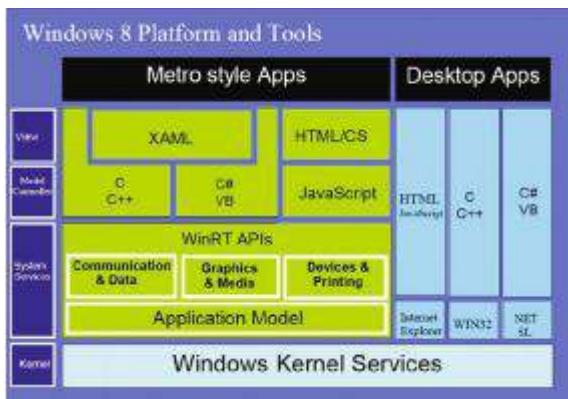
Nella sezione dedicata all'attività laboratoriale sono disponibili alcune lezioni introduttive che permettono di sviluppare applicazioni complete.

## Apps per Windows mobile

Con il rilascio di **Windows 10** il colosso di **Redmond** ha realizzato il concetto di “One Windows”, cioè lo sviluppo di un sistema operativo in grado di funzionare allo stesso modo su dispositivi completamente diversi l'uno dall'altro: dai tradizionali desktop e notebook, fino ad arrivare a tablet e smartphone.

La piattaforma **UWP (Universal Windows Platform)** è stata introdotta con **Windows 8** ma solo con **Windows 10** è disponibile per tutte le famiglie di dispositivi: è possibile realizzare **App** che “girano” perfettamente in tutti i dispositivi con un solo set di **API** (app universali distribuite in un pacchetto con estensione **Appx**).

L'architettura Runtime di **Windows 10** è la stessa di **Windows 8**.



Un'app universale è legata alla piattaforma tramite Metro che nella versione 10 prende il nome di **Apps Store**: è gestito da uno strato di esecuzione chiamato **Windows Runtime o WinRT**.

Apps Store supporta quattro metodologie di sviluppo, già note agli utenti di **.NET**: ai linguaggi **C #**, **Visual Basic** o **C ++** è possibile integrare **HTML**, **JavaScript** o programmazione grafica accelerata usando **DirectX**.

**Microsoft Visual Studio 2015** fornisce un modello di **app UWP** per ogni linguaggio e, una volta completato il progetto, permette di generare un pacchetto dell'app e inviarlo direttamente a

**Windows Store** per rendere la app disponibile pubblicamente in modo che possa essere installata in qualsiasi dispositivo **Windows 10**.



## Zoom su...

### WINDOWS APP STUDIO

Microsoft ha realizzato anche uno strumento **web based** destinato ai "non programmati": **Windows App Studio**, che consente a chiunque di sviluppare **app** per **Windows 10**. Per creare un'applicazione non è necessario conoscere un linguaggio di programmazione, ma è sufficiente scegliere tra i modelli di **app** proposti e personalizzarne il funzionamento.

L'app realizzata può essere scaricata e provata immediatamente sui propri dispositivi, inoltre i programmati possono scaricare il sorgente **Visual Studio** per portare eventuali modifiche e personalizzare a proprio piacimento l'applicazione.

Un ulteriore punto a favore di **App Studio** consiste nel fatto che le applicazioni realizzate sono compatibili anche con **Windows 8.1** e **Windows Phone 8.1**.



What will 5G bring to you?	What's new with 5G?	What will 5G bring to you?	What's new with 5G?
amazing volume amazingly fast	spectrum extension; millimetre waves; cell densification; increase spectrum efficiency; advanced antennas; 3D beam-forming techniques; new electronic components; back-haul optimization; D2D; moving networks (vehicle based cells)	massive amount of connected things & people	new waveform; cell densification; much less signalling traffic & no synchronisation; RAN architecture
always best connected	combination of 4G, 3G, Wi-Fi, & new radio access to create an integrated & dynamic radio access network; connectivity management mechanisms	energy efficiency	millimetre waves for front-haul & backhaul; new operation mechanisms for dense networks; pooling of base station processing on-demand consumption; massive machine communication, power amplifiers; DSP (digital signal processing) – enabled optical transceivers; harvesting ambient energy; optimization of sleep mode switching
no perceived delay	ultra-low latency; software-defined networks; decoupling functional architecture from the underlying physical infrastructure; network intelligence closer to users; MEC (mobile edge computing), D2D	flexible programmable networks	software-defined networks; network function virtualisation; decoupling functional architecture from the underlying physical infrastructure, APIs

## Verifichiamo le conoscenze



### 1. Risposta multipla

**1** Le reti mobili in base ai tipi di mobilità possono essere di classe:

- a. access mobility
- b. terminal mobility
- c. personal mobility
- d. wireless mobility

**2** Nel caso di access mobility esiste una relazione fissa:

- a. tra l'utente e il proprio terminale
- b. tra l'utente e la propria SIM
- c. tra il terminale e la rete fissa
- d. tra la SIM e l'utente

**3** Nel caso di terminal mobility esiste una relazione fissa:

- a. tra l'utente e il proprio terminale
- b. tra l'utente e la propria SIM
- c. tra il terminale e la sua SIM
- d. tra la SIM e la rete dell'utente

**4** Nel caso di personal mobility esiste una relazione fissa:

- a. tra l'utente e il proprio terminale
- b. tra l'utente e la propria SIM
- c. tra il terminale e la rete fissa
- d. tra la SIM e la rete dell'utente

**5** Nei sistemi personal mobility l'acronimo PCS sta a indicare:

- a. Personal Communications System
- b. Personal Computing Services
- c. Personal Computing System
- d. Personal Communications Services

**6** La trasmissione dei dati digitali avviene a partire dalla:

- a. 1<sup>a</sup> generazione
- b. 2<sup>a</sup> generazione
- c. 2,5<sup>a</sup> generazione
- d. 3<sup>a</sup> generazione

**7** Esistono sostanzialmente quattro tipologie di dispositivi mobili:

- a. Pocket PC
- b. Tablet PC
- c. Phone PC
- d. Smartphone
- e. UMPC (Ultra Mobile PC)

**8** Il sistema operativo iOS è strutturato in quattro livelli di astrazione:

- a. il Core OS layer
- b. il Core Services layer
- c. il Touch layer
- d. il Cocoa Touch layer



### 2. Vero o falso

**1** Reti mobili e reti wireless sono sinonimi.

V F

**2** I sistemi access mobility sono tipicamente indoor con raggio di azione inferiore ai 500 m.

V F

**3** A partire dalla III generazione inizia la diffusione del sistema UMTS.

V F

**4** Nel nostro paese la rete 4G ha una velocità di connessione di 1,5 Mbps.

V F

**5** La 5G porterà la velocità massima teorica di navigazione con smartphone e tablet a 20 Gbps.

V F

**6** iOS oltre che per iPhone e ora è utilizzato anche negli iPod, iPad e Apple TV.

V F

**7** Il Cocoa Touch layer è uno dei quattro livelli di astrazione di Android.

V F

**8** Il primo SDK messo a disposizione da Apple fu annunciato nel 2010 da Steve Jobs.

V F

**9** Eclipse è stato il primo ed è il più diffuso prodotto utilizzato per il mondo Java.

V F

**10** La piattaforma UWP (Universal Windows Platform) è stata introdotta con Windows 7.

V F

# Android: un sistema operativo per applicazioni mobili

In questa lezione impareremo...

- il concetto di applicazione
- ad analizzare i vari componenti
- il ciclo di vita di un'activity

## ■ Android



Android è un sistema operativo per dispositivi mobili sviluppato inizialmente da **Android Inc.** e acquisito da **Google**, il gigante di Internet, nel 2005.

Fondamentalmente **Android** non è stato sviluppato da zero, è infatti un sistema operativo che si basa su diverse versioni del **kernel Linux**: ciò contraddistingue questo sistema operativo dagli altri per la sua natura open source e per la sua versatilità, infatti può funzionare su qualsiasi dispositivo mobile.

Il primo smartphone dotato di piattaforma **Android** è stato l'**HTC Dream**, presentato il 22 ottobre del 2008.



► Il **kernel** costituisce il nucleo di un sistema operativo e possiede il compito di fornire l'accesso all'hardware ai processi in esecuzione. ►

### AREA digitale

Versioni di Android

Anche se è stato progettato principalmente per **smartphone** e **tablet**, ha avuto diffusione anche con interfacce utente specializzate per televisori (**Android TV**), automobili (**Android Auto**), orologi da polso (**Android Wear**), occhiali (**Google Glass**) e altri.



Android TV



Android Auto

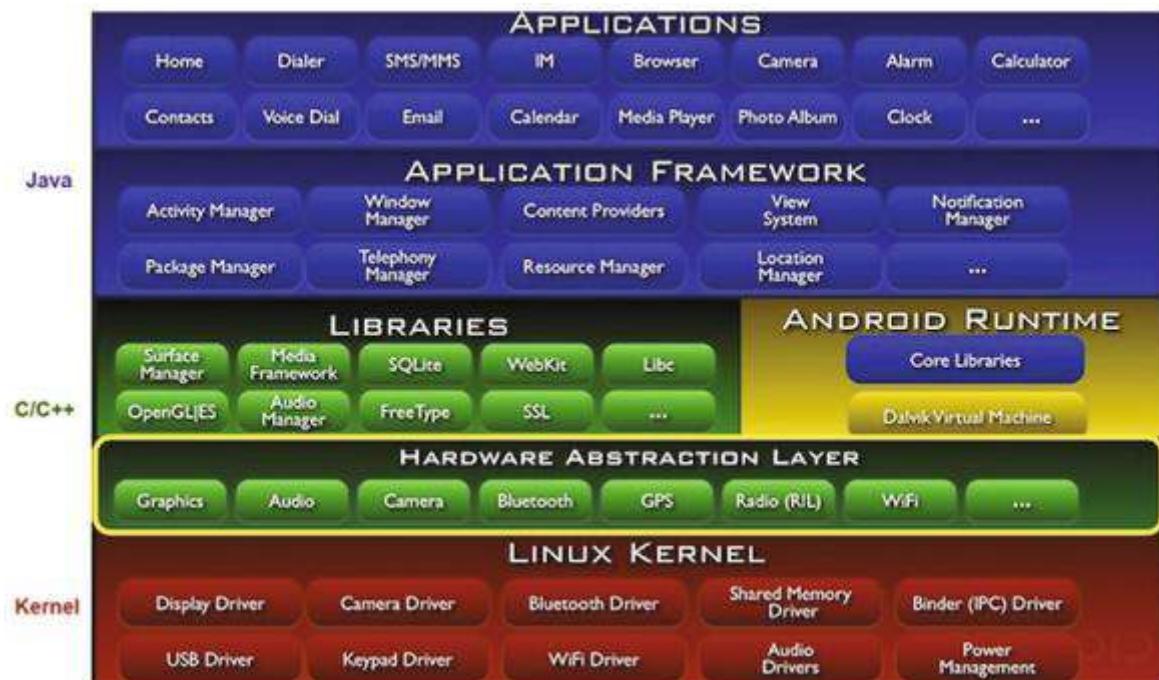


Android Wear



Google Glass

Il **sistema operativo** è costituito da uno stack software (ovvero un set di sottosistemi software) che possiamo individuare nel seguente schema:



Architettura di Android

Possiamo individuare quattro strati nell'**Android Stack**:

- ▷ **Linux Kernel Layer**;
- ▷ **Native Layer**;
- ▷ **Application Framework Layer**;
- ▷ **Applications layer**.

Il **kernel di sistema operativo** basato su Linux è composto da applicazioni **Java** che vengono eseguite su uno speciale **framework applicativo**, scritto anch'esso in **Java** e orientato agli oggetti che comprende librerie compatibili basate su **Apache Harmony** sul quale si “appoggiano” le **applications**, un insieme di funzioni base messe a disposizione degli sviluppatori per effettuare telefonate, mandare SMS, interagire con Internet ecc.; il sistema si completa con un **Hardware Abstraction Layer (HAL)**, quello che viene indicato come il **middleware**, che comprende anche le **librerie** e le **API** scritte in C.



L'**Hardware Abstraction Layer (HAL)** svolge un ruolo determinante che ha contribuito al successo di **Android** in quanto comprende un insieme di funzioni che tengono conto di tutte le differenze fra dispositivi fisici diversi, realizzando uno shell tale da virtualizzare l'hardware e permettere allo stesso software di funzionare su dispositivi differenti. In tal modo, cioè con la presenza di uno strato **HAL**, si rende un dispositivo compatibile agli altri ed eventuali modifiche dell'hardware non richiedono aggiornamenti al software, ma solo adattamenti dello strato di interfacciamento.

L'utilizzo di questo strato software aggiuntivo consente di ottenere un elevato *livello di servizio* per gli utenti e un elevato *livello di astrazione* per i programmatore.

Le applicazioni utente vengono eseguite tramite la **Dalvik Virtual Machine (DVM)**, una macchina virtuale adattata per l'uso su dispositivi mobili e dotata di **compilatore just-in-time (JIT)**. Dalla versione 5.0 questa virtual machine è stata rimpiazzata con **ART (Android RunTime)** integrato in **Android 5.0 Lollipop**.



## Zoom su...

### COMPILATORE JUST-IN-TIME

Un compilatore just-in-time permette un tipo di compilazione, conosciuta anche come traduzione dinamica, con la quale è possibile aumentare le performance dei sistemi di programmazione che utilizzano il bytecode, traducendo il bytecode nel codice macchina nativo in fase di runtime. L'obiettivo finale dei sistemi **JIT** è di combinare i vantaggi della compilazione del bytecode con quelli della compilazione nativa, aumentando le prestazioni quasi al pari di una compilazione direttamente in linguaggio macchina.

Ricordiamo, a titolo di curiosità, che il sistema operativo **Android** è composto in tutto da 12 milioni di righe di codice che comprendono 3 milioni di righe di XML, 2.8 milioni di righe di C, 2.1 milioni di righe di Java e oltre 1.75 milioni di righe di C++.

## ■ La struttura di un'applicazione Android

Esistono numerose **applicazioni**, chiamate **app**, sviluppate per questa piattaforma e, probabilmente, molte ancora compariranno in tempi futuri.

Si tratta di programmi eseguiti dall'utente tramite l'interfaccia grafica del terminale **Android**. Le **app** sono di tipo **Event Driven**, cioè guidate dagli eventi gestiti all'interno del dispositivo mobile, come ad esempio il **touch** dello schermo, le azioni dei sensori ecc.

Possono essere di due tipi:

- **Applicazioni** vere e proprie, che occupano tutto lo schermo principale come per esempio il browser web standard di **Android**;
- **Widget** che occupano una piccola e fissata porzione dello schermo principale come per esempio l'orologio standard di **Android**.

**Android** è fornito di una serie di *applicazioni preinstallate* tra le quali un browser, basato su **WebKit**, una rubrica e un calendario: quando si vuole aggiungere una funzionalità non presente, come ad esempio un software per l'ufficio, un videogioco o una immagine di sfondo, si usa ricercarla nel **Market** (come **Play Store**) e "installarla", ossia copiarla all'interno del dispositivo affinché sia sempre presente e utilizzabile.

### AREA digitale



I diversi tipi di tocco su display touch

Le applicazioni **Android** sono composte da quattro componenti fondamentali:

- **Activity**;
- **Service**;
- **Broadcast Receiver**;
- **Content Provider**.

Le applicazioni sono formate da uno o più di questi elementi, tuttavia ciascuna app deve contenere almeno una **activity**.

## Activity

Le **activity** sono l'elemento fondamentale delle applicazioni, rappresentano il blocco di codice che interagiscono con l'utente utilizzando lo schermo e i dispositivi di input messi a disposizione dal dispositivo. Utilizzano componenti **GUI**, come ad esempio **pulsanti**, **caselle di testo**, **pulsanti radio** ecc., presenti nel package `android.widget`. Le activity sono probabilmente il modello più diffuso in **Android**, e vengono create ereditando la classe `android.app.Activity`.



## Service

I **service** sono programmi che vengono eseguiti in background e non interagiscono direttamente con l'utente. Un servizio può ad esempio riprodurre un brano **mp3**, oppure leggere segnali dai sensori **GPS**, mentre l'utente utilizza delle activity per fare altre operazioni. Un servizio si realizza estendendo la classe `android.app.Service`.

## Broadcast Receiver

Un **Broadcast Receiver** viene utilizzato quando si deve intercettare un particolare **evento** di sistema, come ad esempio quando si scatta una foto o quando parte la segnalazione di batteria scarica. La classe da estendere è `android.content.BroadcastReceiver`.

## Content Provider

I **Content Provider** sono utilizzati per esporre dati e informazioni. Costituiscono un canale di comunicazione tra le differenti applicazioni installate nel sistema. Si può creare un Content Provider estendendo la classe astratta `android.content.ContentProvider`.

## Intent

È doveroso ricordare che ciascuno di questi può attivarne un altro effettuando apposite invocazioni di sistema: questa operazione, detta **intenzione**, viene codificata con un **Intent** utilizzabile come normale classe Java ma che sottintende un potentissimo strumento di comunicazione di **Android**.



**Zoom su...**

### ACTIVITY E APPLICAZIONI

Spesso questi due concetti vengono confusi. In generale:

- **Activity**: sono associate a una singola e ben precisa attività che l'utente può svolgere, come ad esempio selezionare una data da un **Time Picker**, oppure inserire un nome in una casella di testo.
- **Applicazione**: contengono delle activity, oltre ad altri elementi, come ad esempio l'applicazione Appunti possiede una specifica activity per modificare una nota oppure una activity per gestire la data di un appuntamento.

## ■ Il ciclo di vita di una activity

I **dispositivi mobili** per i quali è stato creato il sistema operativo **Android** non possiedono schermi capienti come quelli dei PC, per cui le finestre di esecuzione dei programmi non possono essere affiancate o sovrapposte. Le **activity Android** sono una sorta di programma in esecuzione con delle caratteristiche peculiari, tra cui quella di essere esclusivamente proprietarie del display.

In generale una **activity** passa attraverso i seguenti stati:

- **RESUMED** o **ACTIVE** o **RUNNING**: è **visibile** ed è in grado di ricevere dati in input;
- **PAUSED**: è parzialmente **visibile**, non riceve nessun input;
- **STOPPED**: **non è visibile**, tuttavia ancora in esecuzione;
- **DESTROYED**: è **rimossa** dalla memoria del dispositivo mobile.

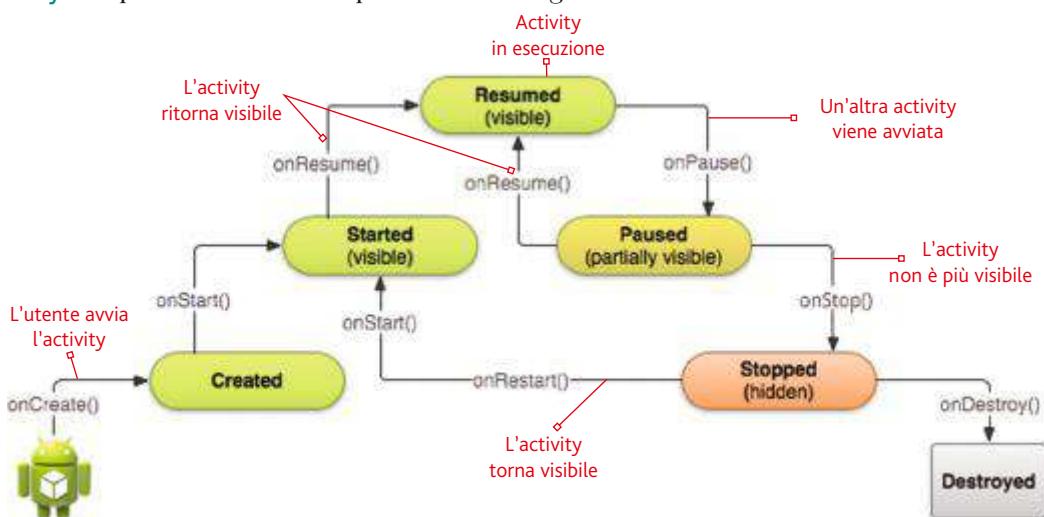
Possiamo eseguire più attività simultaneamente, ma solo una potrà prendere il controllo del display che, come abbiamo detto prima, è una risorsa esclusiva. L'attività che occupa il display è **in esecuzione** e interagisce direttamente con l'utente; altre attività ancora in esecuzione vengono, per così dire, **ibernata** in **background**, per ridurre al minimo il consumo delle risorse del sistema, in questo caso della risorsa elaborazione da parte del microprocessore. L'utente può ripristinare un'attività ibernata e riprenderla da dove l'aveva interrotta, riportandola in primo piano: l'attività dalla quale si sta allontanando verrà ibernata e mandata in sottofondo al posto di quella ripristinata.

Il cambio di attività può anche avvenire a causa di un **evento esterno**, come ad esempio nel caso di una telefonata in arrivo: se il telefono squilla mentre si sta usando la calcolatrice, quest'ultima sarà automaticamente **ibernata** e posta in **background** e l'utente, conclusa la chiamata, potrà richiamare l'attività interrotta e riportarla in vita, riprendendo i calcoli esattamente da dove li aveva interrotti (**resumed**).

Le attività **ibernate** non appesantiscono il microprocessore, in quanto non richiedono elaborazione. Pertanto il sistema operativo non mostra all'utente le attività ibernate. Le attività di Android infatti non mettono a disposizione un pulsante di chiusura con cui terminarne l'esecuzione; l'utente può pertanto solo mandare un'attività in ibernazione.

La rimozione di una attività può avvenire spontaneamente, perché ha terminato il suo compito, oppure per iniziativa del sistema operativo che decide che non è più utile oppure non ha più memoria a disposizione.

Vediamo quali sono i principali **passaggi di stato** di un'attività analizzando alcuni **metodi** della classe **Activity** che possiamo ridefinire per intercettare gli eventi di nostro interesse:



Analizziamo i principali **metodi** della classe **Activity** in modo dettagliato:

- ▶ **protected void onCreate (android.os.Bundle savedInstanceState):** viene richiamato alla creazione dell'attività, l'argomento **savedInstanceState** restituisce al metodo un eventuale stato dell'attività passato da un'altra istanza che è stata terminata, l'argomento vale **null** qualora non vi sia alcuno stato precedentemente salvato;
- ▶ **protected void onRestart():** viene richiamato per segnalare che l'attività è stata riavviata dopo essere stata precedentemente arrestata;
- ▶ **protected void onStart():** viene richiamato per segnalare che l'attività viene resa visibile sullo schermo;
- ▶ **protected void onResume():** viene richiamato per segnalare che l'attività inizia l'interazione con l'utente;
- ▶ **protected void onPause():** viene richiamato per segnalare che l'attività termina l'interazione con l'utente;
- ▶ **protected void onStop():** viene richiamato per segnalare che l'attività non è più visibile sullo schermo;
- ▶ **protected void onDestroy():** viene richiamato per segnalare che l'applicazione è stata terminata.

Per poter modificare il codice di questi metodi dobbiamo eseguire un **override** del metodo della classe madre, prestando sempre attenzione ad inserire, nella prima riga di codice di ciascuno di questi metodi, il costruttore della classe base che stiamo ridefinendo attraverso l'operatore **super**.

```

>MainActivity.java ×
package com.example.emanuele.myapplication;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}

```

## ■ Il file APK

Le applicazioni software vengono distribuite sotto forma di pacchetto autoinstallante in un file con estensione **.APK** (**Android Package**), che è un file compresso simile al formato **JAR** contenente il software (file con estensione **.dex**), le sue risorse (immagini, suoni ecc.) e alcuni file **XML**.

All'interno di questo file c'è anche un **certificato** che permette l'installazione di un pacchetto **.APK** su un dispositivo **Android**: il certificato deve essere presente in qualsiasi pacchetto, altrimenti **Android** non installerà l'applicazione al suo interno.

Il certificato viene creato dallo sviluppatore dell'applicazione che può scegliere se crearne uno di “debugging” (quindi a uso interno) oppure uno di “mercato” (per la distribuzione): in questo caso può anche decidere la tipologia della sua diffusione delle copie, cioè se libera oppure limitata. Il **distributore** ci aggiungerà poi una sua chiave, che potrà successivamente revocare, se necessario, e in caso di revoca l'applicazione non sarà più installabile né eseguibile in nessun dispositivo **Android**. È anche possibile realizzare e distribuire direttamente il pacchetto **.APK** senza passare dai distributori; in questo caso lo sviluppatore si auto-crea il certificato e l'utente, all'atto della installazione di tale applicazione, riceverà un avviso che sta installando un software di questo tipo, cioè **self-signed** (quindi non certificato e l'installazione è ... “*a suo rischio e pericolo*”).

### CLIL Android 7.0 Nougat

Android 7.0 brings new features for performance, productivity, and security. Test your apps with new system behaviors to save power and memory. Take advantage of multi-window UI, direct reply notifications, and more.

### Multi-window Support

In Android 7.0, we're introducing a new and much-requested multitasking feature into the platform — multi-window support. Users can now pop open two apps on the screen at once.

On phones and tablets running Android 7.0, users can run two apps side-by-side or one-above-the-other in split-screen mode. Users can resize the apps by dragging the divider between them.

On Android TV devices, apps can put themselves in picture-in-picture mode, allowing them to continue showing content while the user browses or interacts with other apps.

Especially on tablets and other larger-screen devices, multi-window support gives you new ways to engage users. You can even enable drag-and-drop in your app to let users conveniently drag content to or from your app — a great way to enhance your user experience. It's straightforward to add multi-window support to your app and configure how it handles multi-window display. For example, you can specify your activity's minimum allowable dimensions, preventing users from resizing the activity below that size. You can also disable multi-window display for your app, which ensures that the system will only show your app in full-screen mode.

### Notification Enhancements

In Android 7.0 we've redesigned notifications to make

them easier and faster to use. Some of the changes include:

- ▶ **Template updates:** We're updating notification templates to put a new emphasis on hero image and avatar. Developers will be able to take advantage of the new templates with minimal adjustments in their code.
- ▶ **Messaging style customization:** You can customize more of the user interface labels associated with your notifications using the `MessagingStyle` class. You can configure the message, conversation title, and content view.
- ▶ **Bundled notifications:** The system can group messages together, for example by message topic, and display the group. A user can take actions, such as Dismiss or Archive, on them. If you've implemented notifications for Android Wear, you'll already be familiar with this feature.
- ▶ **Direct reply:** For real-time communication apps, the Android system supports inline replies so that users can quickly respond to an SMS or text message directly within the notification interface.
- ▶ **Custom views:** Two new APIs enable you to leverage system decorations, such as notification headers and actions, when using custom views in notifications.



### Quick Path to App Install

One of the most tangible benefits of ART's JIT compiler is the speed of app installs and system updates. Even large apps that required several minutes to optimize and install in Android 6.0 can now install in just a matter of seconds. System updates are also faster, since there's no more optimizing step.

To learn more about the consumer features of Android 7.0, visit [www.android.com](http://www.android.com)

## Verifichiamo le conoscenze



### 1. Risposta multipla

**1** Android è un sistema operativo:

- a. sviluppato nel 2005
- b. specifico per smartphone
- c. si basa su Linux
- d. scritto da Google Inc

**2** Il kernel di Android ha:

- a. la Java Virtual Machine (JVM)
- b. la Dalvik Virtual Machine (DVM)
- c. la Symbian Virtual Machine (SVM)
- d. la Linux Virtual Machine (LVM)

**3** Android ha una parte scritta in:

- a. C
- b. C++
- c. Java
- d. Python

**4** Le app sono di tipo:

- a. procedurale
- b. sequenziale

- c. event driven
- d. iterativo

**5** Le applicazioni Android sono composte da quattro componenti fondamentali:

- a. Activity
- b. Service
- c. Intent
- d. Broadcast receiver
- e. Content provider

**6** Una activity tra i suoi stati NON ha lo stato:

- a. created
- b. resumed
- c. slept
- d. stopped
- e. started
- f. paused
- g. destroyed



### 2. Vero o falso

**1** Le applicazioni sono formate almeno una activity.

V F

**2** I service sono programmi che vengono eseguiti in foreground.

V F

**3** Un Broadcast Receiver è in grado di intercettare un particolare evento di sistema.

V F

**4** I Content Provider sono utilizzati per esporre dati e informazioni.

V F

**5** Una activity è composta da una o più applicazioni.

V F

**6** Nelle app le finestre di esecuzione non possono essere affiancate o sovrapposte.

V F

**7** Le attività ibernante appesantiscono il microprocessore.

V F

**8** onStart si verifica quando l'activity diviene visibile all'utente.

V F

**9** onResume si verifica quando l'activity riprende l'esecuzione.

V F

**10** onStop si verifica quando un'activity è terminata.

V F

**11** onRestart si verifica quando l'activity viene arrestata e riavviata.

V F

**12** Per la APK esistono certificati self-signed.

V F

# ESERCITAZIONI DI LABORATORIO 1

## ANDROID STUDIO: INSTALLAZIONE E CONFIGURAZIONE



◀ IDE è l'acronimo di **Integrated Development Environment**, che significa ambiente di sviluppo integrato. Si tratta in sintesi di un software che aiuta i programmatore nello sviluppo del codice mettendo a disposizione del programmatore un **editor** di codice sorgente, un **compilatore** o un **interprete**, un **tool** di **building** automatico, e un **debugger**. ▶

Per installare la **JDK** dobbiamo collegarci al sito: <http://www.oracle.com/technetwork/java/javase/downloads/>. Prima dobbiamo selezionare la versione di **Java JDK** e quindi il sistema operativo desiderato. A volte può essere necessario dover inserire manualmente il percorso d'installazione della **JDK**, mediante le variabili d'ambiente per Windows: per fare questo apriamo il pannello di controllo, selezioniamo l'icona **Sistema**, quindi **Impostazioni di sistema Avanzate**, scheda **Avanzate** e infine **Variabili Ambiente**.

Durante l'installazione conviene assegnare alla directory di **JDK** un nome facile da ricordare (ad esempio **JAVA2**).

All'interno della sezione variabili di sistema, aggiungiamo una nuova variabile con il nome **JAVA\_HOME** e come percorso, il percorso nel quale abbiamo installato **Java JDK** (ad esempio **C:\Program Files\Java\JAVA2**):

### ■ Android Studio

**Android Studio** è un ambiente di sviluppo (◀ IDE ▶) progettato specificamente per lo sviluppo per la piattaforma **Android**.

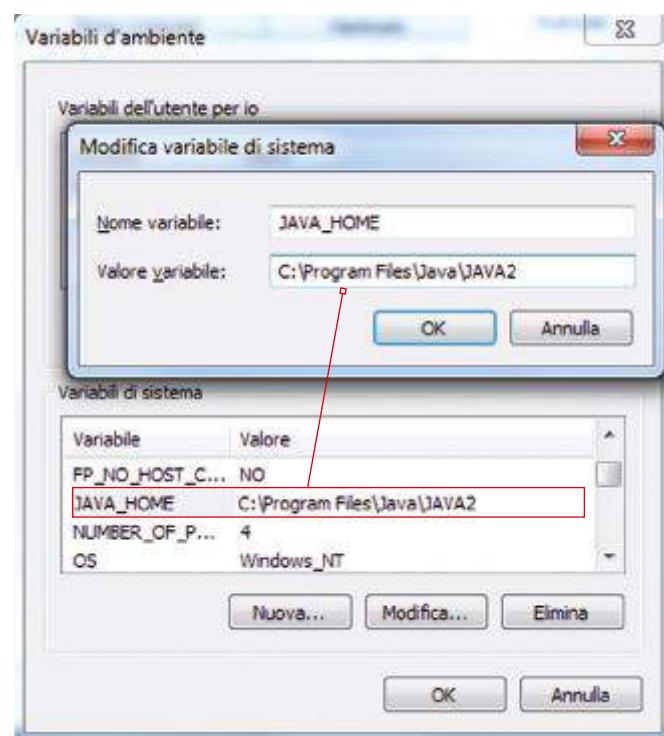
Prima di utilizzare l'ambiente di sviluppo **Android Studio**, dobbiamo installare le librerie e i compilatori Java (**JDK – Java Development Kit**).



Durante l'installazione di **Android Studio** vengono anche installate le librerie ◀ **Android SDK** ▶, necessarie per lo sviluppo di applicazioni native: anche se l'installazione avviene in contemporanea con **Android Studio**, successivamente queste devono essere configurate separatamente.



◀ **Android SDK** è l'insieme delle **librerie** necessarie alla gestione dei dispositivi Android e comprende i componenti necessari per lo sviluppo degli applicativi, come ad esempio i **driver**. ▶



## ■ Scarcare e installare Android Studio

Seguiamo la procedura descritta di seguito per scaricare e installare **Android Studio**.

- 1 Prima di tutto è necessario collegarsi al sito dal quale scaricare **Android Studio** e le relative librerie: [developer.android.com/sdk](http://developer.android.com/sdk).
- 2 Selezioniamo il pacchetto per il sistema operativo che stiamo utilizzando sul nostro PC (nel nostro caso Windows) e quindi clicchiamo su **Download Android Studio for Windows**.
- 3 Dopo aver letto e confermato le condizioni di utilizzo proposte, clicchiamo sul pulsante **Download Android Studio** per iniziare a scaricare il pacchetto.
- 4 Una volta terminato il download ci posizioniamo nella cartella **Download**, a questo punto facciamo doppio clic sul file eseguibile, indicato nella finestra seguente, per iniziare la procedura di **installazione**.

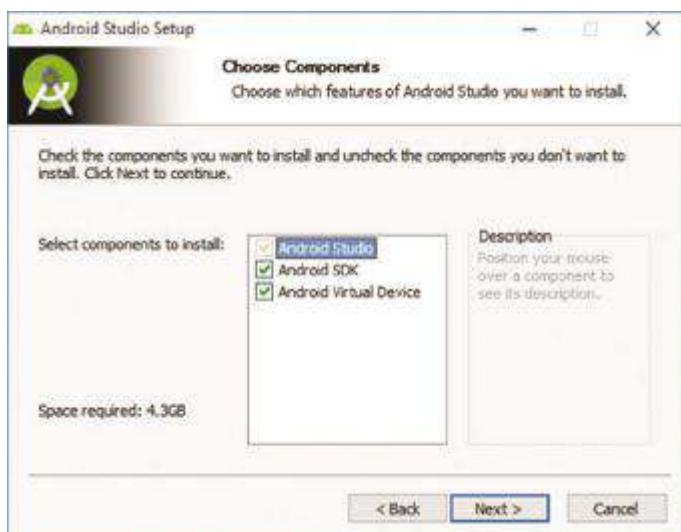


- 5 Dopo aver cliccato sul pulsante [Next] nella finestra di Welcome ora dobbiamo scegliere quali componenti installare: automaticamente verranno installati anche l'Android SDK e l'**AVD Manager**.



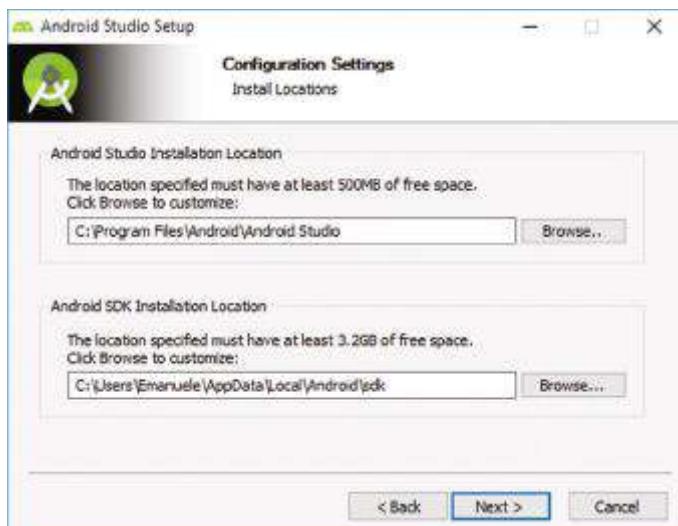
◀ **AVD Manager** è un componente aggiuntivo che permette di creare e gestire degli emulatori Android. AVD è infatti l'acronimo di **Android Virtual Device** e grazie a esso possiamo infatti testare e fare il debug di applicazioni **Android** senza aver bisogno di un dispositivo fisico. ►

Se abbiamo già installato sia **Android SDK** che **AVD Manager**, dobbiamo togliere il segno di spunta sulle caselle di scelta e passare all'inserimento manuale dei percorsi di installazione.

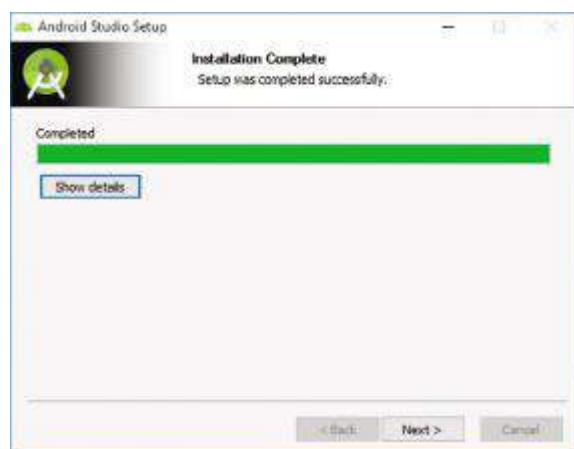


Nel nostro caso i componenti aggiuntivi non sono installati pertanto lasciamo le impostazioni suggerite dal programma e clicchiamo su **[Next]** per proseguire.

- 6 Dopo aver letto e accettato la licenza d'uso di **Android SDK** confermando con **I Agree**, dobbiamo scegliere i percorsi di installazione dei diversi componenti. Accettiamo il percorso proposto. Ricordiamoci di annotare i percorsi di installazione per modifiche future. ►
- 7 Decidiamo se creare un collegamento sul **Desktop** per avviare velocemente **Android Studio**, quindi clicchiamo su **[Install]** per avviare l'installazione vera e propria.



- 8 Una volta che l'installazione è stata completa clicchiamo su **[Next]** per continuare: ►
- 9 Clicchiamo infine su **[Finish]** per avviare **Android Studio**: ▼

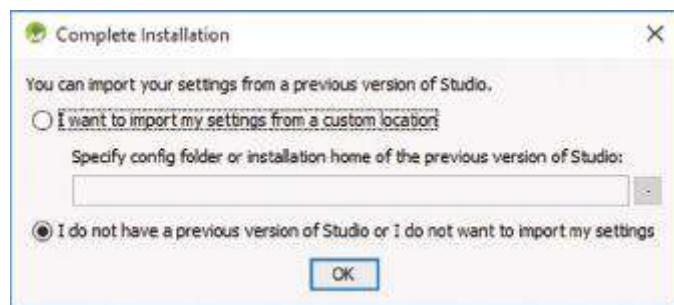


La procedura di installazione non è ancora terminata, mancano ancora alcuni file relativi all'**SDK** che **Android Studio** installerà solo durante la creazione del progetto relativo alla nostra prima applicazione.

## ■ Creare una applicazione

La procedura illustra come creare la nostra prima applicazione per terminare così l'installazione dei componenti aggiuntivi necessari all'uso di **Android Studio**.

- 1 Aprendo **Android Studio** ci viene richiesto di importare le impostazioni da una precedente versione di **Android Studio**: dato che questa è la prima installazione selezioniamo la seconda opzione e clicchiamo su **[OK]**.
- 2 A questo punto viene richiesto se consentire l'accesso a Internet per far comunicare **Android Studio** e la rete attraverso il **Firewall**: è necessario confermare se si vuole utilizzare ◀ **Gradle** ▶, che verifica periodicamente la presenza di aggiornamenti necessari alle librerie di **Android SDK**.



◀ **Gradle** è uno dei più avanzati strumenti di **build automation** del momento che permette di automatizzare in particolare la compilazione, la documentazione e il packaging dei programmi nonché di eseguire dei test automatici per verificare il comportamento dell'applicativo in diverse situazioni. ►

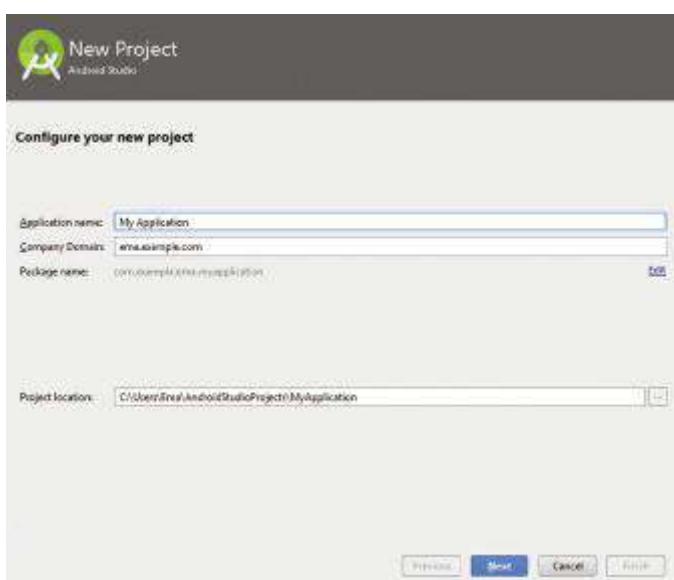


- 3 Appare ora la videata principale di **Android Studio** dalla quale possiamo creare un nuovo progetto, aprire degli esistenti, importare progetti creati in precedenza con altri pacchetti di sviluppo (ad esempio **Eclipse**) e accedere a tutte le impostazioni disponibili. Procediamo con la creazione di un nuovo progetto selezionando la prima voce, **Start a new Android Studio project:** ►



- 4 A questo punto dobbiamo scegliere il **nome** e il **percorso** della nostra applicazione: il nome che sceglieremo sarà lo stesso della **cartella** con la quale il **progetto** sarà identificato e riconosciuto da **Android Studio**.

Facciamo attenzione a scegliere con attenzione il nome da assegnare al progetto: infatti non sarà più possibile modificarlo in seguito, se non con procedure molto elaborate.



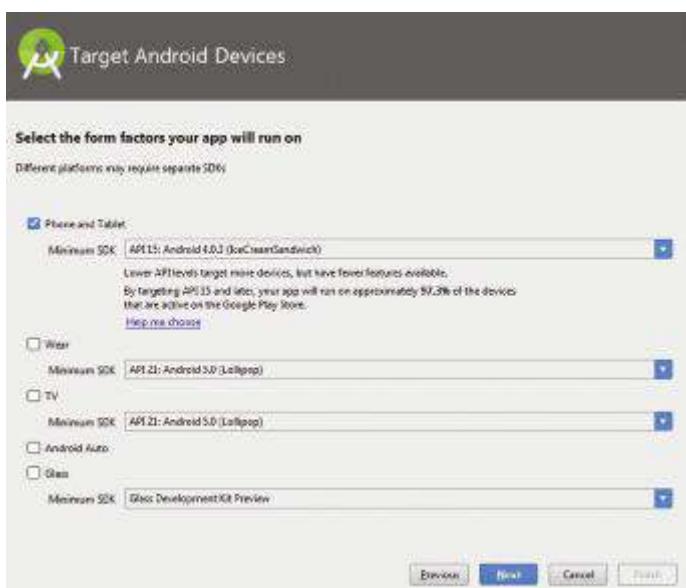
- 5 Una volta definito il nome dell'applicativo e il percorso di salvataggio dobbiamo scegliere il **tipo** di applicazione che andremo a sviluppare. I tipi sono raggruppati in cinque categorie:

- **Phone and Tablet:** è il tipo che ci interessa, riguarda in particolare lo sviluppo di ◀ **applicazioni native** ► per smartphone e tablet dotati di sistema operativo **Android**. Quando si seleziona questa opzione si deve anche indicare la versione minima di **Android** su cui girerà l'applicazione;
- **Wear:** permette di sviluppare applicativi per gli smartwatch dotati di sistema operativo **Android Wear**. Anche in questo caso è necessario indicare una versione minima di compatibilità;
- **TV, Android Auto, Glass:** permettono di sviluppare applicativi per quella ristretta parte di dispositivi che utilizzano le rispettive versioni di **Android**.

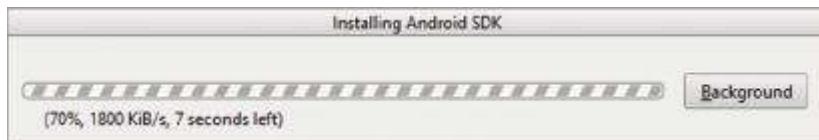


◀ Definiamo **applicazioni native** le applicazioni che sono state sviluppate utilizzando strumenti per il controllo diretto dei dispositivi interessati e non utilizzando software di terze parti che permettono un notevole riciclo del codice a fronte di una scarsa qualità dello stesso. ►

Selezioniamo quindi **Phone and Tablet**, lasciamo la versione suggerita (4.0.3) che copre il 97,3% dei dispositivi in circolazione, quindi clicchiamo su **[Next]**



- 6 Appare la finestra di creazione del primo progetto che installa contestualmente l'**Android SDK**:



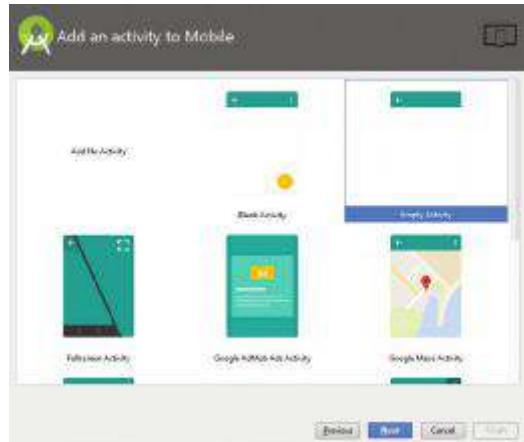
- 7 Terminata l'installazione ci viene mostrata una schermata di conferma di avvenuta installazione, della quale possiamo anche visualizzare i dettagli. Procediamo con la creazione della nostra prima applicazione cliccando su **[Next]**.
- 8 Passiamo adesso a selezionare il tipo di **Activity** che vogliamo inserire: ne esistono di diverse, ciascuna con una precisa funzione.



**Activity** è uno dei principali componenti di ogni applicazione **Android** e costituisce quella parte di applicazione che permette la comunicazione con l'utente. È infatti utilizzata per mostrare informazioni all'utente e comprende parte dei metodi per l'elaborazione delle informazioni stesse. ►

Vediamole in dettaglio:

- **Blank Activity**: quasi vuota, contiene un menu a tendina per le impostazioni e una piccola scorciatoia modificabile a nostro piacimento.
- **Empty Activity**: è la tipologia più semplice, non contiene nulla se non una **TextView** con scritto **HelloWorld!**
- **Fullscreen Activity**: è impostata per un funzionamento a schermo intero, rendendo quindi nascosta la barra delle notifiche.
- **GoogleMaps Activity**: integra un'istanza delle mappe di Google.
- **Settings Activity**: contiene dei componenti grafici molto utili per la regolazione delle impostazioni dell'applicazione.



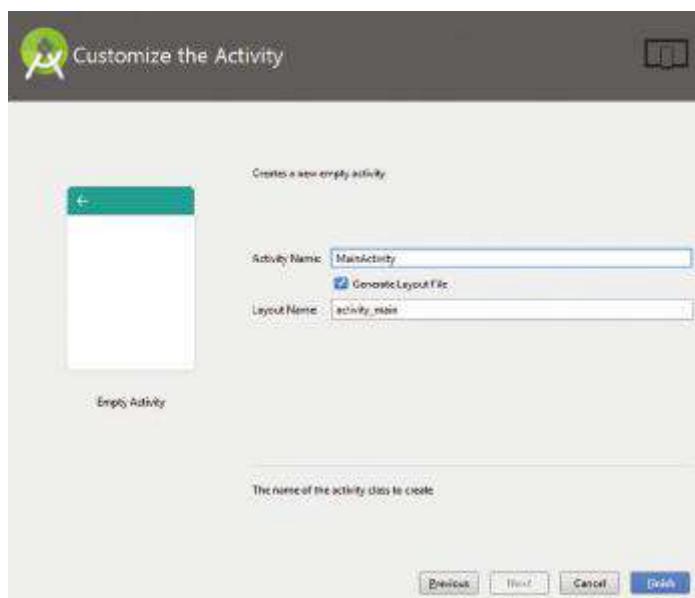
- **NavigationDrawer Activity:** integra un menu laterale molto utile per la navigazione tra varie activity.

Vi sono anche altre tipologie di activity preimpostate, ma esulano da quelli che sono i nostri obiettivi di apprendimento. Selezioniamo **Empty Activity** e clicchiamo su **[Next]** per procedere.

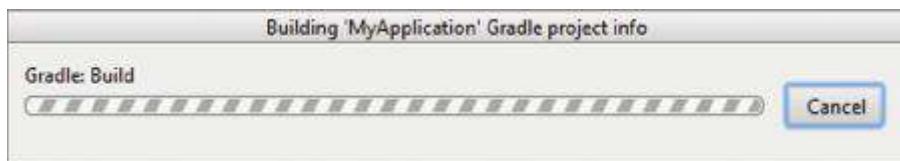
- 9 Dobbiamo adesso scegliere il nome da assegnare alla **activity** e al relativo **layout**. Nel caso non volessimo generare automaticamente il file di **layout** associato all'**activity** possiamo togliere la spunta da “**Generate Layout File**”. In questo caso lasciamo i nomi proposti da **Android Studio** e clicchiamo su **[Finish]** per continuare.



◀ Il **layout** è il componente che definisce l’interfaccia grafica che verrà visualizzata dall’utente ed è associato ad una o più activity. In sostanza il **layout** costituisce la parte visiva mentre l’**activity** rappresenta tutto il codice “che ci sta sotto”. ►



- 10 Appare la schermata che mostra la creazione del progetto con le specifiche definite in precedenza.



In **Android Studio** sono presenti moltissime scorciatoie che ci permettono di velocizzare la fase di programmazione. All’avvio del progetto appare una schermata denominata **Tip of the day**, che suggerisce qualche trucco utile.

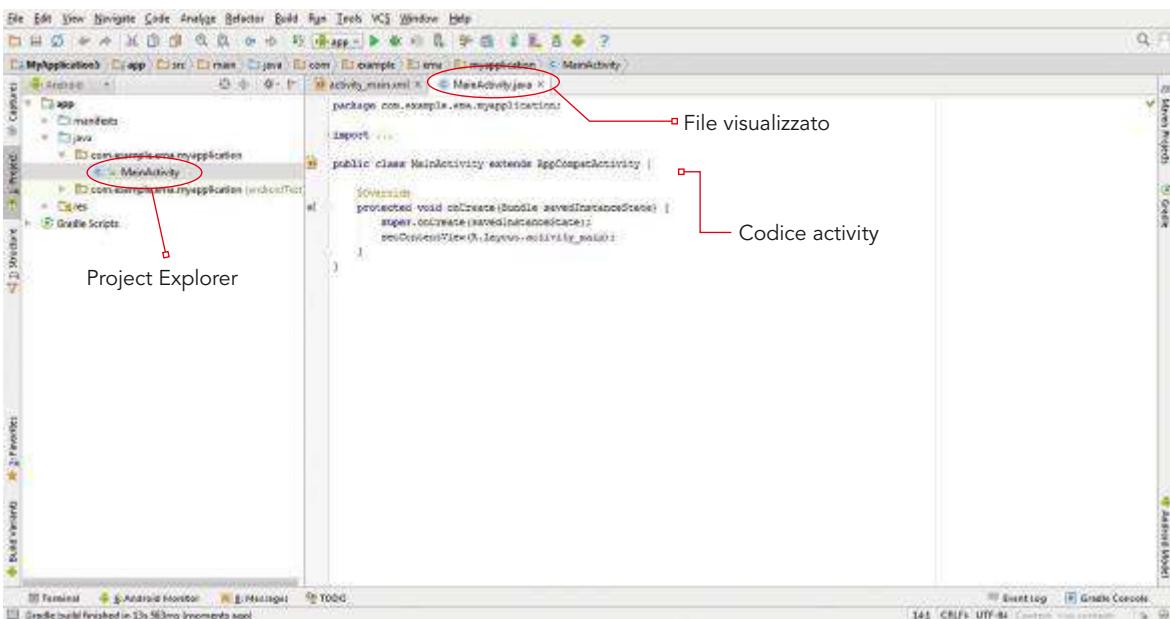
- 11 Dopo aver letto qualche consiglio chiudiamo la schermata cliccando su **[Close]**, finalmente avremo accesso a quella che è la pagina principale dalla quale possiamo gestire ogni aspetto e codice della nostra applicazione.

# ESERCITAZIONI DI LABORATORIO 2

## L'INTERFACCIA GRAFICA DI ANDROID STUDIO

### ■ L'ambiente di lavoro

Vediamo come è strutturata la finestra principale dell'ambiente di lavoro **Android Studio**: sulla colonna di sinistra vi è il **Project Explorer**, grazie al quale è possibile passare agevolmente da un file ad un altro del progetto. Nella finestra centrale vi è l'editor che mostra, in questo caso, il codice dell'**Activity** selezionata (**MainActivity.java**).



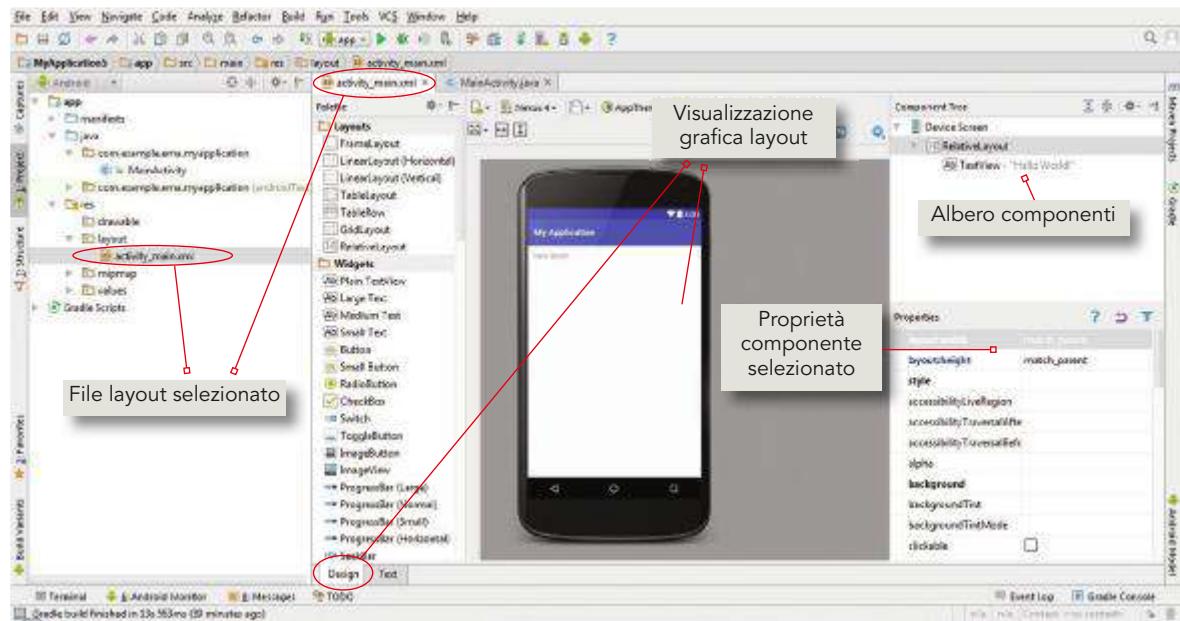
Utilizziamo il **Project Explorer** per visualizzare il file **activity\_main.xml**, che descrive il **layout** del nostro applicativo. **Android Studio** ci propone subito un'interpretazione grafica del layout, molto utile qualora volessimo verificare l'aspetto della nostra applicazione senza aviarla su un dispositivo.



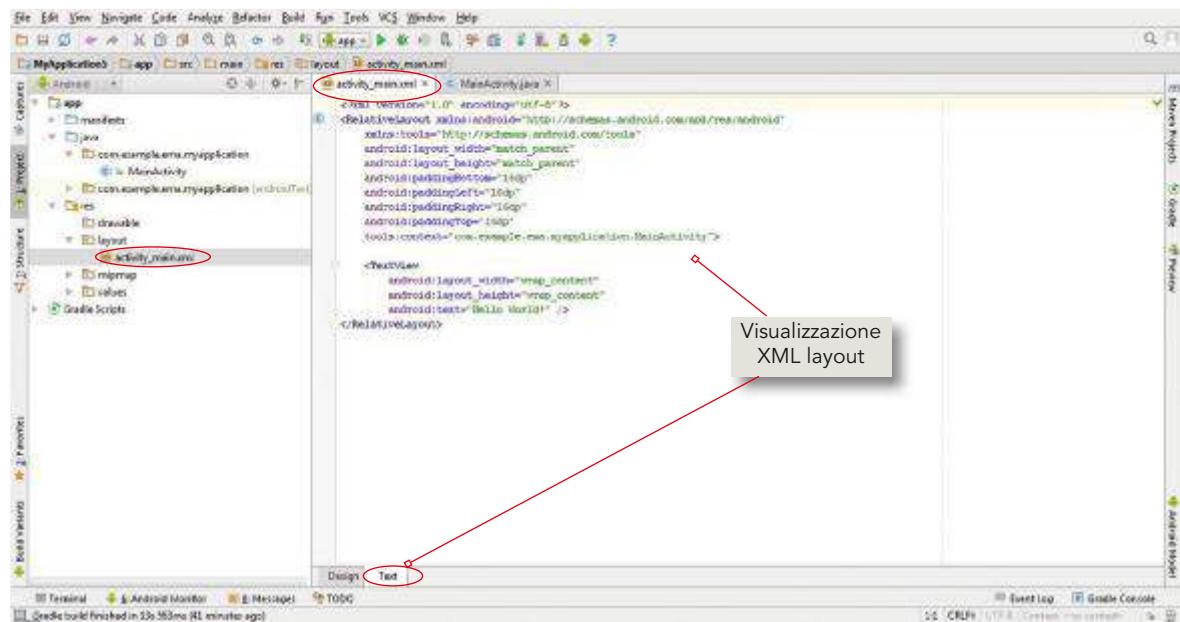
◀ Il **layout** descrive il comportamento grafico dell'**Activity** alla quale è associato, contiene principalmente i **widget** e le loro posizioni all'interno dell'interfaccia. ►

I **widget** sono i **controlli grafici** che consentono l'interazione con l'utente, come ad esempio i pulsanti di azione, le caselle di testo o liste di opzioni. Il termine deriva dalla contrazione dei termini **windows** e **gadget**.

Notiamo sulla destra due riquadri importanti: **Component Tree**, che ci mostra tramite una struttura annidata i **widget** presenti all'interno del nostro layout, e **Properties**, che visualizza la lista delle proprietà relative al controllo selezionato.



Cliccando su **Text** (accanto a **Design**) passiamo alla visualizzazione del layout in modalità testuale. In particolare il layout è definito in linguaggio **XML** e a volte può risultare molto più agevole modificare il layout utilizzando questa modalità.



## ■ Il Project Explorer

Il **Project Explorer** è uno strumento che ci permette di esplorare e navigare tra i file di cui è composto il nostro progetto. In particolare ci permette di accedere a tre file che sono la base del funzionamento della nostra applicazione:

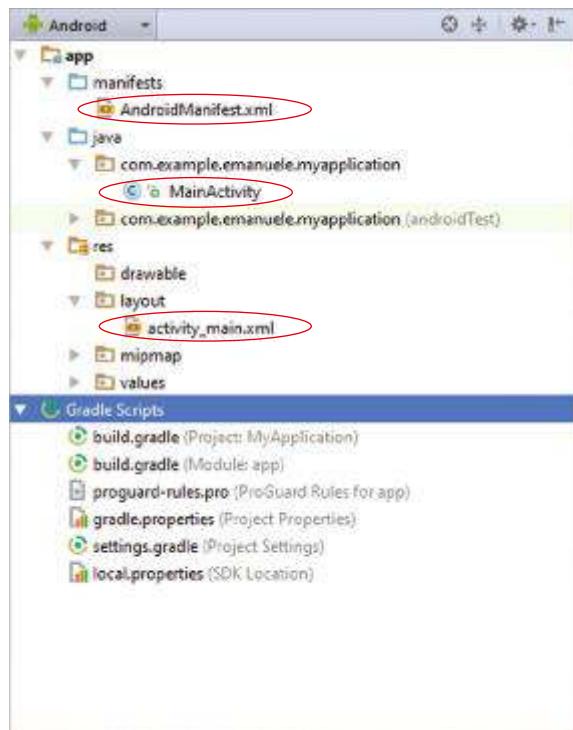
- **AndroidManifest.xml**
- **MainActivity.java**
- **activity\_main.xml**

### AndroidManifest.xml

Quando creiamo un nuovo progetto, viene creato automaticamente il descrittore dell'applicazione, il ►file manifest► chiamato **AndroidManifest.xml**.



► I file manifest consentono di definire la struttura e i metadati XML dell'applicazione. Include un nodo per ogni componente (Attività, Servizi, Content Providers e Broadcast Receiver) e attraverso gli Intent Filter e i Permission determina come ogni componente interagisce con gli altri e le altre applicazioni. ►



Contiene inoltre il nome del package e altre informazioni, come ad esempio:



AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.emanuele.myapplication" >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="My Application"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity android:name=".MainActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Deve essere memorizzato nella cartella principale dell'applicazione e descrive i componenti dell'applicazione, in modo tale che il sistema operativo possa conoscere i componenti e le librerie usate dall'applicazione e necessarie per la sua corretta allocazione in memoria.

## MainActivity.java

Come possiamo osservare, pur creando un'applicazione vuota **Android Studio** inserisce già delle porzioni di codice. In particolare effettua l'**override** del metodo **onCreate()**, richiamando il costruttore della classe madre e settando il layout dell'Activity con il metodo **setContentView()**.



```

package com.example.emanuele.myapplication;

import ...

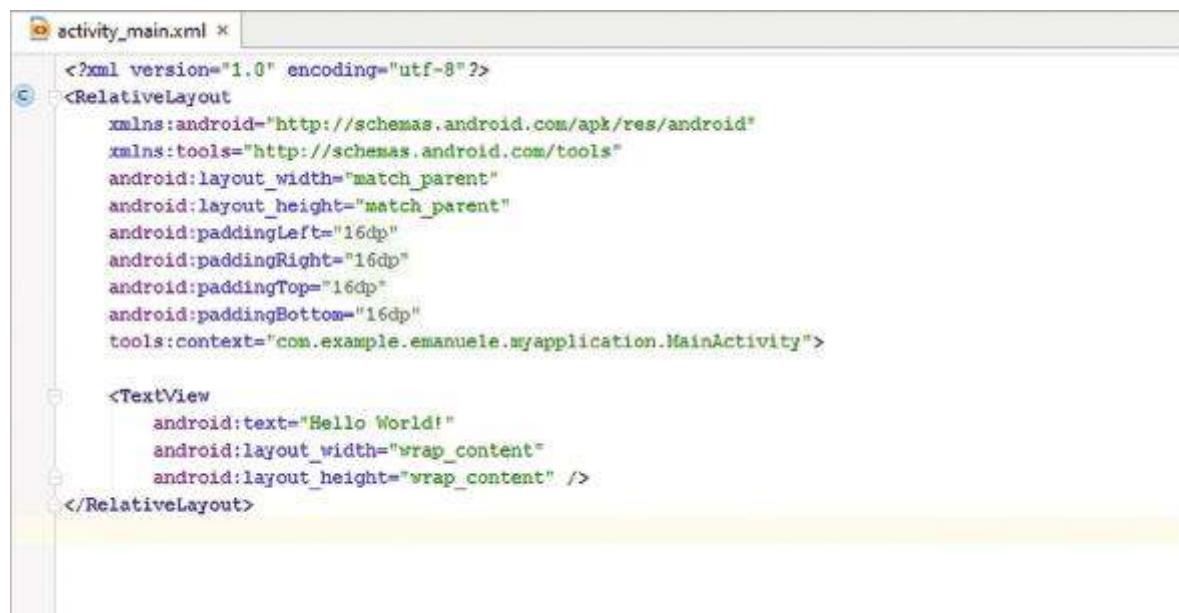
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}

```

## activity\_main.xml

Contiene il codice **XML** che descrive il layout della **activity**. Fornisce informazioni riguardo al contenitore più esterno e ai **widget** in esso contenuti, in questo caso una **TextView** con scritto “Hello World!”.



```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    tools:context="com.example.emanuele.myapplication.MainActivity">

    <TextView
        android:text="Hello World!"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</RelativeLayout>

```

Vogliamo ora provare ad eseguire la nostra prima applicazione. Per poterla provare abbiamo bisogno di un emulatore **Android** funzionante e/o di un dispositivo **Android** collegato in **debug mode**.

## ■ Creare un emulatore

La procedura seguente mostra come configurare l'**emulatore** necessario per effettuare il test e il **debug** delle nostre applicazioni.

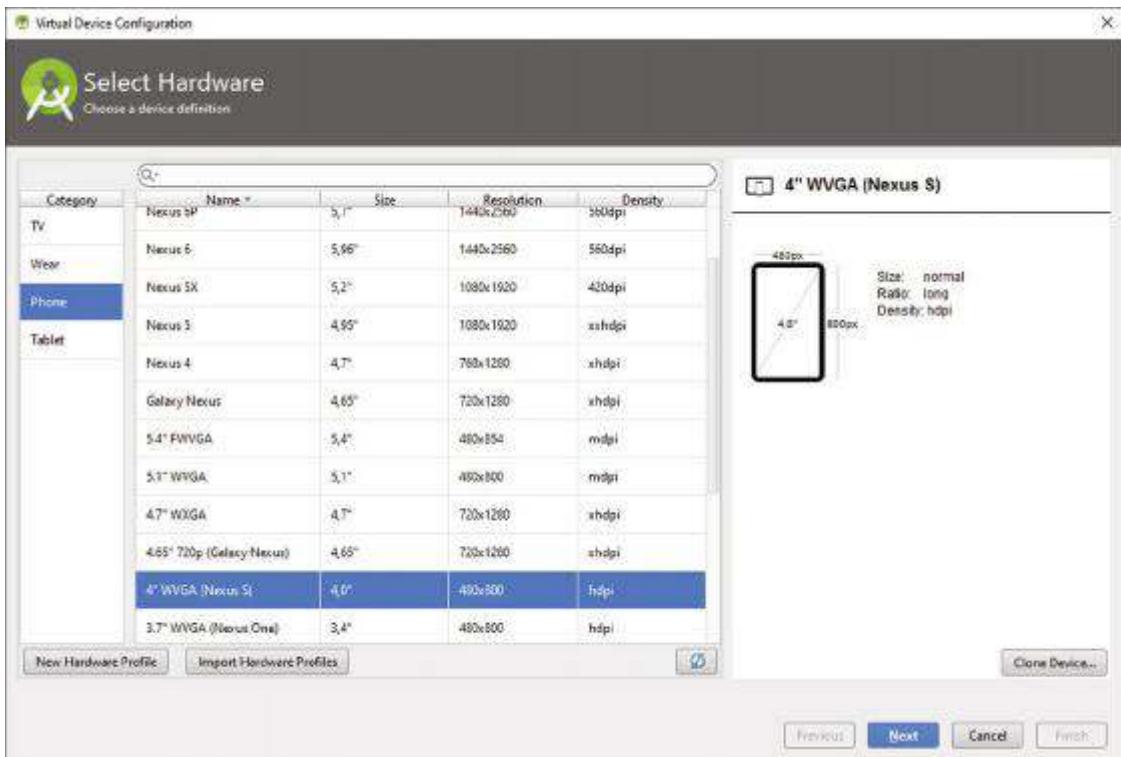
- 1 Dal menu **Tools** di **Android** selezioniamo **AVD Manager**.
- 2 Siccome non è presente alcun emulatore procediamo creandone uno nuovo cliccando su **Create Virtual Device**:



- 3 La seguente finestra consente di selezionare la **risoluzione** in pixel dell'emulatore.

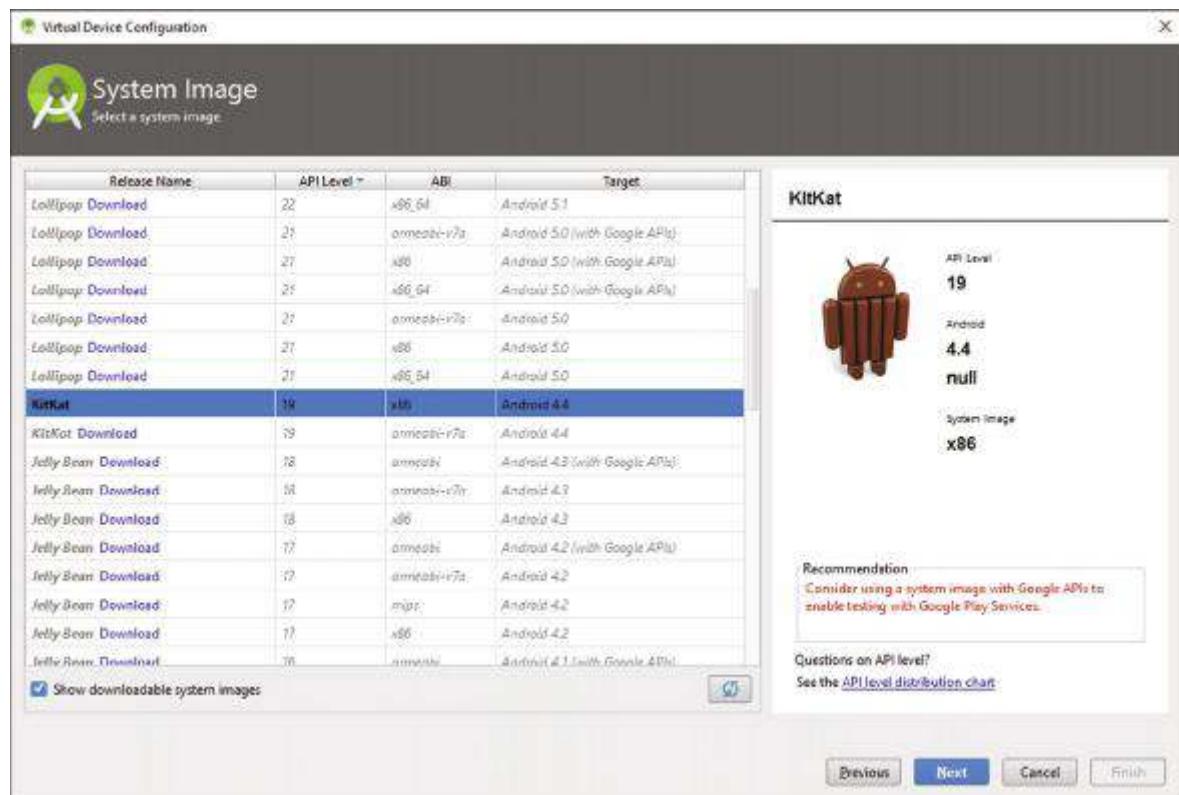
Scegliere una risoluzione troppo alta può portare a dei rallentamenti nell'esecuzione dell'emulatore, nonché a un aumento della memoria **RAM** necessaria al PC per il funzionamento dell'emulatore stesso.

In questo caso sceglieremo un dispositivo mobile **Nexus 4** con risoluzione pari a **480x800**:

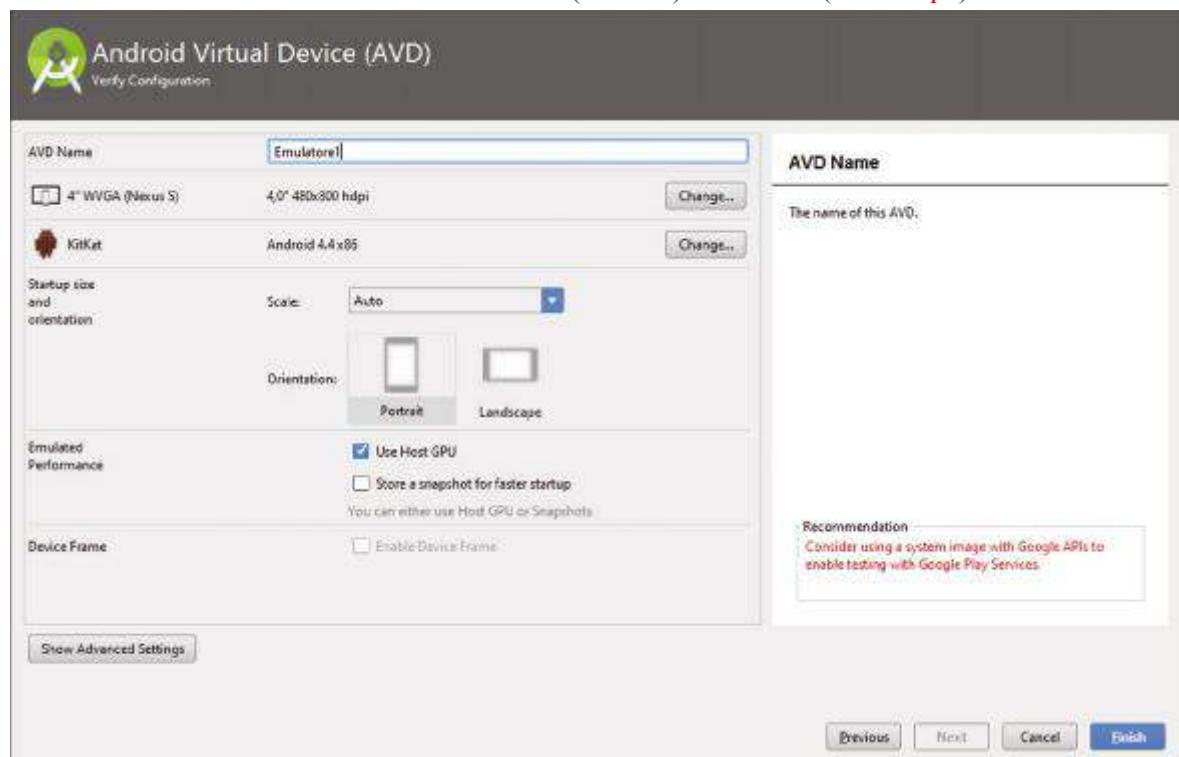


- 4 Selezioniamo ora la versione di sistema operativo **Android** che dovrà essere eseguita dall'emulatore, selezionandola tra quelle disponibili.

La scelta del sistema operativo influisce molto sulle prestazioni dell'emulatore, pertanto consigliamo di utilizzare una versione non troppo recente per non appesantire l'elaborazione del processore (ad esempio la versione **x86**).

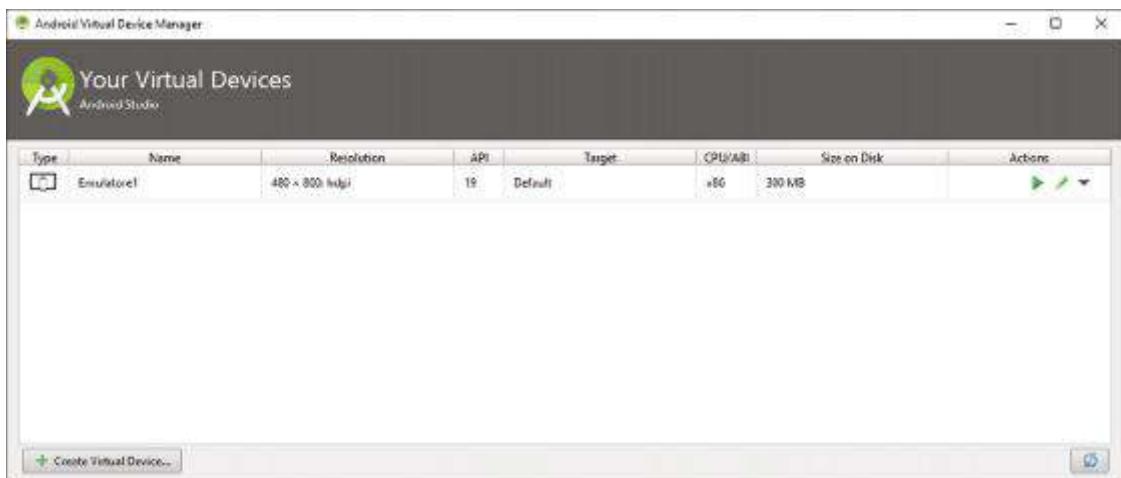


- 5 Selezioniamo infine il **nome** da assegnare all'emulatore (in questo caso **Emulatore1**) e l'**orientamento** dello schermo all'avvio: orizzontale (**Portrait**) o verticale (**Landscape**):



- 6 Viene proposto ora l'elenco degli emulatori creati finora. Selezioniamo l'emulatore che desideriamo mandare in esecuzione e quindi clicchiamo sul pulsante **Play ▶** della colonna **Actions**, per avviarlo:

Creare più emulatori con differenti risoluzioni e versioni di **Android**, può essere utile per verificare come si comporta la nostra applicazione su dispositivi con caratteristiche fisiche e software diverse.



- 7 A questo punto viene mostrato l'emulatore in esecuzione: è pronto per testare le nostre applicazioni. Oltre alla schermata principale, nella quale possiamo utilizzare il mouse per simulare il tocco dello schermo (chiamato **tap**) viene mostrata una colonna, collocata sul lato destro del dispositivo, che emula i pulsanti fisici presenti sul dispositivo. ►

Il prossimo paragrafo spiega come configurare un dispositivo **Android** per utilizzarlo per il debug al posto dell'emulatore.



## ■ Configurazione dispositivo fisico

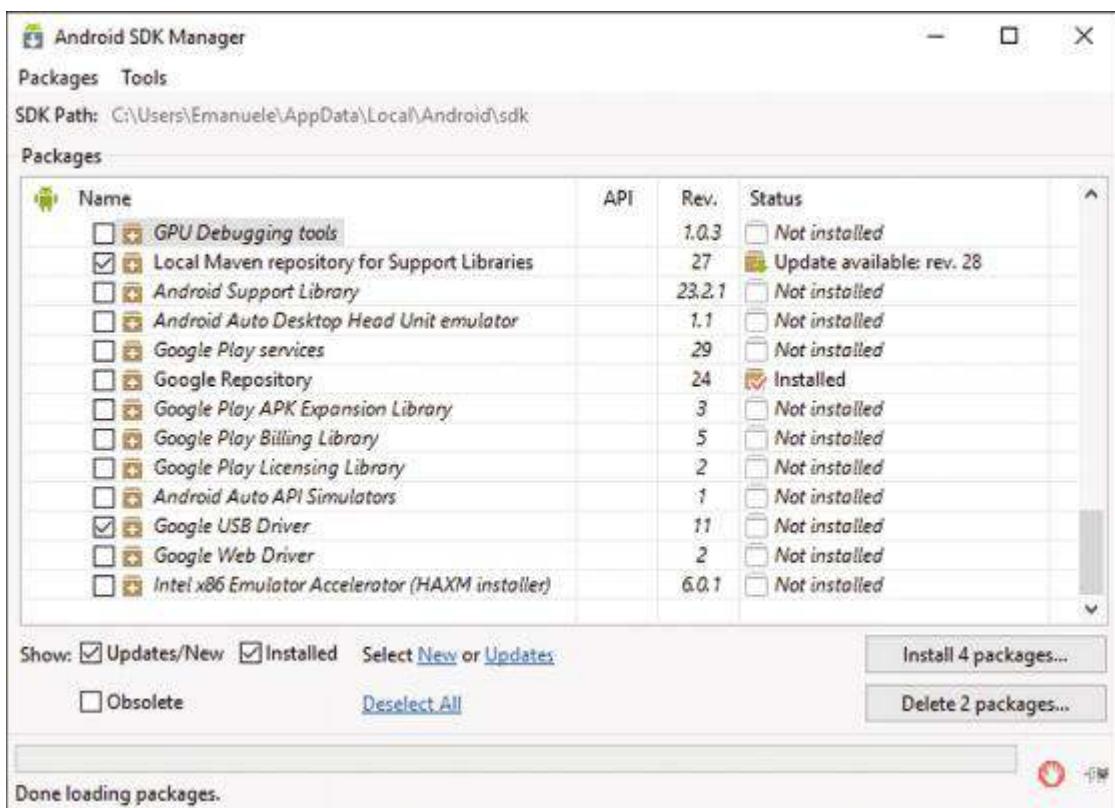
Per effettuare il debug delle applicazioni possiamo utilizzare il dispositivo fisico, cioè lo smartphone o il tablet vero e proprio dotato di sistema operativo **Android**. Per fare questo dobbiamo tuttavia installare i **driver** necessari seguendo la procedura descritta di seguito.

Se volessimo invece utilizzare un dispositivo mobile dotato di sistema operativo **Windows** sarà necessario prima di tutto installare dei particolari driver (**USB drivers**) che lo possano rendere compatibile con **Android Studio**. Al link seguente possiamo trovare alcuni driver disponibili: <https://developer.android.com/studio/run/oem-usb.html>.

- 1 Posizioniamoci nella cartella `C:\Users\NomeUtente\AppData\Local\Android\sdk` e avviamo l'eseguibile **SDK Manager**.

Nome	Ultima modifica	Tipo	Dimensione
add-ons	13/11/2015 00:17	Cartella di file	
build-tools	10/03/2016 14:25	Cartella di file	
docs	13/11/2015 00:18	Cartella di file	
extras	13/11/2015 00:17	Cartella di file	
platforms	10/03/2016 14:27	Cartella di file	
platform-tools	13/11/2015 00:18	Cartella di file	
sources	13/11/2015 00:18	Cartella di file	
system-images	10/03/2016 14:30	Cartella di file	
temp	10/03/2016 14:30	Cartella di file	
tools	13/11/2015 00:18	Cartella di file	
AVD Manager.exe	13/11/2015 00:18	Applicazione	216 KB
SDK Manager.exe	13/11/2015 00:18	Applicazione	216 KB

- 2 Selezioniamo gli aggiornamenti disponibili e **Google USB Driver** e procediamo con l'installazione.



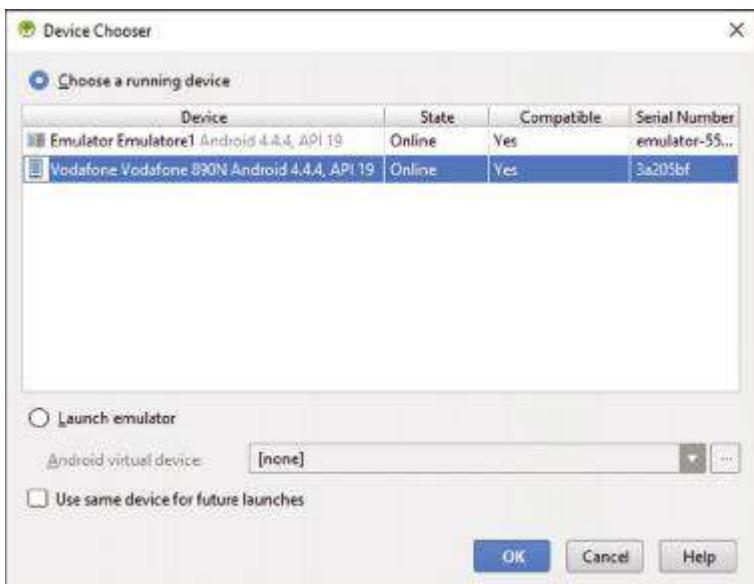
- 3 Ora possiamo collegare il nostro dispositivo **Android** utilizzando l'apposito cavo **USB**. Per poter utilizzare il dispositivo per il debug bisogna abilitare le **Impostazioni sviluppatore** dal menu impostazioni e attivare la funzione **Abilita Debug USB**.



Ogni volta che collegiamo il dispositivo al PC ci verrà chiesta un'ulteriore autorizzazione per il debug.

## ■ Mandare in esecuzione un'app

- Per avviare la nostra applicazione selezioniamo **Run app** dal menu **Run**, otteniamo una schermata simile a questa: ▼



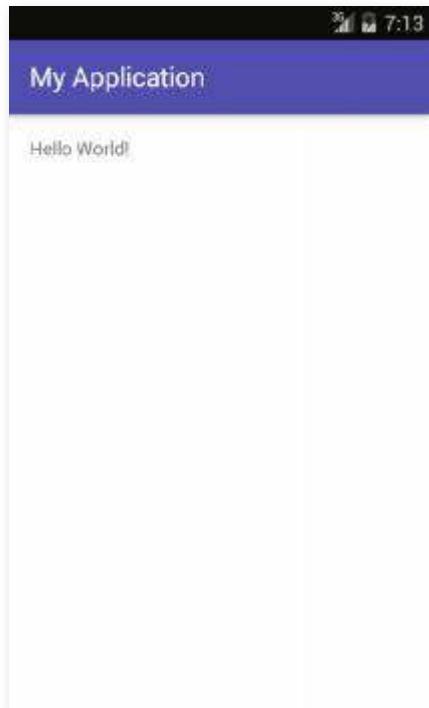
Il numero di voci visualizzate dipende da quanti emulatrici abbiamo avviato e se vi sono o meno dispositivi fisici collegati. In questo caso abbiamo un emulatore avviato e un dispositivo fisico collegato tramite **USB**. Selezioniamo il dispositivo (**Device**) sul quale si desidera avviare l'applicazione e confermiamo con **Ok**.

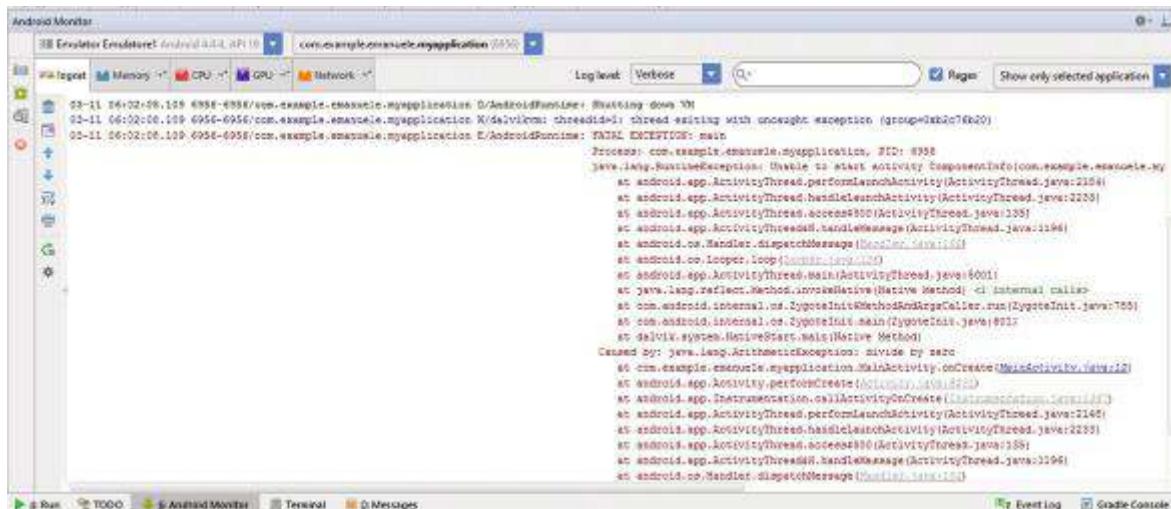
- Sullo schermo del nostro emulatore/dispositivo vediamo adesso l'applicazione in esecuzione, in questo caso si tratta solo di una schermata bianca con la scritta “Hello World!”

Una volta avviata l'applicazione è disponibile un nuovo strumento in **Android Studio**: si tratta di **Android Monitor**, serve per verificare le attività svolte dall'applicazione. Vediamone le cinque principali:

- ▶ **Logcat**: è uno strumento di log, raccoglie tutti i log che l'applicazione invia tramite appositi comandi. Tiene traccia delle funzioni che sono state eseguite e di quelle in esecuzione. Mostra inoltre eventuali errori di **runtime**, come si vede nella finestra a pagina seguente, nella quale possiamo notare un tipico errore causato da una **division by 0**.
- ▶ **Memory**: mostra un grafico che rappresenta l'utilizzo della memoria RAM da parte dell'applicazione nel corso del tempo.
- ▶ **CPU, GPU e Network** mostrano alcuni grafici, analogamente rispetto a **Memory**, legati all'utilizzo rispettivamente del processore, della scheda grafica e della rete Internet.

Nella colonna di sinistra appaiono alcuni pulsanti relativi a funzioni poco utilizzate, in particolare i primi due permettono di acquisire uno **screenshot** dello schermo dell'emulatore o dispositivo collegato e di registrare un **video** di funzionamento dell'applicazione.





Finestra di Logcat che mostra errori di runtime

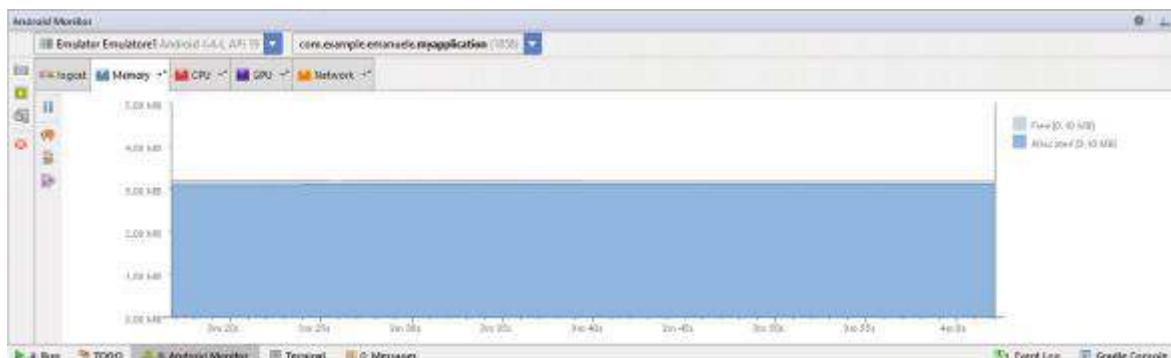


Grafico di Memory che mostra l'utilizzo della RAM

## ■ Effettuare il debug con Android Studio

Il programmatore, durante l'attività di scrittura di un programma deve tener conto dei possibili errori che si producono. Tali errori possono essere raggruppati in tre categorie:

- **errori di compilazione** causati dall'utilizzo di parole che non appartengono al linguaggio oppure dalla costruzione non corretta di frasi istruzioni del codice;
- **errori in fase di esecuzione**, chiamati anche errori di **run time**, segnalati durante l'esecuzione del programma;
- **errori logici** che generano risultati diversi da quelli attesi.

Per evitare di commettere errori, il programma deve essere progettato tenendo conto di tutti i possibili valori che l'utente potrà immettere durante l'esecuzione.



◀ Il termine **debugger** ha un'origine molto lontana nel tempo; venivano infatti chiamati così i lavoratori incaricati di ripulire le valvole dei primi computer dai nidi di alcuni tipi di coccinelle (dall'inglese **bug**, "coccinella", appunto). Col tempo il termine ha preso un'accezione diversa e attualmente indica un software che ripulisce il programma dagli errori. ►

L'ambiente **Android Studio** mette a disposizione uno strumento che consente di individuare i diversi tipi di errore e di apportare al codice le opportune correzioni, chiamato ◀ **debugger** ►.

L'attività di individuazione e correzione degli errori del codice sorgente viene comunemente chiamata fase di **debugging**.



## DEBUGGING

L'attività di **debugging** consiste nel rilevare ed eliminare gli errori di programmazione, in tale fase ci occupiamo innanzitutto degli errori sintattici, che impediscono di mandare in esecuzione un programma.

Un tipico errore è rappresentato dalla mancanza del **punto e virgola** al termine dello **statement**.

Tuttavia non è detto che il compilatore rilevi tutti gli errori, soprattutto se sono di carattere logico. In generale, quando gli errori sono più di uno, conviene procedere cercando di risolvere il primo della lista e proseguire così fino all'ultimo. Inoltre molto spesso alcuni errori si propagano, quindi risolvendo il primo otteniamo una lista assai più ridotta.

Non conviene quindi sempre eseguire alla lettera le indicazioni del compilatore, ma si deve partire da tali indicazioni come spunto per esaminare il codice e capire l'azione che dobbiamo intraprendere.

Per utilizzare il debugger di **Android Studio** dobbiamo applicare tre concetti basilari:

- l'impostazione dei punti di arresto (**breakpoint**);
- l'**ispezione** del contenuto delle variabili;
- l'esecuzione del programma **step by step** (una riga alla volta).

## I punti di arresto (breakpoints)

Per impostare un **breakpoint** dobbiamo fare doppio click accanto alla riga di codice desiderata: apparirà un pallino rosso. Per poter facilmente verificare il funzionamento del debug dobbiamo aprire il file **MainActivity.java** e modificarlo come segue. Notiamo l'aggiunta di un **breakpoint**, necessario per il test della funzionalità.



► Un **breakpoint** è sostanzialmente uno strumento che consente di eseguire un programma con la possibilità di interromperlo quando si verificano determinate condizioni, allo scopo di acquisire informazioni su un programma in esecuzione. Per impostare un **breakpoint** dobbiamo fare doppio click accanto alla riga di codice desiderata: apparirà un pallino rosso. ►

Con un click del pulsante destro sopra il pallino di un **breakpoint** possiamo eventualmente impostare i **breakpoint condizionali**, utili a interrompere l'esecuzione del programma soltanto in caso di condizione verificata:

```

MainActivity.java X
package com.example.emanuele.myapplication;

import ...

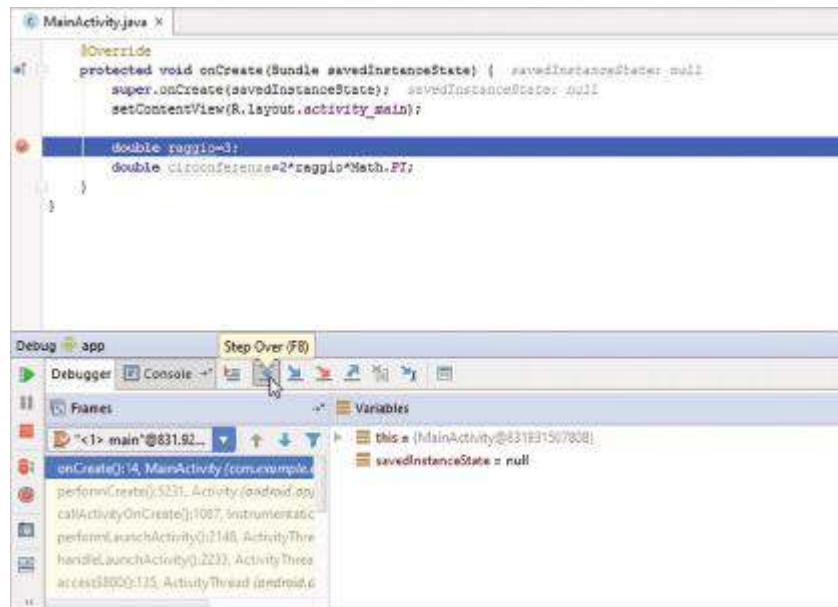
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        double raggio=3;
        double circonferenza=2*raggio*Math.PI;
    }
}

```

Per avviare il debug selezioniamo la voce **Debug 'app'** dal menu **Run**, quindi selezioniamo il dispositivo sul quale effettuare il debug, e attendiamo l'inizializzazione della schermata **Debugger**.

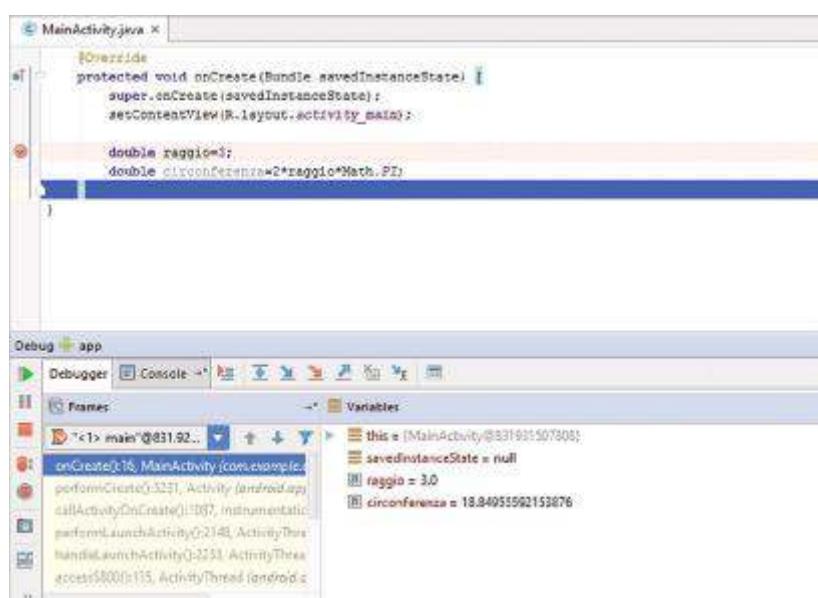


Una volta avviato il **debugger** possiamo notare nella sezione **Variables** i contenuti delle variabili man mano che il programma viene eseguito.

Per riprendere l'esecuzione del programma possiamo utilizzare il rettangolino giallo affiancato dal triangolo verde (**Resume**) che riprende l'esecuzione normale del programma fino al prossimo breakpoint impostato o alla fine del programma.

Il quadratino rosso (**Terminate**) termina l'esecuzione del programma.

Possiamo anche eseguire il programma in modalità passo passo (**step by step**); per fare questo è possibile utilizzare una funzione chiamata **Step Into** che esegue le istruzioni successive, una alla volta, entrando anche nei metodi, se richiamati nel proseguo del programma, oppure è possibile usare la funzione **Step Over**, che esegue le istruzioni successive, una per volta, senza tuttavia eseguirle all'interno dei metodi.



Dopo aver eseguito le due istruzioni successive nel programma visto sopra, possiamo osservare il valore calcolato e memorizzato all'interno della variabile **circonferenza**.

## ■ Toast

Abbiamo visto come effettuare il calcolo e l'assegnazione ad una variabile, tuttavia senza il debug non avremmo mai potuto ottenerne il risultato a video. Esiste un modo assai semplice per comunicare dei dati in output sul display del dispositivo mobile, si tratta della classe **Toast**, che mette a disposizione dei messaggi a scomparsa automatica chiamati appunto **toast**.

Proviamo a modificare il codice dell'esempio precedente come segue:



◀ Il nome **toast** deriva dall'elettrodomestico, proprio come il toast che viene catapultato fuori dal tostapane al termine della cottura, anche il messaggio di testo viene visualizzato per alcuni istanti sullo schermo, per poi scomparire. La forma più usata prevede di impostare i parametri **durata** e **testo** del messaggio, senza indicazioni sulla posizione in cui verrà visualizzato, che in genere è in basso e al centro dello schermo. ►

```
MainActivity.java x
package com.example.emanuele.myapplication;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        double raggio=3;
        double circonferenza=2*raggio*Math.PI;
        Toast toast = Toast.makeText(this,"La circonferenza è " + circonferenza,Toast.LENGTH_LONG);
        toast.show();
    }
}
```

Notiamo che il metodo **makeText** della classe **Toast** vuole in ingresso tre parametri:

- 1 il contesto al quale il **Toast** va associato (in questo caso **this**);
- 2 il testo da visualizzare nel messaggio;
- 3 la durata di tempo del messaggio che può assumere due valori che corrispondono a due costanti presenti nella classe **Toast**: **LENGTH\_SHORT** per una breve durata e **LENGTH\_LONG** per una durata maggiore.

Uno dei tipici errori che viene commesso è quello di non invocare il metodo **show()**, senza di questo metodo nessun messaggio viene visualizzato: sostanzialmente è come se il "toast restasse dentro al tostapane, con il rischio di bruciarsi!".

Testiamo adesso l'applicazione sull'emulatore, notiamo che all'apertura dell'applicazione viene mostrato il messaggio che comunica il valore della circonferenza, sparando dopo pochi secondi. ►





## Prova adesso!

- Testare una applicazione sul dispositivo mobile
- Utilizzare il debugger
- Utilizzare la classe Toast

Progetta e realizza completamente il codice in Java per Android per ciascuno dei problemi proposti.

- 1 Crea una applicazione e testala sul tuo dispositivo mobile (in caso tu non l'avessi a disposizione utilizza l'emulatore di Android Studio e armati di pazienza!), in grado di generare 10 messaggi brevi di tipo **Toast** che visualizzino i primi 10 numeri primi.
- 2 Crea un progetto che mostri una serie di messaggi di tipo **Toast** indicanti i nomi dei sette nani (Eolo, Mammolo, Cucciolo, Brontolo, Pisolo, Dotto, Gongolo).
- 3 Crea un progetto che mostri un messaggio di tipo **Toast** indicante il numero pigreco: verifica quanti decimali al massimo si possono visualizzare.
- 4 Crea un progetto che mostri una serie di messaggi di tipo **Toast** che mostri dieci numeri casuali, ciascuno compreso tra 1 e 100.
- 5 Crea un progetto che mostri una serie di messaggi di tipo **Toast** indicanti i primi 4 numeri perfetti: un numero si dice perfetto quando è pari alla somma dei suoi divisori.



# ESERCITAZIONI DI LABORATORIO 3

## UTILIZZIAMO I WIDGET NELLE APP ANDROID

### ■ La modifica del layout

Per far comunicare l'utente con il dispositivo mobile, **Android** mette a disposizione moltissimi **widget** utilizzabili dal programmatore nelle proprie app. Alcuni controlli grafici, come ad esempio le **TextView**, hanno il solo scopo di **trasmettere informazioni** all'utente, mentre altre, come le **EditView**, vengono utilizzate per poter **ricevere informazioni** che l'utente digita tramite tastiera. La maggior parte dei **widget** in realtà può essere utilizzata in **entrambi i modi**, modificandone le proprietà per comunicare con l'utente e intercettando determinati eventi a scopo decisionale.

Per poter utilizzare questi controlli all'interno della nostra applicazione dobbiamo modificare il file **activity\_main.xml** che, come già accennato, descrive l'aspetto visivo che la nostra applicazione assume quando viene eseguita.

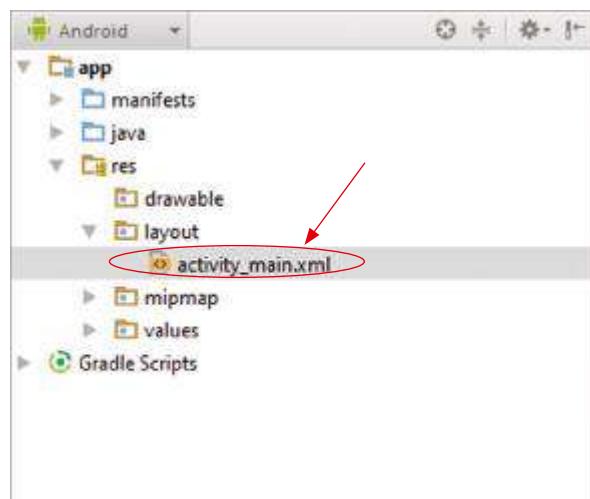
Per poter editare il file e creare componenti e schermate **GUI** possiamo procedere in due modi:

- ▷ trascinando i componenti grafici (**Drag & Drop**);
- ▷ modificando il codice sorgente del file **.xml**.

### Trascinare i componenti grafici

**Android Studio** è dotato di uno strumento grafico che ci permette di aggiungere controlli alla nostra interfaccia semplicemente trascinandoli all'interno dell'anteprima che ci mostra.

Utilizzando il **Project Explorer** ci spostiamo all'interno della cartella **res/layout** e apriamo il file **activity\_main.xml** facendo doppio click su di esso.

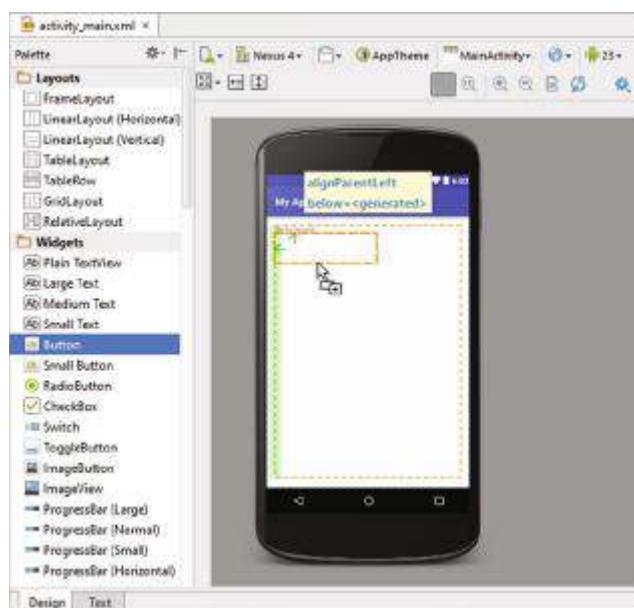


Ci viene quindi mostrata l'anteprima del layout. Selezioniamo un **widget** (in questo caso un semplice **bottone**), lo trasciniamo sull'anteprima dello smartphone quando rilasciamo il pulsante sinistro del mouse lo aggiungiamo al nostro layout nella posizione corrente, come mostrato nella figura a fianco: ►

## Modificare il file xml

Quella che **Android Studio** ci propone come anteprima non è altro che un **rendering** grafico del file **activity\_main.xml** che, come suggerisce l'estensione, contiene una descrizione del layout in formato **xml**: al suo interno possiamo individuare la parte di codice relativa al pulsante che abbiamo appena inserito dal tag:

```
<Button ... />
```



All'interno del tag ci sono alcune **proprietà** che sono state già assegnate da **Android Studio**, come possiamo notare ciascuna **proprietà** viene identificata con la seguente riga di codice:

```
    android:nomeProprietà=valoreProprietà
```

In realtà alcune **proprietà** dei **widget** possono essere osservate anche nell'ambiente grafico del layout.

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    tools:context="com.example.emanuele.myapplication.MainActivity">

    <TextView
        android:text="Hello World!"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/textView" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="New Button"
        android:id="@+id/button"
        android:layout_below="@+id/textView"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

</RelativeLayout>

```

Codice XML relativo al pulsante

Per modificare le proprietà del componente grafico attraverso il layout dobbiamo rientrare in **modalità grafica**: dopo aver selezionato col mouse un controllo, osserviamo la scheda **Properties**, appaiono due colonne, in quella di sinistra appare la **proprietà** e in quella di destra il **valore** ad essa associato:

Tra le principali proprietà comuni a tutti **widget** troviamo:

► **id**: è il nome con il quale il controllo viene univocamente identificato. Utilizzato principalmente per recuperare il riferimento al controllo da codice.

► **Layout:width** indica la larghezza del controllo, oltre a valori in **pixel (px)** e **density pixel (dp)** può assumere i valori **wrap\_content** e **match\_parent**:

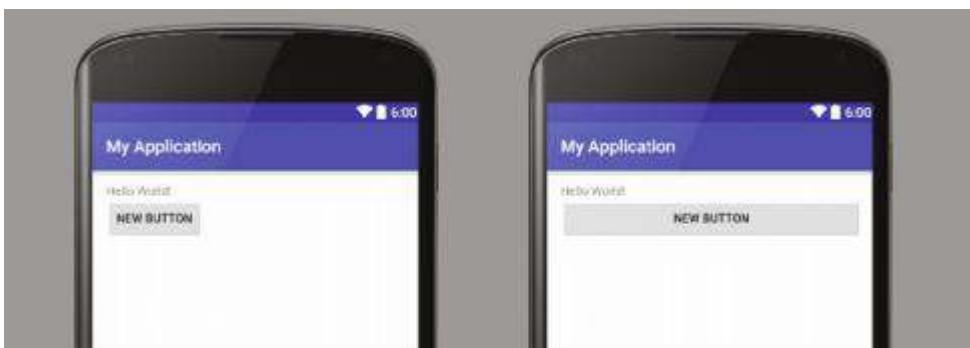
- **wrap\_content**: il controllo si estende in larghezza tanto quanto basta affinché il suo contenuto sia ben visibile (in pratica se il testo all'interno del bottone fosse più lungo il bottone si allungherebbe fino a mostrarlo completamente);
- **match\_parent**: il controllo si estende in larghezza fino al suo contenitore padre.



◀ Acronimo di **Density-Independent Pixel**, **dp** è un'unità di misura astratta basata sulla densità fisica di pixel dello schermo: un **dp** equivale a un pixel su uno schermo con densità pari a **160 dpi**.

È preferibile usare i **dp** ai **pixel** in quando si rende il layout più elastico per meglio adattarsi a schermi con diverse densità di pixel. ►

In questa immagine possiamo osservare la differenza tra i due casi dove nell'immagine di sinistra viene utilizzato il valore **wrap\_content** mentre a destra il valore **wrap\_parent**:



Nel caso di sinistra possiamo apprezzare come il controllo si estenda in larghezza fino alla dimensione del testo contenuto, mentre in quello di destra si espande fino ad occupare lo spazio presente nel contenitore del pulsante stesso, cioè l'intera finestra.

Altre proprietà riguardano il **testo**, il **colore** del testo, i **bordi** del controllo, la **posizione** del controllo rispetto ad altri controlli ecc.

Alcune proprietà dei controlli possono essere modificate durante l'esecuzione richiamando specifici **metodi**.

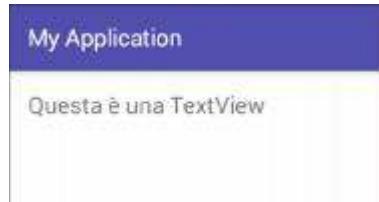
Properties	
layout:width	wrap_content
layout:height	wrap_content
layout:margin	[]
layout:alignEnd	
layout:alignParentEnd	<input type="checkbox"/>
layout:alignParentStart	<input checked="" type="checkbox"/>
layout:alignStart	
layout:toEndOf	
layout:toStartOf	
layout:alignComponent	[top:bottom]
layout:alignParent	[left]

## ■ Widget di base

Vediamo ora alcuni controlli che possiamo definire di base: si tratta dei **widget** largamente utilizzati all'interno delle applicazioni.

### TextView

Gli oggetti **TextView** rappresentano le **etichette di testo**, appartengono alla classe `android.widget.TextView`. Il testo visualizzato può essere definito mediante il metodo `setText()`, che riceve come parametro una stringa.



### EditText

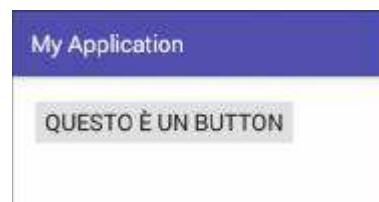
Gli oggetti **EditText** rappresentano le **caselle di testo** e appartengono alla classe `android.widget.EditText`. Estende la classe `TextView` e consente all'utente di immettere del testo. Il testo visualizzato può essere impostato mediante il metodo `setText()`, che riceve come parametro una stringa. Il metodo `getText()` invece, restituisce un oggetto di tipo `android.text.Editable`.



Gli oggetti **Editable** sono simili alle stringhe, e infatti implementano l'interfaccia `java.lang.CharSequence`.

### Button

Rappresentano i **pulsanti** touch presenti sul display, appartengono alla classe `android.widget.Button`. Il controllo espande la classe `TextView`, per questo motivo possiamo impostare il testo mostrato al suo interno con il metodo `setText()` che riceve come parametro una stringa.



### ESEMPIO Convertitore Dollaro/Euro

Vogliamo realizzare un convertitore da Euro a Dollari e viceversa utilizzando i tre widget seguenti:

- un pulsante (**Button**);
- una casella di testo (**EditText**);
- una etichetta di testo (**TextView**).

- 1 Per prima cosa passiamo a creare un nuovo progetto utilizzando una **Empty Activity** come **activity principale**.
- 2 Il layout è attualmente vuoto, per aggiungere i controlli necessari alla creazione della nostra applicazione dobbiamo aprire il file `activity_main.xml` per aggiungere (trascinandoli) tutti i **widget** necessari (sei in tutto) per leggere i dati che l'utente ha digitato e per comunicare il risultato della conversione, come mostrato nell'immagine a fianco: ►

Come possiamo notare osservando il codice `xml` il nostro layout è composto da 2 etichette (`TextView`), 2 caselle di testo (`EditText`) e 2 pulsanti (`Button`). Dal codice `xml` possiamo anche ricavare gli **id** relativi ad ogni controllo, che ci servono per poter leggere o scrivere informazioni relative a un determinato **widget** da codice.





```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    tools:context="com.example.emanuele.myapplication.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="USD"
        android:id="@+id/textView"
        android:textSize="25dp"
        ... />

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/editText"
        android:text="0"
        android:textSize="25dp"
        ... />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="EURO"
        android:id="@+id/textView2"
        android:textSize="25dp"
        ... />

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/editText2"
        android:text="0"
        android:textSize="25dp"
        ... />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="USD -> EURO"
        android:id="@+id/button"
        android:textSize="20dp"
        android:onClick="usdToEuro"
        ... />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="EURO -> USD"
        android:id="@+id/button2"
        android:textSize="20dp"
        android:onClick="euroToUsd"
        ... />

</RelativeLayout>

```

Mandando in esecuzione l'applicazione constatiamo che è possibile scrivere all'interno delle caselle di testo **EditBox**; ovviamente cliccando su uno dei due pulsanti non succede nulla in quanto non sono ancora state definite le azioni che devono essere eseguite quando il relativo pulsante viene premuto.

- 3 Utilizzando il **Project Explorer** apriamo il file **MainActivity.java** e creiamo quindi due nuovi metodi all'interno del file: **usdToEuro()** e **euroToUsd()**.



◀ **View** è la superclasse di tutti i **widget**, la funzione **findViewById()** restituisce un oggetto di tipo **View**, bisogna poi fare un casting al tipo di **widget** che realmente è associato all'id passato come parametro. ►

Analizziamo ora il codice del metodo **usdToEuro()**: viene dapprima richiamata la funzione **findViewById(String)**, la quale restituisce come oggetto la ▲ **View** ▶ avente l'id che è stato passato come parametro.

In questo caso passiamo come parametro **R.id.editText**, poiché è l'id associato alla **EditText** che contiene il valore in dollari inserito dall'utente.

Una volta recuperato il riferimento alle due **EditText** leggiamo il valore in formato **stringa** inserito dall'utente utilizzando il metodo **getText()** e lo convertiamo nel tipo **double** utilizzando il metodo **Double.parseDouble()** della classe **Double**.

Calcoliamo infine il corrispettivo valore in euro e lo inseriamo nella **EditText** associata al valore in euro tramite il metodo **setText()**.

Speculare il codice del metodo **euroToUsd()** che esegue la conversione da euro a dollari.

The screenshot shows the Android Studio interface. On the left is the code editor with **MainActivity.java** open. The code defines a class **MainActivity** that extends **AppCompatActivity**. It contains two methods: **usdToEuro** and **euroToUsd**. Both methods use **findViewById** to get references to two **EditText** views, then use **Double.parseDouble** to convert their text to **double** values, perform the conversion, and set the result back to the **EditText** views. On the right is the **Properties** panel, which is currently focused on the **onClick** property of a button. A dropdown menu shows several options, with **usdToEuro** selected.

```

package com.example.emanuele.myapplication;

import ...

public class MainActivity extends AppCompatActivity {

    final double cambioEuroDollaro=1.1177;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void usdToEuro(View v){
        EditText edit_usd = (EditText)findViewById(R.id.editText);
        EditText edit_euro = (EditText)findViewById(R.id.editText2);
        double usd = Double.parseDouble(edit_usd.getText().toString());
        double euro = usd*cambioEuroDollaro;
        edit_euro.setText(String.valueOf(euro));
    }

    public void euroToUsd(View v){
        EditText edit_usd = (EditText)findViewById(R.id.editText);
        EditText edit_euro = (EditText)findViewById(R.id.editText2);
        double euro = Double.parseDouble(edit_euro.getText().toString());
        double usd = euro*cambioEuroDollaro;
        edit_usd.setText(String.valueOf(usd));
    }
}

```

- 4 Dopo aver creato i due metodi, affinché l'applicazione funzioni, è necessario che questi vengano associati agli eventi **onClick** dei rispettivi pulsanti. Per fare ciò ci spostiamo nella visualizzazione grafica delle proprietà del controllo, selezioniamo la proprietà **onClick** e impostiamo il metodo creato in precedenza.



**Zoom su...**

## UTILIZZO DEI LISTENER

È possibile realizzare lo stesso funzionamento istanziando un **listener** sull'evento **onCLick** nell'**onCreate** dell'activity e definire allo stesso tempo i metodi **onClick** dei relativi pulsanti.

```

MainActivity.java X

package com.example.emanuele.myapplication;

import ...

public class MainActivity extends AppCompatActivity {

    final double cambioEuroDollaro=1.1177;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button button_usdToEuro = (Button) findViewById(R.id.button);
        button_usdToEuro.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                usdToEuro(v);
            }
        });

        Button button_euroToUsd = (Button) findViewById(R.id.button2);
        button_euroToUsd.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                euroToUsd(v);
            }
        });
    }

    public void usdToEuro(View v){
        EditText edit_usd = (EditText) findViewById(R.id.editText);
        EditText edit_euro = (EditText) findViewById(R.id.editText2);
        double usd = Double.parseDouble(edit_usd.getText().toString());
        double euro = usd / cambioEuroDollaro;
        edit_euro.setText(String.valueOf(euro));
    }

    public void euroToUsd(View v){
        EditText edit_usd = (EditText)findViewById(R.id.editText);
        EditText edit_euro = (EditText)findViewById(R.id.editText2);
        double euro = Double.parseDouble(edit_euro.getText().toString());
        double usd = euro*cambioEuroDollaro;
        edit_usd.setText(String.valueOf(usd));
    }
}

```

- 5 Possiamo ora provare la nostra applicazione e verificare che essa funzioni correttamente.

Come possiamo vedere una volta inserito il valore in euro, cliccando sul bottone EURO->USD otteniamo la conversione in dollari visualizzata nella relativa `EditText`.



## Prova adesso!



**APRI L'ESEMPIO** Convertitore

- 1 Modifica il codice dell'esempio per fare in modo che la conversione avvenga automaticamente durante la digitazione del valore.
- 2 Modifica l'esempio inserendo una terza `EditText` associata al valore in sterline.
- 3 Modifica il codice aggiungendo un bottone che azzerà i valori all'interno delle `TextView` ed `Edit Text`.

- Utilizzare il componente `EditText`
- Utilizzare il componente `Button`
- Utilizzare il componente `TextView`

## ■ Altri widget molto utilizzati

### ImageView

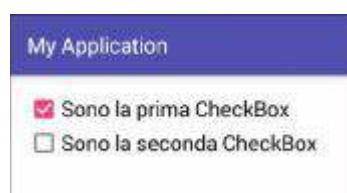
Rappresenta una **immagine** proveniente da un file che deve essere memorizzato in una cartella **drawable**. È presente in `android.widget.ImageView`. Il metodo principale che assegna una immagine al controllo è `setImageResource()`.

Dobbiamo inserire il solo nome del file dell'immagine e non l'estensione. Tuttavia l'estensione può essere solo `.png` oppure `.jpg`.



### CheckBox

Rappresenta un controllo di tipo **casella a scelta multipla** ed è presente nella classe `android.widget.CheckBox`. Estende la classe `Button` e la classe `TextView`, infatti possiamo anche in questo controllo impostare il testo mostrato a fianco delle caselle di spunta, attraverso il metodo `setText()`. Il metodo `isChecked()` restituisce `true` o `false` a seconda che il controllo venga selezionato.



## RadioButton

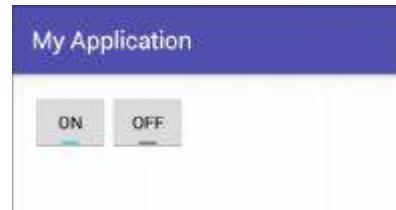
Rappresenta un controllo di tipo **casella a scelta esclusiva** ed è presente nella classe `android.widget.RadioButton`. Estende la classe `Button` e la classe `TextView`, infatti possiamo anche in questo controllo impostare il testo mostrato a fianco delle caselle di spunta, attraverso il metodo `setText()`. Il metodo `isChecked()` restituisce `true` o `false` a seconda che il controllo venga selezionato.

Questi controlli possono essere raggruppati all'interno di un `RadioGroup` in modo che l'utente possa così attivare una sola delle opzioni del gruppo.



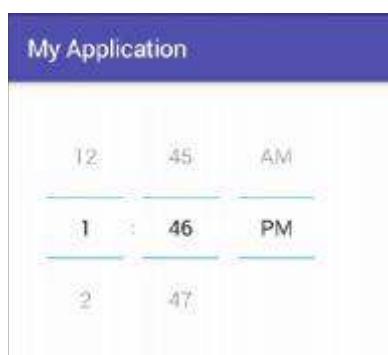
## ToggleButton

Rappresentano degli **interruttori** toccabili sullo schermo e possiedono due stati: attivo (`on`) e non attivo (`off`). Sono presenti nella classe `android.widget.ToggleButton`. Lo stato è indicato dal metodo `isChecked()`, che restituisce `true` o `false` a seconda se il pulsante sia in stato on oppure off. ►



## DatePicker

Rappresentano un controllo per agevolare l'utente nell'immissione di una data nel formato giorno, mese, anno. È un controllo presente nella classe `android.widget.DatePicker`. Possiamo recuperare la data impostata dall'utente grazie ai metodi `getDayOfMonth()`, `getMonth()` e `getYear()`. ►



## TimePicker

Rappresentano un controllo per agevolare l'utente nell'immissione di un orario nel formato ore, minuti. È un controllo presente nella classe `android.widget.TimePicker`. Possiamo recuperare l'orario impostato dall'utente grazie ai metodi `getCurrentHour()` e `getCurrentMinute()`. ►



## ListView

Rappresentano un controllo per agevolare la visualizzazione di numerosi elementi dello stesso tipo. È un controllo presente nella classe `android.widget.ListView`. Per riempirlo bisogna inizializzare un array di stringhe, creare un `arrayadapter`, e chiamare il metodo `setAdapter()` passando come parametro l'`arrayadapter` precedentemente creato. ►

## AREA digitale



Il layout degli elementi grafici

## ESEMPIO *Esempio riepilogativo: riassumi la mia identità*

Vogliamo realizzare un'applicazione che consenta all'utente di inserire dati quali nome, cognome, sesso, data di nascita e eventuali campi d'interesse. L'applicazione deve inoltre poter generare un riassunto in formato testo delle informazioni inserite dall'utente.

Si vuole realizzare un'app con un'interfaccia grafica come la seguente:



Vediamo i **wigdet** da utilizzare:

- **EditText** per consentire all'utente di inserire il proprio nome e cognome;
- **RadioGroup** contenente due **RadioButton**, che permettono all'utente di selezionare il proprio sesso;
- **DatePicker**, grazie al quale l'utente può facilmente selezionare la propria data di nascita;
- **CheckBox**, che rappresentano possibili aree di interesse che l'utente può selezionare.

```
MainActivity.java ×
package com.example.emanuele.myapplication;

import ...

public class MainActivity extends AppCompatActivity {
    String endl = "\r\n";
    EditText nome=null,cognome=null;
    RadioGroup sesso=null;
    DatePicker data=null;
    CheckBox tecnologia=null,arte=null,musica=null,sport=null,auto=null;
    TextView riassunto = null;
```

Modifichiamo il layout per ottenere l'aspetto desiderato:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    nome=(EditText) findViewById(R.id.editTextNome);
    cognome=(EditText) findViewById(R.id.editTextCognome);
    sesso=(RadioGroup)findViewById(R.id.radioGroup);
    data=(DatePicker)findViewById(R.id.datePicker);
```

```

    tecnologia=(CheckBox)findViewById(R.id.checkBoxTecnologia);
    arte=(CheckBox)findViewById(R.id.checkBoxArte);
    musica=(CheckBox)findViewById(R.id.checkBoxMusica);
    sport=(CheckBox)findViewById(R.id.checkBoxSport);
    auto=(CheckBox)findViewById(R.id.checkBoxAuto);

    riassunto=(TextView)findViewById(R.id.textViewRiassunto);
}

```

Quando il bottone **GENERA RIASSUNTO** viene premuto bisogna leggere i valori dei vari widget e utilizzare i valori letti per generare un riepilogo delle informazioni in formato testo. A seconda del tipo di widget la lettura dei dati avviene con metodi diversi.

- ▶ EditText: il metodo **getText()** restituisce il testo all'interno della EditText.
- ▶ CheckBox: il metodo **isChecked()** restituisce true se la casella è spuntata, false altrimenti.
- ▶ RadioGroup: il metodo **getCheckedRadioButtonId()** restituisce l'id del RadioButton selezionato all'interno del gruppo. Grazie all'id possiamo recuperare un riferimento al RadioButton selezionato e ottenere informazioni ad esso associate.
- ▶ DatePicker: il metodo **getYear()** restituisce l'anno selezionato, **getMonth()** restituisce il numero del mese selezionato (partendo da 0) e **getDayOfMonth()** restituisce il numero del giorno selezionato.

Combiniamo queste funzioni per ottenere un resoconto in formato testo delle informazioni inserite dall'utente. Visualizziamo il riassunto all'interno di una TextView.

```

public void generaResoconto(View v) {
    RadioButton selezionato = (RadioButton) findViewById(seesso.getCheckedRadioButtonId());
    String s="Nome: " + nome.getText().toString() + endl
        + "Cognome: " + cognome.getText().toString() + endl
        + "Sesso: " + selezionato.getText().toString() + endl
        + "Data di nascita: " + data.getDayOfMonth()
        + "/" + (data.getMonth()+1)
        + "/" + data.getYear() + endl
        + "Interessi: ";

    if(tecnoLogia.isChecked()) s+= "tecnoLogia ";
    if(arte.isChecked()) s+= "arte ";
    if(musica.isChecked()) s+= "musica ";
    if(sport.isChecked()) s+= "sport ";
    if(auto.isChecked()) s+= "auto ";

    riassunto.setText(s);
}

```

Esempio di funzionamento: ▶





## Prova adesso!



**APRI L'ESEMPIO** Generatore riassunto

- Utilizzare il componente RadioButton
- Utilizzare il componente Button
- Utilizzare il componente RadioGroup

- 1 Modifica il codice dell'esempio aggiungendo controlli per lo stato civile (celibe, sposato, divorziato, vedovo).
- 2 Modifica il codice aggiungendo controlli per le possibili lingue conosciute (italiano, inglese, francese, tedesco, spagnolo).
- 3 Modifica l'esempio affinché il colore del bottone cambi a seconda che sia stato selezionato Uomo o Donna.
- 4 Realizza una interfaccia dove un alunno si iscrive ad un corso di recupero inserendo i propri dati, scegliendo la materia, i giorni della settimana, gli orari e il professore desiderato.
- 5 Progetta una interfaccia grafica per realizzare un semaforo composto da tre cerchi di colore diverso: si dovrà poi poter modificare il raggio di ciascuna di esse mediante uno slider e visualizzarlo e attivare un timer associato ad ogni colore in modo che si accendano in sequenza rispettando quello dei semafori che regolano il traffico automobilistico.
- 6 Progetta una interfaccia grafica per realizzare un programma per la gestione del gioco del tris: il programma dovrà posizionare alternativamente il simbolo X e O a seconda del giocatore selezionato, verificare la presenza di un vincitore e con due pulsanti resettare o uscire dal gioco.
- 7 Progetta una interfaccia grafica per realizzare un "gioco dei puzzle", dove viene scelta un'immagine e vengono definiti il numero di tessere (9 - 16 - 25 - 36), delle quali una rimarrà vuota per poter garantire possibilità di spostamento delle altre; l'applicazione visualizzerà un puzzle in disordine e mediante lo spostamento di un tassello alla volta nel posto libero il giocatore dovrà ricomporre la figura.



# ESERCITAZIONI DI LABORATORIO 4

## UN'APP COMPLETA: LA CALCOLATRICE

### ■ La calcolatrice

Utilizzando le conoscenze acquisite finora possiamo realizzare una semplice calcolatrice, combinando i controlli `EditText` e `Button` per ottenere il layout indicato di seguito:



Per implementare il funzionamento della calcolatrice utilizziamo alcune variabili globali:

- **input**: è un oggetto di tipo `EditText` memorizza il riferimento alla casella di testo all'interno della quale l'utente inserisce i numeri;
- **operator**: di tipo `char` memorizza l'operazione che deve essere eseguita;
- **temp**: di tipo `double` memorizza il risultato parziale delle operazioni effettuate.

```
public class MainActivity extends AppCompatActivity {  
    EditText input=null;  
    char operator ='+';  
    double temp=0;
```

Per dichiarare un oggetto di classe `EditText` utilizziamo la forma:

`classe nome_oggetto=null;`

dove la `classe` è in questo caso `EditText`, il nome dell'oggetto è `input`.

Suddividiamo i pulsanti in tre categorie, associando la stessa funzione all'evento `onClick()` per tutti i pulsanti di quella categoria:

- **numerici**: fanno parte di questa categoria tutti i pulsanti che rappresentano le cifre da 0 a 9 e la virgola;
- **operazionali**: fanno parte di questa categoria i pulsanti che rappresentano le quattro operazioni fondamentali e il pulsante che rappresenta l'uguale;
- **reset**: fa parte di questa categoria solo il pulsante "C" (Clear).



◀ Il `sender` è un parametro passato a una funzione che rappresenta l'oggetto che ha invocato la funzione stessa. Effettuando i dovuti casting è possibile ottenere tutte le informazioni associate al controllo chiamante. ►

Siccome più pulsanti richiamano la stessa funzione, utilizzeremo il parametro ◀ `sender` ▶ per riconoscere quale bottone ha invocato la funzione.

All'`onClick()` dei pulsanti della categoria **Numerici** associamo la funzione `append()`, che concatena la cifra rappresentata dal bottone al valore già presente nella casella di testo chiamata `input`.

```
public void append(View v) {
    Button b = (Button)v;
    if (input.getText().toString().equals("0")){
        input.setText(b.getText());
    }else {
        input.getText().append(b.getText());
    }
}
```

La funzione non gestisce la possibilità di inserire due virgolette di seguito poiché questo compito può essere delegato alla `EditText` andando a modificarne alcune proprietà.



Associamo ai pulsanti della categoria **operazionali**, che rappresentano le operazioni matematiche, la funzione `operation()`, che provvede all'aggiornamento del risultato parziale e la variabile globale `operator` che, come abbiamo detto prima, rappresenta la prossima operazione che deve essere eseguita.

Se viene premuto il tasto "=" che indica la terminazione della operazione viene collocato il risultato della stessa all'interno della casella di testo `input` altrimenti viene azzerata la casella `input` per permettere all'utente di inserire il secondo operando dell'operazione appena selezionata.

```

public void operation(View v){
    char nextOperator = ((Button)v).getText().charAt(0);
    Double numero = Double.parseDouble(input.getText().toString());

    switch (operator){
        case '+':
            temp+=numero;
            break;
        case '-':
            temp-=numero;
            break;
        case '*':
            temp*=numero;
            break;
        case '/':
            temp/=numero;
            break;
    }

    if (nextOperator=='='){
        input.setText(String.valueOf(temp));
    }else {
        input.setText("0");
    }

    operator=nextOperator;
}
}

```

Infine dobbiamo associare il pulsante **C** alla funzione **clear**, che si occupa di azzerare il contenuto della **EditText** e il risultato parziale memorizzato all'interno della variabile **temp**.

```

public void clear(View v){
    input.setText("0");
    operator='+';
    temp=0;
}
}

```

Possiamo ora mandare in funzione l'app e verificarne il funzionamento.



## Prova adesso!



### APRI L'ESEMPIO Calcolatrice

- Utilizzare il componente Button
- Utilizzare il componente EditText

- 1 Modifica il codice dell'esempio per aggiungere la funzione elevamento a potenza.
- 2 Modifica il codice aggiungendo la conversione dei valori da decimale a binario e viceversa.
- 3 Modifica l'esempio affinché ci sia la possibilità di memorizzare un risultato temporaneo per poterlo riutilizzare in futuro.
- 4 Crea una nuova versione della calcolatrice che consenta l'utilizzo delle parentesi.

## AREA digitale



Esercizi per il recupero / Esercizi per l'approfondimento



## AREA digitale



**Lab.5** Utilizziamo i sensori

## AREA digitale



**Lab.6** La connessione a database locali in Android

# 3

# I socket e la comunicazione con i protocolli TCP/UDP

- L1 I socket e i protocolli per la comunicazione di rete
- L2 La connessione tramite socket

## Esercitazioni di laboratorio

- ① Java socket; ② Java socket: realizzazione di un server TCP;
- ③ Realizzazione di un server multiplo in Java; ④ Java socket: un'animazione client-server; ⑤ Il protocollo UDP nel linguaggio Java; ⑥ Applicazioni multicast in Java; ⑦ Un esempio completo con le Java socket: "la chat"; ⑧ I socket nel linguaggio C; ⑨ Server TCP in C; ⑩ Client TCP in C; ⑪ Il protocollo UDP nel linguaggio C; ⑫ Un esempio completo con i socket: asta online

### Conoscenze

- Conoscere i protocolli di rete
- Acquisire il modello di comunicazione in una network
- Avere il concetto di socket e conoscere le tipologie di socket
- Conoscere la comunicazione multicast
- Conoscere le caratteristiche della comunicazione con i socket Java
- Conoscere le caratteristiche della comunicazione con i socket C

### Competenze

- Effettuare la connessione col protocollo TCP e UDP
- Acquisire il protocollo UDP nel linguaggio Java
- Utilizzare le classi Classe Socket e ServerSocket
- Progettare applicazioni client-server in Java
- Progettare applicazioni client-server in C

### Abilità

- Realizzare un server e client TCP in Java
- Realizzare un server multiplo in Java
- Realizzare un server UDP in Java
- Realizzare un server e un client TCP in C
- Realizzare un server UDP in C

## AREA *digitale*

-  Esercizi
-  Confronto ISO/OSI e Internet
- Connectionless e connection-oriented
- API – Application Programming Interface
- Un esempio di verifica
- Funzioni di conversione Big-Endian/ Little-Endian
- Abilitare telnet in Windows 8 e Windows 8.1
- Funzioni GETSOCKOPT() e SETSOCKOPT()



### Esempi proposti

Consulta il DVD in allegato al volume



### Soluzioni

Puoi scaricare il file anche da  [hoepliscuola.it](http://hoepliscuola.it)

# I socket e i protocolli per la comunicazione di rete

In questa lezione impareremo...

- ▶ i protocolli di rete
- ▶ il modello di comunicazione in una network
- ▶ il concetto di socket

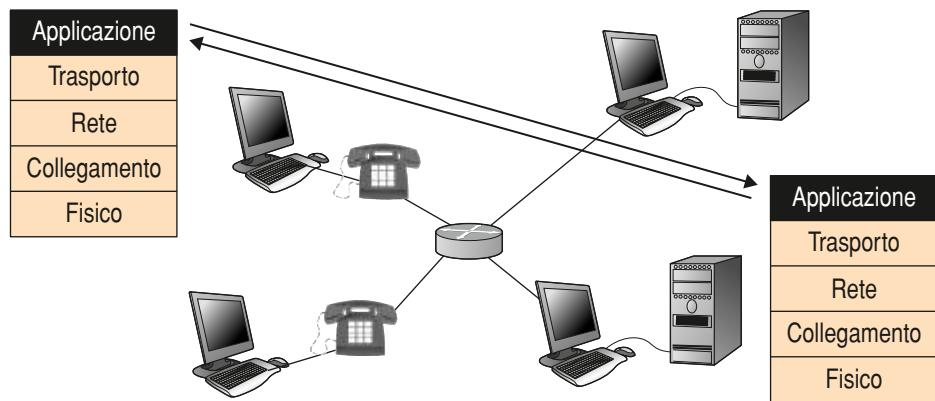
## ■ Generalità

Riprendiamo alcune definizioni già note, come il concetto di applicazione di rete e di protocollo.

### Applicazione di rete

In generale un'applicazione di rete è costituita da un insieme di programmi che vengono eseguiti su due o più computer **contemporaneamente**: questi operano interagendo tra loro utilizzando delle risorse comuni, accedendo cioè **concorrentemente** agli archivi (database), mediante la rete di comunicazione che li connette.

L'applicazione di rete prende anche il nome di **applicazione distribuita** dato che non viene eseguita su di un solo elaboratore (concentrata).





## APPLICAZIONE DISTRIBUITA

Un'applicazione distribuita è un'applicazione composta da più elementi cooperanti posti in esecuzione su macchine diverse all'interno di una rete di calcolatori.

I **processi** hanno la necessità di **scambiare messaggi** con gli altri **processi** della medesima applicazione, sia che essi appartengano alla stessa rete locale oppure che siano remoti e quindi dislocati dall'altra parte del globo: per comunicare tra loro questi processi devono mettersi in "contatto" tramite i loro indirizzi e utilizzare i servizi offerti dal **livello di applicazione**.

## Protocollo di comunicazione

Due o più computer possono cooperare se sono collegati attraverso un sistema di comunicazione e utilizzano un *meccanismo comune di dialogo*: il **protocollo di comunicazione**.



## PROTOCOLLO DI COMUNICAZIONE

Con **protocollo di comunicazione** si indica l'insieme di regole di comunicazione che devono essere seguite da due interlocutori affinché possano comprendersi.

In una rete di calcolatori il **protocollo di comunicazione** stabilisce tutti gli aspetti della comunicazione, da quelli fisici (per esempio supporto fisico, meccanismo di segnalazione ecc.) a quelli più specificamente logici (per esempio meccanismo di commutazione, regole di codifica dell'informazione ecc.).

I **protocolli** utilizzati dai calcolatori sono organizzati secondo una gerarchia che prevede che ogni protocollo si appoggi ai protocolli di livello più basso per fornire un servizio di qualità superiore

## ESEMPIO

Il protocollo che stabilisce le *regole di codifica dell'informazione* si appoggia a un protocollo di trasporto che stabilisce *come debbano essere trasportati i dati*.

Ricordiamo che la **pila protocollare di Internet** (o pila **TCP/IP**) è costituita da cinque livelli, a differenza della pila **ISO/OSI** che ha sette livelli, e l'unità di informazione che viene gestita a ogni livello assume *forma e nome diverso*, dato che a ogni passaggio vengono aggiunte le informazioni necessarie a ciascun protocollo per poter portare termine i propri compiti:

- al livello **applicazione** viene elaborato il tipo di dato **messaggio**;
- al livello **trasporto** viene elaborato il tipo di dato **segmento**;
- al livello **rete** viene elaborato il tipo di dato **datagram**;
- al livello **collegamento** (link) viene elaborato il tipo di dato **frame**;
- al livello **fisico** non corrisponde alcun tipo di dato (ma solo un segnale fisico).

5 Applicazione
4 Trasporto
3 Rete
2 Collegamento
1 Fisico

Messaggio

Segmento

Datagram

Frame

**AREA digitale**



Confronto ISO/OSI e Internet

Lo **strato di trasporto** della rete **Internet** mette a disposizione delle applicazioni attive in ciascun host due distinti protocolli di trasporto:

1. **TCP (Transmission Control Protocol)**;
2. **UDP (User Datagram Protocol)**.

**TCP** e **UDP** svolgono funzioni diverse, cioè offrono servizi diversi allo **strato applicativo**: ogni applicazione (processo) in esecuzione, prima di poter trasmettere messaggi sulla rete, sceglie il protocollo da usare per poter realizzare la comunicazione. La differenza fondamentale tra i due protocolli, che saranno descritti dettagliatamente nelle prossime due lezioni, è che:

- il protocollo **TCP** è orientato alla connessione (**connection-oriented**), ed è affidabile dato che consente il controllo dell'integrità dell'informazione contenuta nei pacchetti e dispone di un sistema per segnalare l'errore al mittente (scambio di messaggi di acknowledge);
- il protocollo **UDP** è senza connessione (**connectionless**) e quindi non affidabile.

Sia il **TCP** che l'**UDP** usano **IP** a livello di network. Il nome del protocollo completo di comunicazione viene ottenuto dalla combinazione delle sigle di tali protocolli: per esempio, con **TCP/IP** si individua l'insieme di questi standard.



## ■ Le porte di comunicazione e i socket

Prima di entrare nelle specifiche e di analizzare le primitive che permettono la realizzazione della comunicazione in rete è necessario riprendere alcuni concetti fondamentali necessari per realizzare la comunicazione tramite network.

Affinché un processo, presente su un determinato **host**, invii un messaggio a un qualsiasi altro **host**, il processo **mittente** deve identificare il processo **destinatario** in modo univoco.

Generalmente ogni PC ha **una sola porta fisica** di connessione al network: se più applicazioni necessitano di utilizzare la rete, devono essere riconosciute in qualche modo, permettendo così di ricevere e spedire dati in modo corretto. Questo metodo di riconoscimento viene effettuato tramite le cosiddette **porte logiche** identificate da un numero detto **port** o **port address**: tale numero non è però sufficiente a descrivere in maniera univoca una connessione, infatti più processi su **host** differenti potrebbero avere il medesimo **port** ed essere quindi riconosciuti come un processo unico.

Una **porta logica** ha valore numerico specificato su 2 byte (da 0 a 65535) che identifica un particolare canale utilizzabile per la comunicazione. In questo modo è possibile instaurare simultaneamente più comunicazioni cosicché due applicazioni possono comunicare l'una con l'altra indipendentemente dal fatto che sulla rete stiano avvenendo altre comunicazioni: è sufficiente utilizzare **porte logiche** diverse.

I numeri di porta da **0** a **1023** sono riservati ad applicazioni particolari (quali **HTTP**, **FTP**, **DNS**, **TelNet** ecc.), costituiscono i cosiddetti “**well-known port numbers**” e possono essere usati solo dai server in apertura passiva.

I numeri di porta da **1024** a **49151** sono riservati a **porte registrate** e sono usati da alcuni servizi ma possono anche essere utilizzati dai client (tutti i client usano normalmente le porte a partire dalla numero 1024 per collegarsi a un sistema remoto).

I numeri di porta da **49152** a **65535** sono liberi per essere assegnati dinamicamente dai processi applicativi.

Well-Known Ports	Registered Ports	Dynamic and/or Private Ports
0      1023      1024	49151      49152	65535

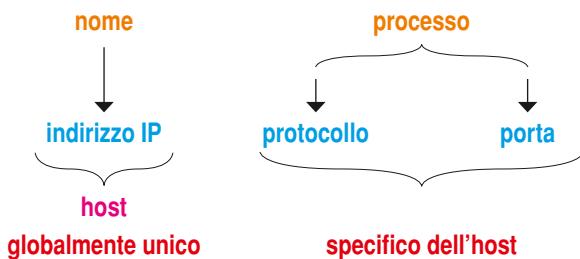
La lista completa dei *well-known port numbers* è reperibile al sito <http://www.iana.org/assignments/port-numbers>. Di seguito sono riportati alcuni esempi.

Porta logica	Protocollo di rete
7	ECHO
21/tcp	FTP (file transfer protocol)
22/tcp	SSH (Secure SHell)
23/tcp	TELNET
25/tcp	SMTP (Simple Mail Transfer Protocol)
42	WINS (Windows Internet Naming Service)
53	DNS (Domain Name Service)
80/tcp	HTTP (Hyper Textual Transfer Protocol)
110/tcp	POP3 (Post Office Protocol, v3)
143/tcp	IMAP (Internet Message Access Protocol)
161	SNMP (Simple Network Management Protocol)
389/tcp	LDAP (Lightweight Directory Access Protocol)
443/tcp	HTTPS (Secure HTTP)

È bene sottolineare il fatto che i numeri di **porta logica** sono relativi soltanto al protocollo considerato: una determinata porta per il protocollo **TCP** è diversa dallo stesso numero di porta per il protocollo **UDP** (si tratta effettivamente di porte diverse) anche se in genere viene utilizzato lo stesso numero di porta per un servizio che gestisce entrambi i protocolli.

L'identificazione di un servizio, quindi, avviene combinando l'**indirizzo IP** dell'**host** e della **porta** che viene utilizzata per rendere disponibile il servizio:

- un'**identificazione del nodo** su cui opera il **processo** con cui si desidera comunicare;
- un'**identificazione della porta** utilizzata dal particolare **processo** all'interno di quel nodo.



### ESEMPIO

Un caso tipico è quello rappresentato da un server sul quale sono disponibili più servizi, tra i quali:

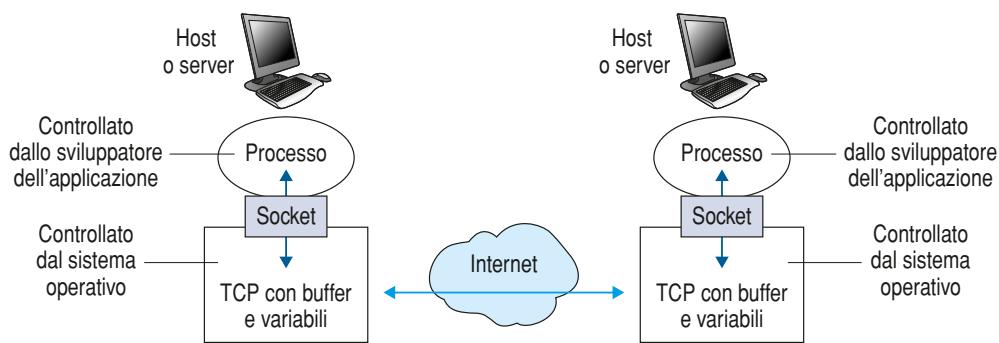
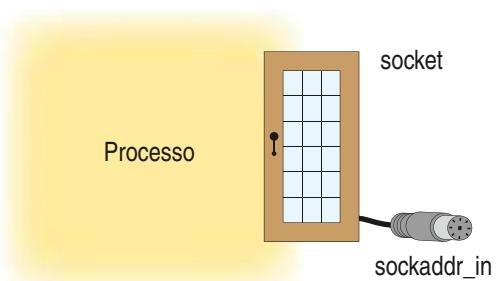
- **e-mail**: viene inviata usando il protocollo applicativo **SMTP**, quindi occorre inviare un messaggio opportunamente codificato alla porta **TCP 25** del **server**;
- **sito Web**: per richiedere una pagina Web si usa il protocollo applicativo **HTTP**, che invia un opportuno messaggio di richiesta alla porta **TCP 80** del **server**;
- per **trasformare un nome** di un calcolatore in **un indirizzo IP** si invia una opportuna richiesta alla porta **UTP 53** del server che offre il servizio **DNS**.

Se un client desidera richiedere uno di questi servizi offerti da questo **server** è necessario che specifichi non solo l'**indirizzo dell'host** ma anche il **tipo di servizio desiderato**.

L'identificazione univoca avviene conoscendo sia l'**indirizzo IP** che il **numero di porta** associato al processo in esecuzione su un **host**: questo meccanismo è già introdotto nelle unità precedenti e prende il nome di meccanismo dei **socket**.



Un **socket** è formato dalla coppia **<indirizzo IP: numero della porta>**; si tratta di un identificatore analogo a una porta, cioè a un punto di accesso/uscita: un processo che vuole inviare un messaggio lo "fa uscire" dalla propria "interfaccia" (**socket** del mittente) sapendo che un'infrastruttura esterna lo trasporterà attraverso la rete fino alla "interfaccia" del processo di destinazione (**socket** del destinatario).



Un **socket** consente quindi di comunicare attraverso la rete utilizzando la pila **TCP/IP** ed è quindi parte integrante del protocollo: le **API** mettono a disposizione del programmatore gli strumenti necessari a codificare la connessione e l'utilizzo del protocollo di comunicazione.

### AREA digitale

API – Application Programming Interface

#### ESEMPIO

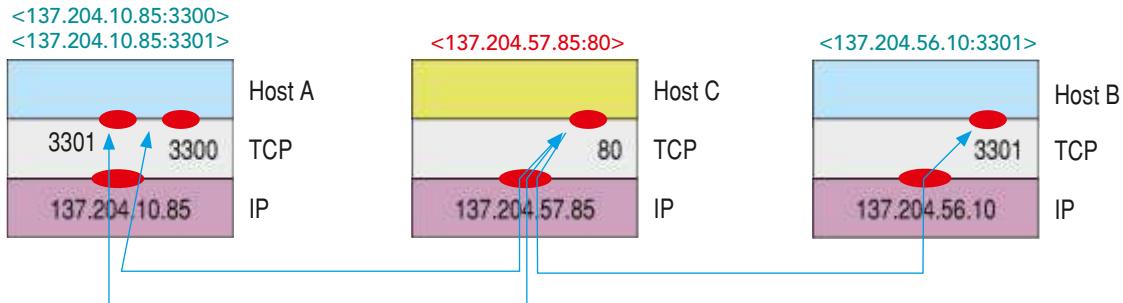
Nell'esempio schematizzato nella figura seguente, due applicazioni dell'host A e una dell'host B con i seguenti **socket**:

host A: <137.204.10.85:3300>  
host A: <137.204.10.85:3301>  
host B: <137.204.56.10:3301>

si connettono alla stessa porta 80 dell'host C richiedendo un servizio http al **socket**:

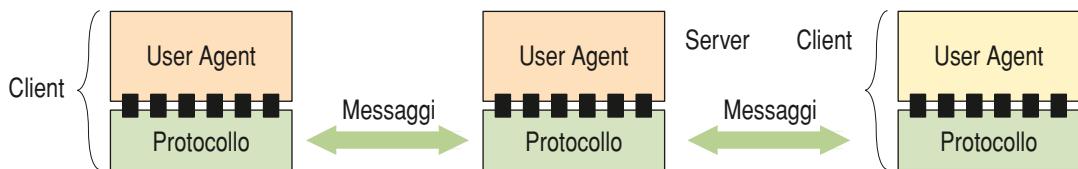
host C: <137.204.57.85:80>

avendo due il medesimo **indirizzo IP** sono univocamente individuati mediante il valore del **socket**.



L'applicazione di rete può essere vista come composta da due parti:

- ▷ una **user agent**, che funge da interfaccia tra l'utilizzatore dell'applicazione e gli aspetti comunicativi;
- ▷ l'implementazione dei **protocolli**, che permettono all'applicazione di integrarsi con la rete.



### ESEMPIO

Possiamo individuare queste due componenti per esempio in un **browser Web**:

- ▷ l'**interfaccia utente** che serve a visualizzare i documenti ricevuti, a permettere la loro navigazione e a richiedere nuovi documenti specificando la loro **URL**;
- ▷ il **motore del browser** che è la parte che si preoccupa di inviare le richieste ai vari server e di ricevere le risposte.

Esiste una struttura chiamata **association** che consente d'identificare **ogni singola connessione in modo univoco** e che contiene le seguenti informazioni:

Protocollo (TCP, UDP...)	Indirizzo IP (locale)	Indirizzo IP (remoto)
	Porta (locale)	Porta (remota)

### ESEMPIO

Un esempio di **association** è: **TCP, 192.168.1.2, 1500, 192.168.1.14, 21**.

Nel dettaglio:

- ▷ **TCP** è il protocollo da utilizzare;
- ▷ **192.168.1.2** è l'indirizzo IP del **PC mittente**;
- ▷ **1500** è il port address del processo locale;
- ▷ **192.168.1.14** è l'indirizzo IP del **PC destinatario**;
- ▷ **21** è il port address del processo remoto.



**Zoom su...**

### SOCKET: MASCHILE O FEMMINILE?

Le regole per la definizione del genere maschile o femminile per un termine tecnico proveniente dalla lingua inglese, che viene usato così com'è in italiano, sono molto vaghe: il neutro inglese "dovrebbe" utilizzare l'articolo del corrispondente termine tradotto. Inoltre i termini inglesi che vengono incorporati nell'italiano vanno usati generalmente al singolare, anche quando esprimono quantità multiple. La traduzione dall'inglese di **socket** può essere:

- ▷ "presa elettrica, boccola": in questo caso si "dovrebbe" utilizzare al femminile;
- ▷ "zoccolo, connettore": si "potrebbe" quindi anche utilizzare il maschile.

Di fatto la regola che viene utilizzata in generale è... "in base a come suona": se suona bene maschile si utilizza al maschile, se suona bene al femminile si utilizza il femminile!

Nella letteratura tecnica la maggioranza riporta **socket** al maschile ma è spesso presente anche al femminile.

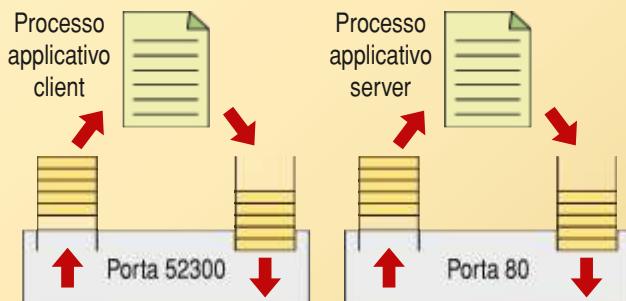
## Socket e i processi client-server

Il modello **client-server** è organizzato in due moduli, chiamati appunto **server** e **client**, operanti su macchine diverse:

- il **server** svolge le operazioni necessarie per realizzare un servizio;
- il **client**, generalmente tramite una interfaccia utente, acquisisce i dati, li elabora e li invia al **server** richiedendo un servizio.

Quindi in rete sono presenti calcolatori su cui girano processi **server** che erogano servizi e sono in attesa di ricevere richieste di connessione da parte di processi **client** interessati a usufruire di tali servizi.

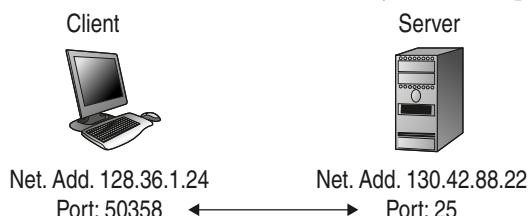
Il **client** deve conoscere l'indirizzo IP e il numero di porta usato dal **server** per potersi collegare al **socket di destinazione**: per esempio, il servizio **HTML** del **server** è sulla porta 80 mentre quello del **client** è libero, a scelta tra quelli disponibili (per esempio 52300), ma deve essere trasmesso al **client** nel segmento di richiesta in modo che il **server** possa successivamente inviare la risposta.



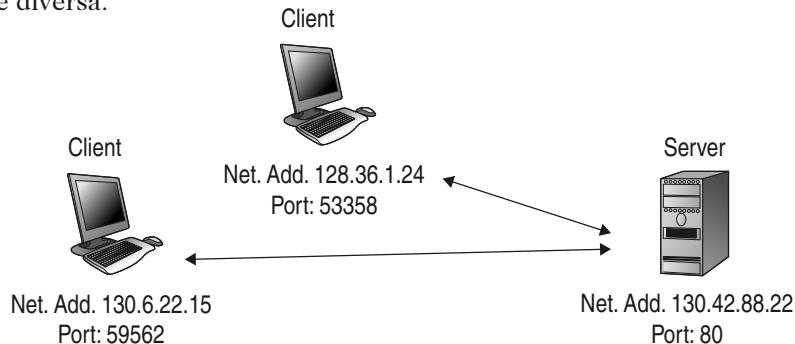
Naturalmente allo stesso **server** arrivano contemporaneamente richieste da **client** diversi che potrebbero anche utilizzare la stessa porta, come nel precedente esempio la 3301, o richieste da porte diverse dello stesso host: il collegamento avviene univocamente tramite i **socket**.

### ESEMPIO

Un **client** si connette alla porta di un **server SMTP** remoto (servizio di posta elettronica).



Due **client** accedono alla stessa porta di un **server HTTP**; non c'è comunque ambiguità, perché la coppia di **socket** è diversa.



Quando un **client** apre un **socket**, non dovrebbe utilizzare una porta nota, in quanto riservata per l'offerta di servizi.



**TCP** is a transport layer protocol used by applications that require guaranteed delivery. It is a sliding window protocol that provides handling for both timeouts and retransmissions.

TCP establishes a full duplex virtual connection between two endpoints. Each endpoint is defined by an IP address and a TCP port number. The operation of TCP is implemented as a finite state machine.

The byte stream is transferred in segments. The window size determines the number of bytes of data that can be sent before an acknowledgement from the receiver is necessary.

The Internet Protocol (**IP**) is the method or protocol by which data is sent from one computer to another on the Internet. Each computer (known as a host) on the Internet has at least one IP address that uniquely identifies it from all other computers on the Internet.

**UDP** (User Datagram Protocol) is a transport layer protocol defined for use with the IP network layer protocol. It is defined by RFC 768 written by John Postel. It provides a best-effort datagram service to an End System (IP host).

The service provided by UDP is an unreliable service that provides no guarantees for delivery and no protection from duplication (e.g. if this arises due to software errors within an Intermediate System (IS)). The simplicity of UDP reduces the overhead from using the protocol and the services may be adequate in many cases.

**UDP** provides a minimal, unreliable, best-effort, message-passing transport to applications and upper-layer protocols. Compared to other transport protocols, UDP and its UDP-Lite variant are unique in that they do not establish end-to-end connections between communicating end systems. UDP communication consequently does not incur connection establishment and teardown overheads and there is minimal associated end system state.

Because of these characteristics, **UDP** can offer a **very efficient communication transport** to some applications, but has no inherent reliability. A second unique characteristic of UDP is that it provides no inherent congestion control mechanisms. On many platforms, applications can send UDP datagrams at the line rate of the link interface, which is often much greater than the available path capacity, and doing so would contribute to congestion along the path, applications therefore need to be designed responsibly.

One increasingly popular use of UDP is as a **tunneling protocol**, where a tunnel endpoint encapsulates the packets of another protocol inside UDP datagrams and transmits them to another tunnel endpoint, which decapsulates the UDP datagrams and forwards the original packets contained in the payload. Tunnels establish virtual links that appear to directly connect locations that are distant in the physical Internet topology, and can be used to create virtual (private) networks. Using UDP as a tunneling protocol is attractive when the payload protocol is not supported by middleboxes that may exist along the path, because many middleboxes support UDP transmissions.

UDP does not provide any communications security. Applications that need to protect their communications against eavesdropping, tampering, or message forgery therefore need to separately provide security services using additional protocol mechanisms.

An **acknowledge** character (ACK) is a transmission control character transmitted by the receiving station as an acknowledgement, an affirmative response to the sending station.

A network **socket** is an endpoint of a connection across a computer network. Today, most communication between computers is based on the Internet Protocol; therefore most network sockets are Internet sockets. More precisely, a socket is a handle (abstract reference) that a local program can pass to the networking application programming interface (API) to use the connection, for example "send this data on this socket". Sockets are often represented internally as simple integers, which identify which connection to use.

The term **association** is used to completely specify the two processes that comprise a connection: (protocol,local-address,local-port,foreign-address,foreign-port).

## Verifichiamo le conoscenze

### 1. Risposta multipla

**1** Quale tra le seguenti non è una applicazione di rete?

- a. Posta elettronica
- b. Condivisione di file P2P
- c. Scheduler dei processi
- d. Telefonia via Internet
- e. Videoconferenza in tempo reale
- f. TV in streaming
- g. Telnet

**2** Qual è il significato di API?

- a. Application Protocol Internet
- b. Application Protocol Interface
- c. Application Programming Interface
- d. Application Programming Internet

**3** Quale tra i seguenti servizi offerti non è garantito delle strato di trasporto alle applicazioni?

- a. Trasferimento dati affidabile
- b. Ampiezza di banda
- c. Velocità di comunicazione
- d. Temporizzazione
- e. Sicurezza

**4** In cosa consiste una association?

- a. Nella coppia protocollo di connessione – socket locale
- b. Nella coppia protocollo di connessione – socket remoto
- c. Nella coppia socket locale – socket remoto
- d. Nella ennupla protocollo di connessione – socket locale – socket remoto

**5** Che cosa si intende per port address?

- a. l'indirizzo fisico di una porta
- b. il numero associato a una porta logica
- c. l'indirizzo di memoria di una porta
- d. l'indirizzo TPC/IP di un PC

**6** A che cosa serve l'association?

- a. Consente di identificare ogni singola connessione in modo univoco
- b. Consente di identificare ogni singolo pc nella rete
- c. Consente di associare una porta logica a una porta fisica
- d. Consente di associare un PC a un indirizzo IP

### 2. Associazione

**1** Associa le seguenti porte (7, 21, 22, 80, 110) ai servizi.

- a) ..... HTTP
- b) ..... FTP
- c) ..... POP3
- d) ..... ECHO
- e) ..... TELNET

### 3. Vero o falso

- |   |     |
|---|-----|
| <b>1</b> Il protocollo HTTP è a livello di trasporto.   | V F |
| <b>2</b> Il protocollo TCP è a livello di applicazione.   | V F |
| <b>3</b> L'applicazione di rete prende anche il nome di applicazione distribuita.                     | V F |
| <b>4</b> L'applicazione di rete prende anche il nome di applicazione multiutente.                     | V F |
| <b>5</b> Sullo stesso host possono essere in esecuzione molti processi.                               | V F |
| <b>6</b> Sullo stesso host può essere in esecuzione una sola applicazione.                            | V F |
| <b>7</b> Un protocollo è una parte integrante di un'applicazione ed è sviluppato all'interno di essa. | V F |
| <b>8</b> In una architettura client-server gli indirizzi IP devono essere statici.                    | V F |
| <b>9</b> Un socket è composto da un numero di porta di un processo e dall'indirizzo IP del terminale. | V F |
| <b>10</b> Prende il nome di indirizzo del socket un numero di porta concatenato a un indirizzo IP.    | V F |

# La connessione tramite socket

In questa lezione impareremo...

- ▷ le famiglie e le tipologie di socket
- ▷ le modalità di connessione col protocollo TCP e UDP
- ▷ la comunicazione multicast

## ■ Generalità

Il concetto di **socket** è stato sviluppato come estensione diretta del paradigma **UNIX** di **I/O su file**, che si basa sulla sequenza di operazioni **open-read-write-close**:

- ▷ **open**: permette di accedere a un file;
- ▷ **read/write**: accedono ai contenuti del file;
- ▷ **close**: terminazione dell'utilizzo del file.

L'utilizzo dei **socket** avviene pressoché con la stessa modalità ma aggiungendo a questa struttura l'insieme dei parametri necessari a realizzare la connessione tra macchine remote, cioè richiedendo:

- ▷ gli indirizzi;
- ▷ il protocollo e numero di porta;
- ▷ il tipo del protocollo.

Ogni sistema operativo mette a disposizione nelle **API** i meccanismi per realizzare l'interfacciamento tra diversi protocolli: le **socket API** sono delle specifiche **API** di protocollo che hanno origine con **Berkeley BSD UNIX** e che oggi sono disponibili in **Windows**, **Solaris** e **Linux**.

Elenchiamo le funzioni presenti sia in **C** che in **Java** che verranno utilizzate in seguito:

- ▷ **socket()**/**server socket()**: crea un nuovo socket;
- ▷ **close()**: termina l'utilizzo di un socket;
- ▷ **bind()**: collega un indirizzo di rete a un socket;
- ▷ **listen()**: aspetta messaggi in ingresso;
- ▷ **accept()**: comincia a utilizzare una connessione in ingresso;
- ▷ **connect()**: crea una connessione con un host remoto;
- ▷ **send()/write()**: trasmette dati su una connessione attiva;
- ▷ **recv()/read()**: riceve dati da una connessione attiva.

## ■ Famiglie e tipi di socket

### Famiglie di socket

Esistono varie famiglie di **socket** (chiamate anche **domini**) dove ogni famiglia riunisce i **socket** che utilizzano gli stessi protocolli (**Protocol Family**) sottostanti: ogni famiglia supporta un sottoinsieme di stili di comunicazione e possiede un proprio formato di indirizzamento (**Address Family**).

Tra le famiglie di **socket** ricordiamo:

- ▷ **Internet socket (AF\_INET)**: è quella che utilizzeremo nelle nostre applicazioni e permette il trasferimento di dati tra processi posti su macchine remote connesse tramite una **LAN** o **Internet**;
- ▷ **Unix Domain socket (AF\_UNIX)**: permette il trasferimento di dati tra processi sulla stessa macchina Unix.

La principale differenza tra un **socket** nei domini **AF\_UNIX** e **AF\_INET** è l'indirizzo:

- ▷ **AF\_UNIX** è praticamente il **pathname** valido nel file system della macchina;
- ▷ **AF\_INET** è specificato da due valori:
  - **indirizzo IP (32 bit)**, che individua un unico host Internet;
  - **numero di porta (16 bit)**, che specifica una particolare porta dell'host.

Noi tratteremo solo i **socket AF\_INET**.

### ESEMPIO

L'indirizzo di un **socket** in linguaggio **C** per la famiglia **AF\_INET** nella libreria **netinet/in.h** (che naturalmente deve essere inclusa nelle applicazioni) è definito dal seguente record:

```
struct sockaddr_in {
    short int sin_family;           /* famiglia          */
    struct in_addr sin_addr;        /* indirizzo internet */
    unsigned short int sin_port;    /* numero di porta   */
};
```

e una sua istanza è la seguente:

```
sin_family: AF_INET;
sin_addr: 127.0.0.1;
sin_port: 6916;
```

Nel linguaggio **Java** non è necessario fare alcuna precisazione sulla famiglia in quanto di default viene utilizzato il dominio **AF\_INET** nella versione **v4**, cioè con gli indirizzi composti da 4 ottetti.

### Tipi di socket

I **socket** sono fondamentalmente di tre tipi e per ciascuno abbiamo una diversa modalità di connessione:

- ▷ **stream socket**;
- ▷ **datagram socket**;
- ▷ **raw socket**.

Gli **stream socket** e **datagram socket** sono usati a livello applicativo mentre i **raw sockets** sono utilizzati nello sviluppo di protocolli e quindi esulano dagli obiettivi di questa trattazione.

## Stream socket

Con gli **stream socket (SOCK\_STREAM)** si realizza una connessione sequenziale tipicamente asimmetrica, affidabile e full-duplex basata su stream di byte di lunghezza variabile.

Operativamente, ogni processo crea il proprio **endpoint** richiamando la primitiva `socket()` in C o creando l'oggetto `socket` in Java e successivamente:

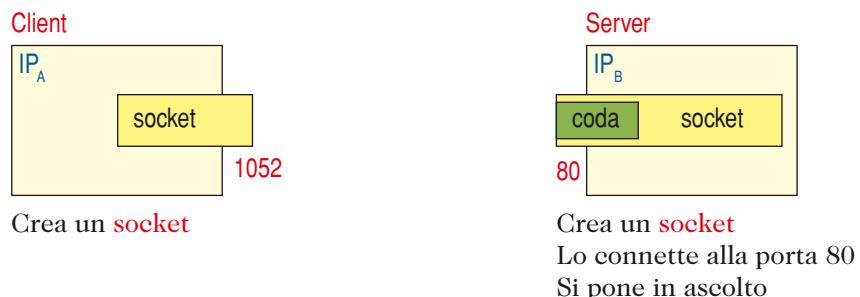
- ▶ il **server** si mette in ascolto, in attesa di un collegamento, e quando gli arriva una richiesta la esaudisce mediante la primitiva `accept()` che crea un nuovo `socket` dedicato alla connessione;
- ▶ il **client** si pone in coda sul `socket` del server e quando viene “accettato” dal **server** crea implicitamente il **binding** con la porta locale.

Tra i due processi **server** e **client**, il **server** è quello che ha controllo maggiore poiché è il processo che inizialmente crea il `socket`: più **client** possono comunicare attraverso lo stesso `socket` ma solo un **server** può essere associato a uno specifico `socket`.

Inoltre il **client** ha bisogno di conoscere l'indirizzo del **server** mentre il **server** acquisisce le informazioni del **client** (e, quindi, anche il suo indirizzo) solo dopo che viene stabilita la connessione.

Vediamo graficamente i singoli passaggi che realizzano la connessione tra **server** e **client**.

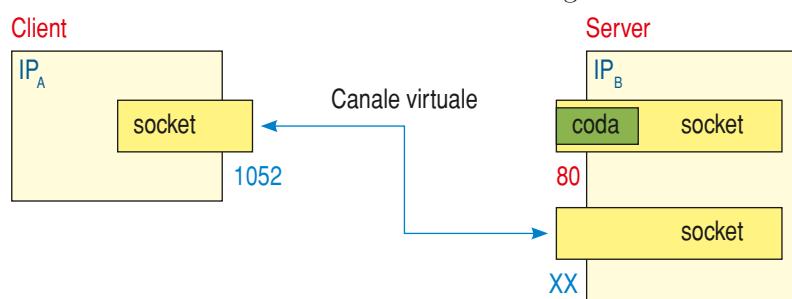
### A Inizializzazione dei processi.



### B Il client effettua la richiesta di collegamento.



### C Il server accetta la richiesta e stabilisce il collegamento.



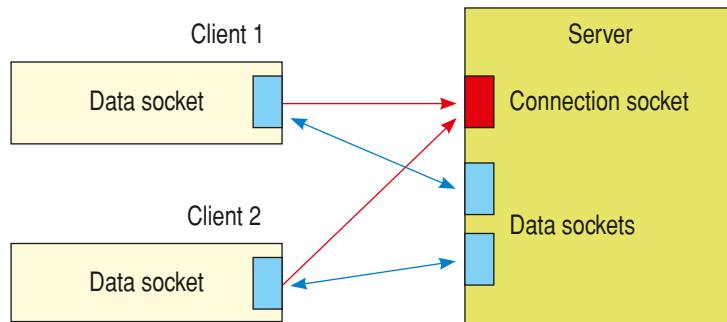
Il **server** accetta la richiesta e realizza “un canale virtuale” tra il **client** e un nuovo `socket` del **server** (indicato **XX**), in modo da lasciare il primo libero per le ulteriori richieste da parte di altri **client**.

I due processi si scambiano i dati (funzioni `read()` e `write()`) fino alla chiusura del canale, che viene effettuata rispettivamente mediante la chiamata della primitiva `close()`.

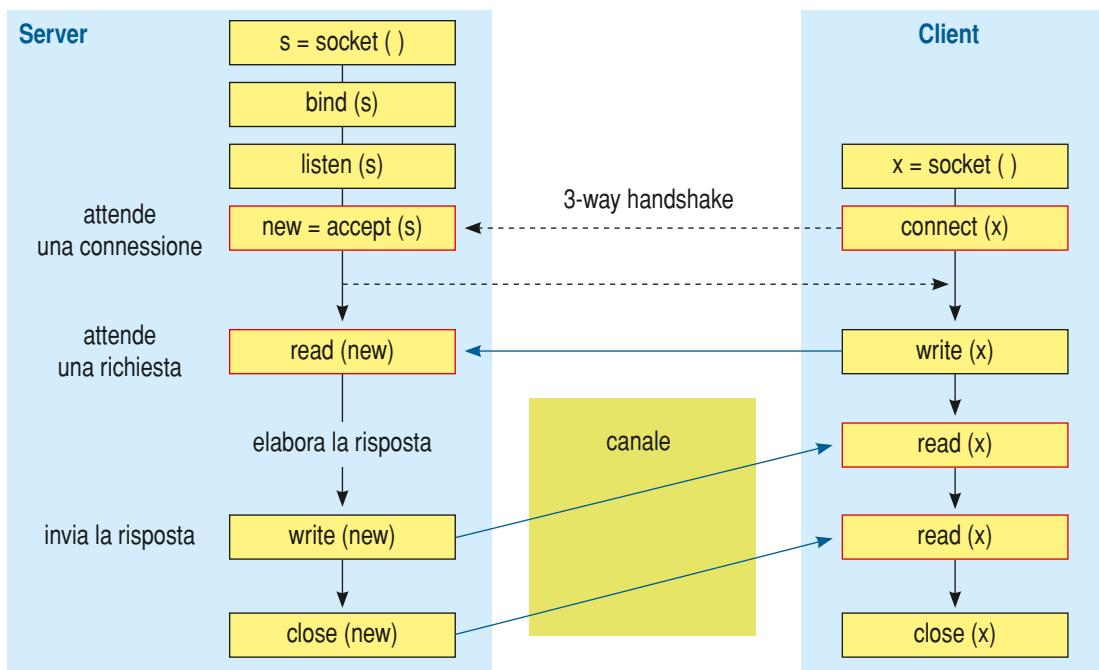
Il protocollo **TCP** è basato su questo tipo di **socket**.

Possiamo osservare che in un **server TCP** abbiamo due tipi di **socket**:

- un tipo per accettare connessioni (condiviso), che chiamiamo **connection socket**;
- un tipo per le operazioni **send()** e **receive()** (non condiviso), che chiamiamo **data socket**.



Lo schema logico completo della comunicazione **TCP** mediante **socket** nel linguaggio **C** è il seguente:



Le **API** del linguaggio **Java** differenziamo i due tipi di **socket** e noi utilizzeremo due entry differenti:

- **ServerSocket(int port)**: per creare una **connection socket** legata alla porta specificata per accettare connessioni da parte del server;
- **Socket(InetAddress address, int port)**: per creare una **data socket** e richiede come parametro una **server socket**, cioè l'esistenza di una connessione (l'indirizzo e la porta).

## Datagram socket

Con i **datagram socket (SOCK\_DGRAM)** viene realizzata la comunicazione che permette di scambiare dati senza connessione (i messaggi contengono l'indirizzo di destinazione e provenienza) mediante il trasferimento di datagrammi che inoltrano messaggi di dimensione variabile, senza garantire ordine o arrivo dei pacchetti, quindi si ha una comunicazione inaffidabile.

Permettono quindi di inviare da un **socket** a più destinazioni e ricevere su un **socket** da più sorgenti: in generale realizzano quindi il modello “molti a molti” e sono supportate nel dominio Internet dal protocollo **UDP**.

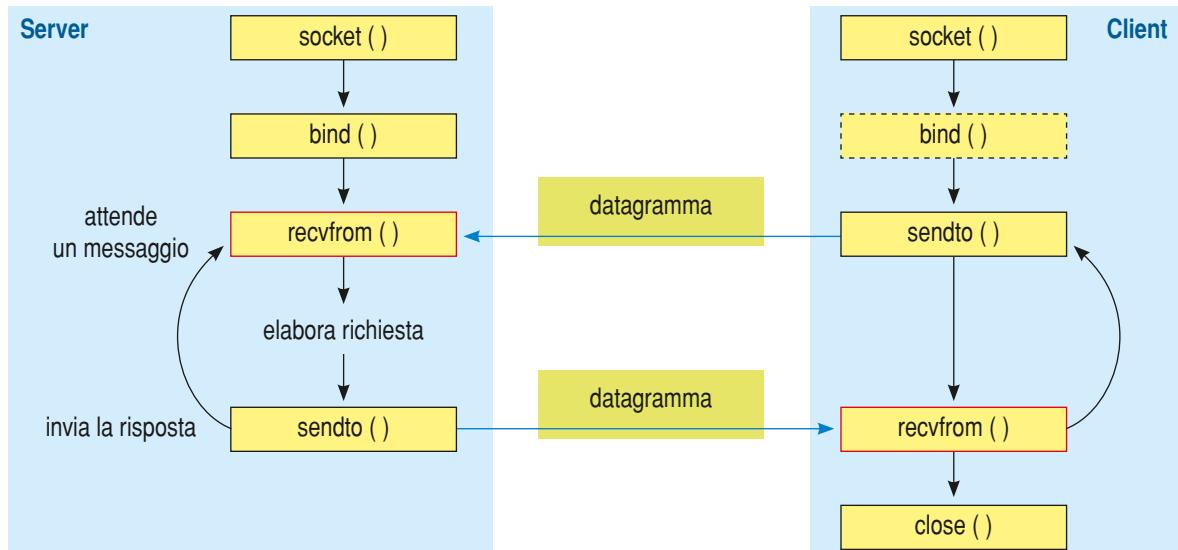
Operativamente, ogni processo crea il proprio **endpoint** richiamando la primitiva che crea il **socket** e successivamente:

- il **server** si mette in attesa di ricevere i dati (mediante la primitiva **receive()** in Java e **recvfrom()** in C) e alla loro ricezione può inviare un risposta (mediante la primitiva **send()** in Java e **sendto()** in C);
- il **client** invia il pacchetto di dati al **server** (mediante la primitiva **send()** in Java e **sendto()** e in C) e può mettersi in attesa di una risposta con le stesse primitive che ha utilizzato il **server** (cioè mediante la primitiva **receive()** in Java e **recvfrom()** in C).

Al termine della comunicazione il **socket** viene chiuso.

Il loro vantaggio è quello di trasferire velocemente i dati e non esistono differenze tra le chiamate effettuate dai vari processi coinvolti nella comunicazione: quindi “decade” anche il concetto di **client** e **server** dato che utilizzano le stesse primitive e solo il loro ordine stabilisce nella applicazione quale processo fa la funzione di **server** e quale di **client**.

Lo schema logico completo della comunicazione **UDP** in linguaggio C mediante **socket** è il seguente:



## ■ Trasmissione multicast

Nella comunicazione di tipo **multicast** un insieme di processi formano un gruppo di **multicast** e un messaggio spedito da un processo a quel gruppo viene recapitato a tutti gli altri partecipanti appartenenti al gruppo.

Tipiche applicazioni **multicast** sono per esempio:

- **usenet news**: pubblicazione e diffusione di nuove notizie in tempo reale;
- **videoconferenze**: ogni host genera un segnale audio video che viene ricevuto dagli host associati ai partecipanti alla videoconferenza;

- **massive multiplayer games:** giochi di ruolo e Internet games dove un elevato numero di giocatori interagiscono in un mondo virtuale;
- **DNS (Domain Name System):** aggiornamenti delle tabelle di naming inviati a gruppi di DNS;
- altre applicazioni quali chat, instant messaging, applicazioni p2p ecc.

Per implementare un sistema **multicast** è necessario poter definire uno schema di indirizzamento dei gruppi e un supporto che registri la corrispondenza tra un gruppo e i partecipanti oltre alla possibilità di ottimizzare l'uso della rete nel caso di invio di pacchetti a un gruppo (tramite multicast router). Il protocollo più utilizzato per il **multicast** in Internet è l'**IGMP (Internet Group Management Protocol)** che serve a garantire la trasmissione, tra **host** e **multicast router** a essi direttamente collegati, dei messaggi relativi alla costituzione dei gruppi: per esempio fornisce a un **host** i mezzi per informare il **multicast router** a esso più vicino che un'applicazione vuole unirsi a un determinato gruppo multicast utilizzando normali datagrammi IP.

Le API multicast devono quindi contenere primitive per:

- **unirsi** a un gruppo di multicast (**join**): identificare e indirizzare univocamente un gruppo;
- **lasciare** un gruppo di multicast (**leave**);
- **spedire** messaggi a un gruppo, cioè a tutti i processi che in quel momento fanno parte del gruppo;
- **ricevere** messaggi indirizzati a un gruppo del quale l'**host** fa parte.

La gestione dei gruppi è di tipo dinamico:

- un **host** può unirsi o abbandonare un gruppo in qualsiasi momento e può appartenere contemporaneamente a più gruppi;
- non è necessario appartenere a un gruppo per poter inviare a esso dei messaggi;
- i membri del gruppo possono appartenere alla medesima rete o a reti fisiche differenti.

Come indirizzo di multicast viene utilizzato un indirizzo IP di classe D, del tipo:

1110 XXXX	.....
-----------	-------

cioè con intervallo tra 224.0.0.0 – 239.255.255.255

Gli indirizzi possono essere:

- **permanenti:** l'indirizzo di **multicast** viene assegnato dalla IANA (*Internet Assigned Numbers Authority*) e rimane assegnato a quel gruppo anche se in un certo momento non ci sono più partecipanti connessi; questi indirizzi sono detti **well-known**;
- **temporanei:** richiedono la definizione di un opportuno protocollo per evitare conflitti nell'attribuzione degli indirizzi ai gruppi ed esistono solo fino al momento in cui esiste almeno un partecipante.

Nei **router** viene mantenuta la corrispondenza tra l'indirizzo di **multicast** e gli indirizzi IP dei singoli host che partecipano al gruppo di **multicast**.

La comunicazione **multicast** utilizza il paradigma **connectionless** dato che devono essere gestite contemporaneamente un alto numero di connessioni.

È anche possibile associare un nome simbolico a un gruppo di **multicast**: nella schermata a fianco, ottenuta all'indirizzo <http://ip-lookup.net> digitando come **IP address** 240.0.0.1, è possibile vedere il nome di una parte di questi gruppi. ►

	IP address	Host name
1	224.0.0.0	base-address.mcast.net
2	224.0.0.1	all-systems.mcast.net
3	224.0.0.2	all-routers.mcast.net
4	224.0.0.3	?
5	224.0.0.4	dvmrp.mcast.net
6	224.0.0.5	ospf-all.mcast.net
7	224.0.0.6	ospf-dsig.mcast.net
8	224.0.0.7	st-routers.mcast.net
9	224.0.0.8	st-hosts.mcast.net
10	224.0.0.9	rip2-routers.mcast.net
11	224.0.0.10	igmp-routers.mcast.net
12	224.0.0.11	mobile-agents.mcast.net
13	224.0.0.12	dhcp-agents.mcast.net
14	224.0.0.13	pim-routers.mcast.net
15	224.0.0.14	rsvp-encapsulation.mcast.net

**ESEMPIO**

**Java**, come vedremo, mette a disposizione un servizio **multicast** nelle API standard (la classe **MultiCastSocket**) che estende la **DatagramSocket** con alcune funzionalità utili per il multicast e permette di definire **socket** su cui spedire/ricevere i **messaggi** verso/da un gruppo di **multicast**.

**SOCKET TYPES**

There are four types of sockets available to the users. The first two are the most commonly used ones, while the last two are rarely used.

Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

- ▶ **Stream Sockets** – Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order – "A, B, C". These sockets use TCP (*Transmission Control Protocol*) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.
- ▶ **Datagram Sockets** – Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets – you build a packet with the destination information and send it out. They use UDP (*User Datagram Protocol*).
- ▶ **Raw Sockets** – These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.
- ▶ **Sequenced Packet Sockets** – They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the *Network Systems* (NS) socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the *Sequence Packet Protocol* (SPP) or *Internet Datagram Protocol* (IDP) headers on a packet or a group of packets, either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

- ▶ **TCP Socket API** – The sequence of function calls for the client and a server participating in a TCP connection is presented in Figure p. 124.

As shown in the figure, the steps for establishing a TCP socket on the client side are the following:

- Create a socket using the **socket()** function;
- Connect the socket to the address of the server using the **connect()** function;
- Send and receive data by means of the **read()** and **write()** functions.
- Close the connection by means of the **close()** function.

- ▶ **UDP Socket API**

There are some fundamental differences between TCP and UDP sockets. UDP is a connection-less, unreliable, datagram protocol (TCP is, instead, connection-oriented, reliable, and stream based). There are some instances when UDP is used instead of TCP. Some popular applications built around UDP are DNS, NFS, SNMP and, for example, some Skype services and streaming media.

Figure p. 125 shows the interaction between a UDP client and server. First of all, the client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the **sendto** function which requires the address of the destination as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the **recvfrom** function, which waits until data arrives from a client. **recvfrom** returns the IP address of the client, along with the datagram, so the server can send a response to the client.

## Verifichiamo le conoscenze



### 1. Risposta multipla

**1** L'utilizzo dei socket richiede l'insieme dei parametri necessari a realizzare la connessione, ovvero:

- a. gli indirizzi
- b. il protocollo e numero di porta
- c. il tipo del protocollo
- d. il tipo di sistema operativo

**2** Possiamo individuare tre tipi fondamentali di modalità di connessione mediante socket:

- a. stream socket
- b. datagram socket
- c. packet socket
- d. raw socket

**3** Con gli stream socket (SOCK\_STREAM) si realizza una connessione (indicare quello errato):

- a. sequenziale
- b. asimmetrica
- c. affidabile
- d. full-duplex
- e. con stream di byte di lunghezza fissa

**4** In un server TCP abbiamo due tipi di socket:

- a. connection socket
- b. stream socket
- c. packet socket
- d. data socket
- e. raw socket

**5** Con i datagram socket (SOCK\_DGRAM) si realizza una connessione (indicare quelli errati):

- a. molti a molti
- b. affidabile
- c. full-duplex
- d. con stream di byte di lunghezza fissa
- e. senza connessione

**6** Tipiche applicazioni multicast sono per esempio (indicare quella errata):

- a. usenet news
- b. videoconferenze
- c. pop mail
- d. massive multiplayer games
- e. DNS (Domain Name System)
- f. chat e instant messaging



### 2. Vero o falso

**1** La funzione bind collega un indirizzo di rete a un socket.

V F

**2** AF\_UNIX è praticamente il pathname valido nel file system della macchina.

V F

**3** AF\_INET è specificato da due valori, indirizzo IP (32 bit) e numero di porta (8 bit).

V F

**4** Il linguaggio Java di default utilizza il dominio AF\_INET nella versione v4.

V F

**5** Gli stream e datagram socket sono usati a livello applicativo.

V F

**6** Il protocollo TCP è basato su stream socket.

V F

**7** Con i datagram socket viene garantito l'ordine di arrivo dei pacchetti.

V F

**8** Nella comunicazione di tipo multicast un messaggio spedito da un processo a quel gruppo viene recapitato a tutti gli altri partecipanti appartenenti al gruppo.

V F

**9** La comunicazione di tipo multicast coinvolge un gruppo di host.

V F

**10** Come indirizzo di multicast viene utilizzato un indirizzo IP di classe E.

V F

# ESERCITAZIONI DI LABORATORIO 1

## JAVA SOCKET

### ■ Java socket: caratteristiche della comunicazione

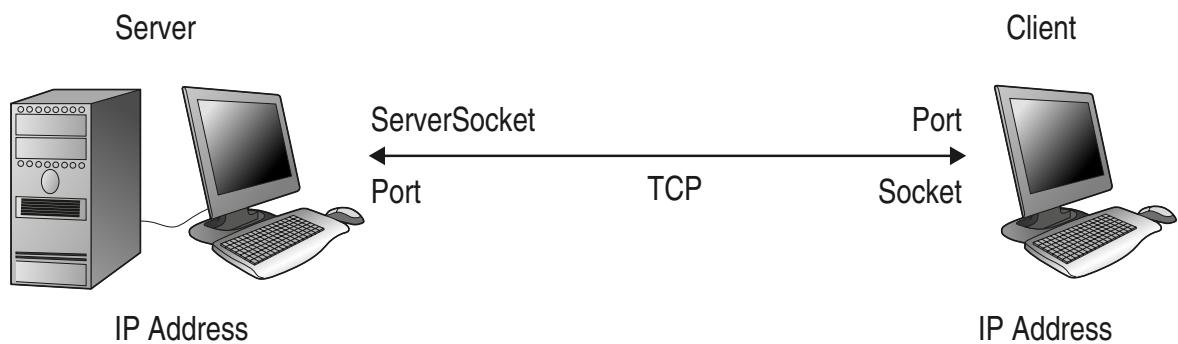
Si è detto che un **socket** è come una porta di comunicazione e tutto ciò che è in grado di comunicare tramite il protocollo standard **TCP/IP** può collegarsi a un **socket** e comunicare attraverso di esso; sappiamo anche che le informazioni “spedite” da un **socket** a un altro prendono il nome di pacchetti **TCP/IP**.

Riassumiamo le caratteristiche fondamentali della comunicazione:

- ▶ tipicamente, la comunicazione è punto-punto (**unicast**), ma esistono estensioni che permettono comunicazioni multi-punto (**multicast**);
- ▶ la sorgente della comunicazione deve conoscere l'identità (indirizzo IP e numero di porta) del destinatario;
- ▶ la **serializzazione** (trasformazione tra dati strutturati e sequenze di byte) e il **marshalling** (conversione tra rappresentazioni di dati diversi) sono a carico delle applicazioni.

Le operazioni necessarie per un trasferimento di dati tra due host sono le seguenti:

- 1 un host che si comporta da **server** apre un canale di comunicazione su una determinata porta e rimane in ascolto, in attesa di una richiesta di connessione (**ServerSocket**);
- 2 un host **client** fa una richiesta di connessione a un **server** (con indirizzo IP e address port conosciuti: **socket** sul **client**);
- 3 il **server** accetta la connessione del **client** e così viene instaurato un canale di comunicazione tra i due host.



Java utilizza la classe **Socket** per la creazione degli oggetti che permettono di utilizzare i **socket** e quindi di stabilire un canale di comunicazione tra un **client** e un **server** attraverso il quale si comunica utilizzando due stream particolari, specializzati per i **socket**, uno per l'input e l'altro per l'output.

Vediamo dapprima singolarmente le classi che Java mette a disposizione per l'utilizzo dei **socket** e successivamente implementiamo un **server** e un **client**:

- ▷ classe **InetAddress**;
- ▷ classe **ServerSocket**;
- ▷ classe **Socket**.

## ■ Classe InetAddress

Un indirizzo di rete **IP v4** è costituito da 4 numeri (da 0 a 255) separati ciascuno da un punto e il **DNS** permette di utilizzare in alternativa il nome simbolico per individuare un particolare host.

La classe **InetAddress** mette a disposizione diversi metodi per astrarre dal particolare tipo di indirizzo specificato (a numeri o a lettere), occupandosi essa stessa di effettuare le dovute traduzioni.

Il diagramma completo della classe è il seguente:

Classe InetAddress
Metodi modificatori
<pre>InetAddress[] getAllByName(String host) InetAddress getByAddress(byte[] addr) InetAddress getByAddress(String host, byte[] addr) InetAddress getByName(String host) InetAddress getLocalHost()</pre>
<pre>byte[] getAddress() String getCanonicalHostName() String getHostAddress() String getHostByAddr(byte[] addr) boolean isAnyLocalAddress() boolean isLinkLocalAddress() boolean isLoopbackAddress() boolean isMCGlobal() boolean isMCLinkLocal() boolean isMCNodeLocal() boolean isMCOrgLocal() boolean isMC SiteLocal() boolean isMulticastAddress() boolean isSiteLocalAddress()</pre>
<pre>boolean equals(Object obj) int hashCode() String toString()</pre>
<pre>byte[][] lookupAllHostAddr(String host) Object run()</pre>

Non sono previsti costruttori e l'unico modo per creare un oggetto di classe **InetAddress** prevede l'utilizzo di *metodi statici*, e in particolare:

- ▷ **public static InetAddress getByName(String host)**

restituisce un oggetto **InetAddress** rappresentante l'host specificato nel parametro host; l'host può essere specificato sia col nome sia con l'indirizzo numerico e se si specifica null come parametro, ci si riferisce all'indirizzo di default della macchina locale;

► **public static InetAddress getLocalHost()**

viene restituito un Inet-Address corrispondente alla macchina locale; se tale macchina non è registrata, oppure è protetta da un firewall, l'indirizzo è quello di **loopback**: 127.0.0.1.

Se l'indirizzo specificato non può essere risolto tramite il **DNS**; questi metodi possono sollevare l'eccezione **UnknownHostException**.

Di particolare utilità sono i tre metodi illustrati di seguito:

**1 public String getHostName()**

restituisce il nome **dell'host** che corrisponde all'indirizzo **IP** dell'InetAddress; se il nome non è ancora noto (per esempio se l'oggetto è stato creato specificando un indirizzo **IP** numerico), verrà cercato tramite il **DNS**; se tale ricerca fallisce, verrà restituito l'indirizzo **IP** numerico sotto forma di stringa.

**2 public String getHostAddress()**

simile al precedente, restituisce però l'indirizzo **IP** numerico, sotto forma di stringa, corrispondente all'oggetto **InetAddress**.

**3 public byte[] getAddress()**

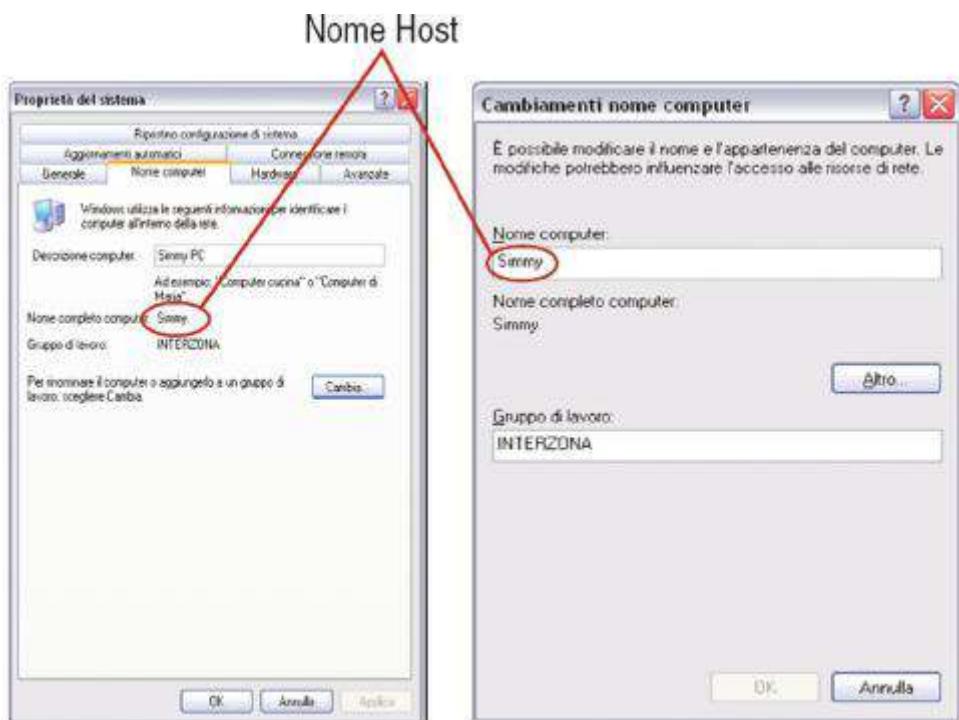
l'indirizzo **IP** numerico restituito sarà sotto forma di matrice di byte; l'ordinamento dei byte è **high byte first** (che è l'ordinamento tipico della rete).

Per esempio, con:

```
String indirizzo = InetAddress.getLocalHost().getHostAddress();
```

l'oggetto *indirizzo* conterrà l'indirizzo **IP** della macchina locale.

La figura che segue rappresenta un esempio del nome dell'host riconosciuto dalla classe **InetAddress** e dal **DNS**.



## ■ Classe ServerSocket

La classe **ServerSocket**, il cui diagramma è riportato di seguito, viene utilizzata per accettare connessioni da parte del **client**.

Classe ServerSocket	
Metodi costruttori	Metodi modificatori
<code>ServerSocket()</code>	<code>void setSpcketFactory(SocketImplFactory fac)</code>
<code>ServerSocket(int port)</code>	<code>ServerSocketChannel getChannel()</code>
<code>ServerSocket(int port, int backlog)</code>	<code>InetAddress getInetAddress()</code>
<code>ServerSocket(int port, int backlog, InetAddress bindAddr)</code>	<code>int getLocalPort()</code>
	<code>SocketAddress getLocalSocketAddress()</code>
	<code>int get/setReceiveAddress()</code>
	<code>boolean get/setReuseAddress()</code>
	<code>int get/setSoTimeout()</code>
	<code>boolean isBound()</code>
	<code>boolean isClosed()</code>
	<code>String toString()</code>
	<code>Socket accept()</code>
	<code>void bind(SocketAddress endpoint)</code>
	<code>void bind(SocketAddress endpoint, int backlog)</code>
	<code>void close()</code>
	<code>void implAccept(Socket s)</code>

Questa classe deve essere istanziata passando come parametro il numero della porta su cui il **server** sarà in ascolto: `ServerSocket(int port)`. L'unico metodo realmente necessario è `accept()`: mediante tale metodo l'oggetto rimane in attesa di richiesta di connessioni da parte di un **client** sulla porta specificata nel costruttore.

Quando una richiesta di connessione va a buon fine viene creato il canale di collegamento e il metodo restituisce un oggetto **Socket** connesso con il **client**.

Per esempio:

```
// creo un server sulla porta 6789
ServerSocket server = new ServerSocket(6789);
// rimane in attesa di un client
Socket client = server.accept();
// chiudo il server per inibire altri client
server.close();
```

Vediamo nel dettaglio le singole istruzioni:

- il **server** si mette in ascolto sulla porta 6789;
- quando riceve una richiesta di connessione, viene creato un oggetto **client** di classe **Socket** (che vedremo in seguito); questo oggetto rappresenta il canale di comunicazione **server-client**;
- con la terza istruzione si chiude il **server**: ciò implica una comunicazione di tipo **Unicast**, dato che, dopo avere instaurato una connessione, il **server** non rimane più in ascolto per alcuna ulteriore richiesta di connessione.

Per poter utilizzare una connessione **multicast** (che realizzeremo più avanti), si dovrebbe instanziare un thread per ogni connessione effettuata, in modo da lasciare il **server** sempre in ascolto.

## ■ Classe Socket

La classe **Socket** permette di definire una connessione **client-server** via **TCP** su entrambi i lati, sia **client** che **server**.

La differenza tra **client** e **server** sta nella modalità di creazione di un oggetto di questo tipo:

- nel **server** l'oggetto **Socket** viene creato dal metodo **accept()** della classe **ServerSocket**;
- il **client** dovrà provvedere a creare un'istanza di **Socket**: per creare un **socket** con un **server** in esecuzione su un certo **host** è sufficiente creare un oggetto di classe **Socket** specificando nel costruttore l'*indirizzo Internet dell'host* e il *numero di porta*.

Dopo che l'oggetto **Socket** è stato istanziato, è possibile ottenere (tramite appositi metodi) due **stream** (uno di input e uno di output): tramite essi è possibile comunicare con l'**host**, e riceverne messaggi. Il diagramma della classe **Socket** è il seguente.

<b>Classe Socket</b>	
Metodi costruttori	Metodi modificatori
<pre>Socket() Socket(SocketImpl port) Socket(String host, int port) Socket(InetAddress address, int port) Socket(String host, int port, InetAddress localAddr, int localPort) Socket(InetAddress address, int port, InetAddress localAddr, int localPort)</pre>	<pre>void setSpcketImplFactory(SocketImplFactory fac)  SocketChannel getChannel() InetAddress getInetAddress() InputStream getInputStream() boolean get/setKeepAlive() InetAddress getLocalAddress() int getLocalPort() SocketAddress getLocalSocketAddress() boolean get/setOOBInline() OutputStream getOutputStream() int getPort() int get/setReceiveBufferSize() SocketAddress get/getRemoteSocketAddress() boolean get/setReuseAddress() int get/setSendBufferSize() int getSoLinger() int get/setSoTimeout() boolean get/setTopNoDelay() int get/setTrafficClass() boolean isBound() boolean isClosed() boolean isConnected() boolean isInputShutdown() boolean isOutputShutdown() void setSoLinger(boolean on, int linger)</pre>
	<pre>String toString()  void bind(SocketAddress bindpoint) void close() void connect(SocketAddress endpoint) void connect(SocketAddress endpoint, int timeout) void sendUrgentData(int data) void shutdownInput() void shutdownOutput()</pre>

Qualsiasi metodo che prenda in ingresso un **InputStream** o un **OutputStream** può comunicare con l'**host** in rete: una volta creato il **Socket** si può comunicare in rete tramite l'utilizzo degli stream.

I costruttori della classe **Socket** sono i seguenti:

- ▷ **public Socket (String host, int port) throws IOException;**
- ▷ **public Socket (InetAddress address, int port) throws IOException.**

Viene creato un oggetto **Socket** connettendosi con l'host specificato (sotto forma di stringa o di **InetAddress**) alla porta specificata. Se sull'host e sulla porta specificata non c'è un **server** in ascolto, viene generata un'**IOException** e viene specificato il messaggio **connection refused**.

Alcuni metodi tra i più utilizzati sono i seguenti:

- ▷ **public InetAddress getInetAddress()**: restituisce un oggetto **Inet-Address** corrispondente all'indirizzo dell'host con il quale il **socket** è connesso;
- ▷ **public InetAddress getLocalAddress()**: restituisce un oggetto **Inet-Address** corrispondente all'indirizzo locale al quale il **socket** è collegato;
- ▷ **static InetAddress getByname(String hostname)**: restituisce una istanza di **InetAddress** rappresentante l'host specificato;
- ▷ **public int getPort()**: restituisce il numero della porta dell'host remoto con il quale il **socket** è collegato;
- ▷ **public int getLocalPort()**: restituisce il numero di porta locale con la quale il **socket** è collegato.

Quando si crea un **socket**, come già detto, ci si collega con un **server** su un determinato host, che è in ascolto su una certa porta; sulla macchina locale viene creato il **socket** su una determinata porta assegnata dal sistema operativo (il primo numero libero):

```
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
```

Quando si comunica attraverso connessioni **TCP**, i dati vengono suddivisi in pacchetti (IP packet), quindi è consigliabile utilizzare degli stream “bufferizzati” evitando così di avere pacchetti contenenti poche informazioni.

La definizione dei due stream, rispettivamente di input e di output, da parte di un **server** verso un **client**, è per esempio la seguente:

```
BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));
DataOutputStream out = new DataOutputStream(client.getOutputStream());
```

Dopo questa definizione l'uso dei **socket** diventa quindi trasparente: su questi oggetti si utilizzano i metodi **readLine()** e **println()**!

L'ultimo metodo che descriviamo è:

```
public synchronized void close() throws IOException
```

Con questo metodo viene chiuso il **socket** (e quindi la connessione) e tutte le risorse che erano in uso vengono rilasciate.

I dati bufferizzati verranno comunque spediti prima della chiusura del **socket**, chiusura anche solo di uno dei due stream associati ai **socket** che comporterà automaticamente la chiusura del **socket** stesso.

Nei metodi precedenti può essere lanciata un'**IOException**, a significare che ci sono stati problemi sulla connessione: questo succede quando uno dei due programmi che utilizza il **socket** chiude la connessione e quindi l'altro programma potrà ricevere una tale eccezione.

## ■ La realizzazione di un client TCP in Java

Possiamo ora realizzare un **client**: per comunicare con un host remoto usando il protocollo **TCP/IP**, si deve creare, per prima cosa, un oggetto **Socket** con tale host, specificando l'indirizzo **IP** dell'host e il numero di porta (naturalmente sull'host remoto dovrà essere presente un **server** in “ascolto” su tale porta).

Possiamo utilizzare indistintamente uno dei due costruttori della classe **Socket** sopra descritta:

```
public Socket (String host, int port) throws IOException;
public Socket (InetAddress address, int port) throws IOException.
```

Con questa istruzione viene creata una connessione con un'applicazione **server** (che nel frattempo è in attesa in una **accept()**) e restituisce il relativo **socket**.

Definiamo le variabili della nostra classe:

```
import java.io.*;
import java.net.*;

public class Client2 {
    String nomeServer = "nomeServer";           // indirizzo del servér
    int portaServer = 6789;                      // porta x servizio
    DataInputStream in;                          // stream di input
    DataOutputStream out;                        // stream di output
    ...
}
```

Realizziamo ora la parte di un metodo che esegue le operazioni per aprire un **socket** con un **server** che si trova a un certo indirizzo (**nomeServer**) ed è in ascolto su una certa porta (**portaServer**) restituendo come parametro di ritorno un oggetto della classe **Socket**.

Una volta creato un oggetto **Socket** otteniamo gli stream a esso associati tramite i due metodi prima descritti della classe **Socket**: a questo punto, la comunicazione può avere inizio, cioè il **client** può scrivere sull'**OutputStream**, come si fa con un normale stream, e ricevere dati dal **server** leggendo dall'**InputStream**.

```
protected Socket connetti () throws IOException
{
    Socket socket = new Socket (nomeServer, portaServer);

    out = new DataOutputStream (socket.getOutputStream ());
    in = new DataInputStream (socket.getInputStream ());
    ...
    return socket;
}
```

L'esempio seguente permette di leggere una stringa di testo dalla tastiera di un **client**. Esso invia la stringa a un **server**, il quale la modifica trasformandola in maiuscolo e la restituisce al **client**. La comunicazione avviene sulla porta 6789 sul **server** locale localhost.

Naturalmente, non possiamo mandarlo in esecuzione senza aver realizzato il **server**, ma per ora lo scriviamo e lo commentiamo, realizzando il **server** successivamente.

Definiamo le variabili da utilizzare nella classe.

```

1 import java.io.*;
2 import java.net.*;
3 public class ClientStr {
4     String nomeServer ="localhost";           // indirizzo server locale
5     int portaServer    = 6789;                // porta x servizio data e ora
6     Socket miosocket;
7     BufferedReader tastiera;                 // buffer per l'input da tastiera
8     String stringaUtente;                   // stringa inserita da utente
9     String stringaRicevutaDalServer;         // stringa ricevuta dal server
10    DataOutputStream outVersoServer;        // stream di output
11    BufferedReader inDalServer;             // stream di input

```

Quindi definiamo il metodo che stabilisce la connessione con il **server**:

```

55 public Socket connetti(){
56     System.out.println("2 CLIENT partito in esecuzione ...");
57     try
58     {
59         // per l'input da tastiera
60         tastiera = new BufferedReader(new InputStreamReader(System.in));
61         // creo un socket
62         miosocket = new Socket(nomeServer,portaServer);
63         // miosocket = new Socket(InetAddress.getLocalHost(), 6789);
64         // associo due oggetti al socket per effettuare la scrittura e la lettura
65         outVersoServer = new DataOutputStream(miosocket.getOutputStream());
66         inDalServer    = new BufferedReader(new InputStreamReader (miosocket.getInputStream()));
67     }
68     catch (UnknownHostException e){
69         System.err.println("Host sconosciuto");
70     }
71     catch (Exception e)
72     {
73         System.out.println(e.getMessage());
74         System.out.println("Errore durante la connessione!");
75         System.exit(1);
76     }
77     return miosocket;
78 }

```

Abbiamo creato un **socket** mediante il costruttore:

```
miosocket = new Socket(nomeserver,portaserver);
```

Trattandosi di **server** locale avremmo anche potuto utilizzare la seguente istruzione:

```
miosocket = new Socket(InetAddress.getLocalHost(), 6789);
```

Per le comunicazioni **I/O** abbiamo definito tre oggetti, il primo della classe **BufferedReader** per effettuare l'input da tastiera e gli altri due, rispettivamente, della classe **DataOutputStream** e **BufferedReader** per scrivere e leggere sul **socket**.

Definiamo un secondo metodo che effettua la conversazione:

- leggiamo la stringa inserita dall'utente e la scriviamo sullo stream del miosocket: quindi ci mettiamo in attesa della risposta del **server** mediante la **inDalserver.readLine()**;

- quando riceviamo la risposta la visualizziamo sullo schermo e terminiamo l'elaborazione chiudendo la porta con **miosocket.close()**.

```

13 public void comunica() {
14     try                    // leggo una riga
15     {
16         System.out.println("4 ... inserisci la stringa da trasmettere al server:"+'\n');
17         stringaUtente = tastiera.readLine();
18         //la spedisco al server
19         System.out.println("5 ... invio la stringa al server e attendo ...");
20         outVersoServer.writeBytes( stringaUtente+'\n');
21         //leggo la risposta dal server
22         stringaRicevutaDalServer=inDalServer.readLine();
23         System.out.println("6 ... risposta dal server "+'\n'+stringaRicevutaDalServer );
24         // chiudo la connessione
25         System.out.println("9 CLIENT: termina elaborazione e chiude connessione" );
26         miosocket.close();
27     }
28     catch (Exception e)
29     {
30         System.out.println(e.getMessage());
31         System.out.println("Errore durante la comunicazione col server!");
32         System.exit(1);
33     }
34 }
```

Concludono il metodo le istruzioni del costrutto **catch** che ci segnalano gli eventuali errori.

Il main è il seguente:

```

56 public static void main(String args[]) {
57     ClientStr cliente = new ClientStr();
58     cliente.connectti();
59     cliente.comunica();
60 }
```



## Prova adesso!

Sulla base del **client** sopra descritto, scrivi un tuo **client** predisposto per connettersi a un **server** sempre presente sul tuo PC che trasmette e riceve una stringa.

La verifica della connessione verrà fatta al termine della successiva esercitazione di laboratorio oppure utilizzando come **server ServerStr.class** che puoi prelevare dal **DVD** allegato, nella cartella **source\UA3**, oppure scaricare dal sito [www.hoepliscuola.it](http://www.hoepliscuola.it) nella cartella materiali nella sezione riservata al presente volume (file **UA3vol3.java.rar**).

# ESERCITAZIONI DI LABORATORIO 2

## JAVA SOCKET: REALIZZAZIONE DI UN SERVER TCP

### ■ Realizzazione di un server

In questa esercitazione realizzeremo il **server** corrispondente al **client** definito nella precedente esercitazione, dopo aver definito le variabili:

```

1 import java.io.*;
2 import java.net.*;
3 import java.util.*;
4
5 public class ServerStr
6 {
7     ServerSocket server      = null;
8     Socket client           = null;
9     String stringaRicevuta  = null;
10    String stringaModificata = null;
11    BufferedReader inDalClient;
12    DataOutputStream outVersoClient;
13
14 }
```

Definiamo un metodo che effettua la connessione: per prima cosa si deve creare un oggetto **ServerSocket** che indichi il numero di porta sulla quale si mette in ascolto (nel nostro esempio, 6789) e mediante il metodo **accept()** aspetta un **client**.

```

19
20 public Socket attendi()
21 {
22     try
23     {
24         System.out.println("Il SERVER partito in esecuzione ...");
25         // creo un server sulla porta 6789
26         server = new ServerSocket(6789);
27         // rimane in attesa di un client
28         client = server.accept();
29         // chiudo il server per inibire altri client
30         server.close();
31         //associo due oggetti al socket del client per effettuare la scrittura e la lettura
32         inDalClient = new BufferedReader(new InputStreamReader (client.getInputStream()));
33         outVersoClient = new DataOutputStream(client.getOutputStream());
34     }
35 }
```

L'esecuzione dell'istruzione successiva avviene solamente se si è instaurata la connessione col **client** su tale porta: prima di iniziare la comunicazione, il **server** deve chiudere il **ServerSocket** per inibire ad altri **client** il collegamento, in quanto quell'indirizzo è già utilizzato (e ora noi stiamo implementando una comunicazione **Unicast**).

Conclude il metodo la gestione dell'errore di connessione:

```

27     catch (Exception e)
28     {
29         System.out.println(e.getMessage());
30         System.out.println("Errore durante l'istanza del server !");
31         System.exit(1);
32     }
33     return client;
34 }
```

Il metodo che effettua la comunicazione è molto simile a quello descritto per il **client**: dopo il saluto di benvenuto, il **server** si pone in attesa di lettura di una stringa dal canale di “input dal **client**”; al suo arrivo, la converte in maiuscolo e la trasmette al **client**.

Quindi si commiata dal **client** e chiude la connessione terminando la sua elaborazione.

```

38     public void comunica()
39     {
40
41         try
42         {
43             // rimango in attesa della riga trasmessa dal client
44             System.out.println("3 benvenuto client, scrivi una frase e la trasformo in maiuscolo. Attendo ...");
45             stringaRicevuta = inDaClient.readLine();
46             System.out.println("6 ricevuta la stringa dal cliente : "+stringaRicevuta);
47
48             //la modifco e la rispedisco al client
49             stringaModificata=stringaRicevuta.toUpperCase();
50             System.out.println("7 invio la stringa modificata al client ...");
51             outVersoClient.writeBytes(stringaModificata+'\n');
52
53             //termina elaborazione sul server : chiudo la connessione del client
54             System.out.println("9 SERVER: fine elaborazione ... buona notte!");
55             client.close();
56         }
57     }
```

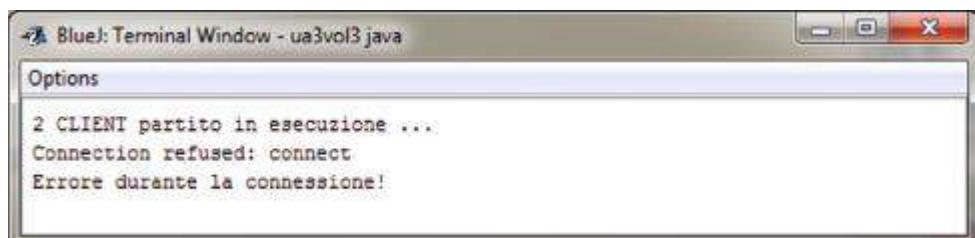
Il main è il seguente:

```

62
63     public static void main(String args[])
64     {
65         ServerStr servente = new ServerStr();
66         servente.attendi ();
67         servente.comunica();
68     }
69 }
```

Mandiamo ora in esecuzione le due classi: dato che utilizziamo la medesima macchina per entrambe le funzioni, dovremo attivare due istanze di **BlueJ**: una che ci permetterà di mandare in esecuzione il **server** e l'altra per il **client**.

L'output si presenta in due finestre diverse: per meglio comprendere la corretta sequenza di esecuzione delle operazioni queste sono state numerate in ordine progressivo. Ricordiamo di mandare in esecuzione prima il **server** altrimenti il **client** termina immediatamente con il seguente messaggio:



Effettuiamo una sequenza completa di operazioni, digitando come stringa da tradurre “ciao server”. ►

```
BlueJ: Terminal Window - ua3vol3.java
Options
1 SERVER partito in esecuzione ...
3 benvenuto client, scrivi una frase e la trasformo in maiuscolo. Attendo ...
6 ricevuta la stringa dal cliente : ciao
7 invio la stringa modificata al client ...
9 SERVER: fine elaborazione ... buona notte!
```

Nella finestra precedente sono riportate tutte le operazioni effettuate dal **server** e nella finestra seguente dal **client**. ►

```
BlueJ: Terminal Window - ua3vol3.java
Options
2 CLIENT partito in esecuzione ...
4 ... inserisci la stringa da trasmettere al server:
ciao
5 ... invio la stringa al server e attendo ...
8 ... risposta dal server
CIAO
9 CLIENT: termina elaborazione e chiude connessione
```

## AREA digitale



Un esempio di verifica



### Prova adesso!

Realizza un'applicazione **client-server** dove il **client** invia un messaggio al **server**, questo conta il numero di vocali e di consonanti, e gli ritorna tali valori: il **client** continua a inviare frasi fino a che il numero di consonanti sia esattamente la metà del numero di vocali. In questo caso termina l'applicazione.

## Esercizi proposti

- 1** Realizza una semplice calcolatrice: il client invia gli operandi e l'operatore al server, il quale esegue l'operazione e restituisce il risultato.
- 2** Realizza un sistema in cui il client riceve dal server un numero progressivo (tipo dispenser dei numeri per la coda dal panettiere).
- 3** Bomba a orologeria: un server spedisce a un client una bomba innescata con una miccia con valore random() e il client la rispedisce successivamente al server; a ogni operazione, entrambi riducono la miccia finché... la bomba scoppia.
- 4** Realizza il noto gioco della battaglia navale tra due giocatori che si trovano su due PC connessi in rete: il server provvede alle operazioni di avvio del gioco per poi diventare a tutti gli effetti il secondo giocatore.
- 5** Realizza il gioco dello “spara all’orso”: un orso si muove orizzontalmente in modo casuale e un cacciatore gli spara spostandosi anch’esso orizzontalmente (il cacciatore è mosso dal giocatore mediante i tasti freccia).
- 6** Gioco del Tris: realizza il gioco del Tris tra due giocatori connessi in rete locale.

# ESERCITAZIONI DI LABORATORIO 3

## REALIZZAZIONE DI UN SERVER MULTIPLO IN JAVA

Nella definizione del **server** nella esercitazione precedente abbiamo visto che all'inizio della comunicazione con il **client** viene chiusa la porta sulla quale il **server** era in attesa (cioè il **ServerSocket**, inizializzato sull'indirizzo 6789): infatti, tale **server** deve essere fermato per inibire ad altri **client** il collegamento contemporaneo sul medesimo indirizzo:

```
System.out.println("Il SERVER partito in esecuzione ...");
// creo un server sulla porta 6789
server = new ServerSocket(6789);
// rimane in attesa di un client
client = server.accept();
// chiudo il server per inibire altri client
server.close();
```

Spesso al **server** devono però connettersi più **client** contemporaneamente, a volte anche in numero non predeterminato (per esempio in una chat tra un **server** e un molteplice numero di **client**). La realizzazione di questo sistema avviene mediante un meccanismo **multithread** per la gestione della ricezione e dell'invio dei messaggi, in quanto non è prevedibile il numero di **client** da servire e il momento in cui ogni singolo **client** fa richiesta di invio di un nuovo messaggio.

I **server multithreaded** vengono tipicamente realizzati seguendo uno dei seguenti schemi:

- **on demand**: viene creato un **thread** per ogni **client** al momento della richiesta di connessione e viene terminato alla chiusura della comunicazione;
- **thread pool**: viene predeterminato e avviato un numero fisso di **thread** e ciascuno di essi viene assegnato alla richiesta di connessione da parte dei **client**; alla chiusura di una connessione il corrispondente **thread** non viene terminato ma viene sospeso per essere riutilizzato in una nuova connessione.

La scelta dello schema più consono varia in funzione delle situazioni, che quindi vanno accuratamente analizzate prima dell'implementazione. Infatti, ciascuno degli schemi sopra descritti presenta pregi e difetti; per esempio: il primo schema è più semplice ma più costoso (in termini di tempo, in quanto ogni volta deve essere creato un **thread**) mentre il secondo richiede un dimensionamento preventivo del pool (che potrebbe essere errato, o sottostimato e sovradimensionato).

Una soluzione ottimale può essere ottenuta con una forma intermedia, ovvero adottando uno schema misto. Modifichiamo ora l'esempio precedente per realizzare una situazione **on demand**: riscriviamo quindi il **server**.

Definiamo la classe con le necessarie variabili.

```

1 import java.net.*;
2 import java.io.*;
3 class ServerThread extends Thread {
4     ServerSocket server      = null;
5     Socket client           = null;
6     String stringaRicevuta   = null;
7     String stringaModificata = null;
8     BufferedReader  inDalClient;
9     DataOutputStream outVersoClient;

```

Modifichiamo il costruttore in questo modo:

```

10
11     public ServerThread (Socket socket){
12         this.client = socket;
13     }

```

Anche il metodo **run()** viene limitato alla chiamata di un nuovo metodo **comunica()** e alla gestione delle eccezioni:

```

14
15     public void run(){
16         try{
17             comunica();
18         }catch (Exception e){
19             e.printStackTrace(System.out);
20         }
21     }

```

Il metodo che effettua la comunicazione è costituito da un ciclo infinito dal quale si esce quando il **client** ha trasmesso la stringa **FINE**: per il resto, si limita ad acquisire una riga dallo stream di input e ritrasmetterla al **client**.

La numerazione dei messaggi aiuta a seguire il flusso della comunicazione.

```

22
23     public void comunica () throws Exception{
24         inDalClient      = new BufferedReader(new InputStreamReader (client.getInputStream()));
25         outVersoClient  = new DataOutputStream(client.getOutputStream());
26         for (;;){
27             stringaRicevuta = inDalClient.readLine();
28             if (stringaRicevuta == null || stringaRicevuta.equals("FINE")){
29                 outVersoClient.writeBytes(stringaRicevuta+" (>server in chiusura...)" + '\n');
30                 System.out.println("Echo sul server in chiusura : " + stringaRicevuta);
31                 break;
32             }
33             else{
34                 outVersoClient.writeBytes(stringaRicevuta+" (ricevuta e ritrasmessa)" + '\n');
35                 System.out.println("6 Echo sul server :" + stringaRicevuta);
36             }
37         }
38         outVersoClient.close();
39         inDalClient.close();
40         System.out.println("9 Chiusura socket" + client);
41         client.close();
42     }
43 }

```

Realizziamo ora la classe più importante, cioè quella che rimane in attesa che un **client** si connetta alla sua porta (come nell'esempio precedente la 6789), mediante la:

```
11 // rimane in attesa di un client
12 client = server.accept();
```

Al momento della connessione, facciamo effettuare un'echo del **socket** ora istanziato che ci permette di osservare come a ogni nuovo **client** che si connette viene mantenuta attiva la *local port* 6789 mentre troveremo volta per volta un valore diverso per la port dove viene "trasferito".

Quindi viene creato un thread passando il **socket** al costruttore e successivamente viene avviato.

```
44 public class MultiServer{
45     public void start(){
46         try{
47             ServerSocket serverSocket = new ServerSocket(6789);
48             for (;;)
49             {
50                 System.out.println("1 Server in attesa ... ");
51                 Socket socket = serverSocket.accept();
52                 System.out.println("3 Server socket " + socket);
53                 ServerThread serverThread = new ServerThread(socket);
54                 serverThread.start();
55             }
56         }
57         catch (Exception e){
58             System.out.println(e.getMessage());
59             System.out.println("Errore durante l'istanza del server !");
60             System.exit(1);
61         }
62     }
}
```

In questo esempio il **server** non termina mai la propria esecuzione: il corpo del metodo **start()** è costituito da un ciclo infinito d'attesa sulla **accept()** per la generazione di un numero indefinito di **client**.

Il **main** si riduce alla seguente istruzione:

```
63
64     public static void main (String[] args){
65         MultiServer tcpServer = new MultiServer();
66         tcpServer.start();
67     }
```

Facciamo alcune modifiche anche al **client** per realizzare il metodo che permette di inviare più messaggi al **server**: il metodo modificato è **comunica()**.

In questo metodo realizziamo un ciclo infinito mediante l'istruzione **for(;;)**, dove ogni iterazione di tale ciclo esegue:

- ▶ input di una riga da parte dell'utente;
- ▶ trasmissione al **server**;
- ▶ ricezione dell'echo da parte del **server**;

e termina quando l'utente inserisce la parola **FINE** chiudendo la connessione.

```

13 public void comunica() {
14     for (;;)                                // ciclo infinito: termina con FINE
15     try{
16         System.out.println("4 ... utente, inserisci la stringa da trasmettere al server:");
17         stringaUtente = tastiera.readLine();
18         //la spedisco al server
19         System.out.println("5 ... invio la stringa al server e attendo ...");
20         outVersoServer.writeBytes( stringaUtente+'\n');
21         //leggo la risposta dal server
22         stringaRicevutaDalServer=inDalServer.readLine();
23         System.out.println("7 ... risposta dal server "+'\n'+stringaRicevutaDalServer );
24         if (stringaUtente.equals("FINE")) {
25             System.out.println("8 CLIENT: termina elaborazione e chiude connessione" );
26             miosocket.close();                      // chiudo la connessione
27             break;
28         }
29     }

```

E in caso di errore viene catturata l'eccezione:

```

29 }
30 catch (Exception e)
31 {
32     System.out.println(e.getMessage());
33     System.out.println("Errore durante la comunicazione col server!");
34     System.exit(1);
35 }

```

Il metodo **connetti()**, non molto diverso da quello precedentemente descritto, nell'esercitazione di laboratorio 1, è il seguente:

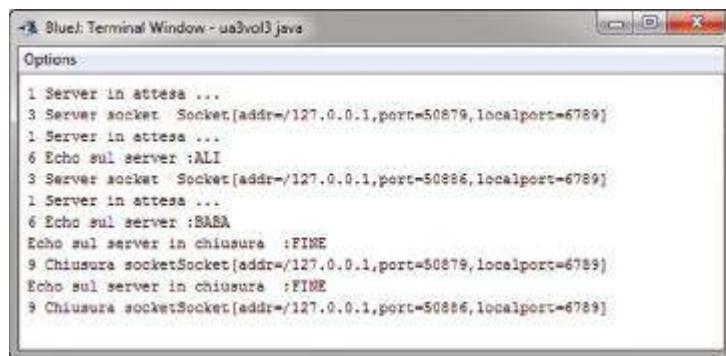
```

96 public Socket connetti(){
97     System.out.println("2 CLIENT partito in esecuzione ...");
98     try{
99         // input da tastiera
100        tastiera = new BufferedReader(new InputStreamReader(System.in));
101        // miosocket = new Socket(InetAddress.getLocalHost(), 6789);
102        miosocket = new Socket(nomeServer,portaServer);
103        // associo due oggetti ai socket per effettuare la scrittura e la lettura
104        outVersoServer = new DataOutputStream(miosocket.getOutputStream());
105        inDalServer    = new BufferedReader(new InputStreamReader (miosocket.getInputStream()));
106    }
107    catch (UnknownHostException e){
108        System.err.println("Host sconosciuto"); }
109    catch (Exception e){
110        System.out.println(e.getMessage());
111        System.out.println("Errore durante la connessione!");
112        System.exit(1);
113    }
114    return miosocket;
115 }

```

Il resto della classe (chiamata public class **Multiclient()**) è identico al **client** descritto nel laboratorio precedente.

Vediamo gli output generati da una esecuzione: mandiamo dapprima in esecuzione il **server** e successivamente due **client**.



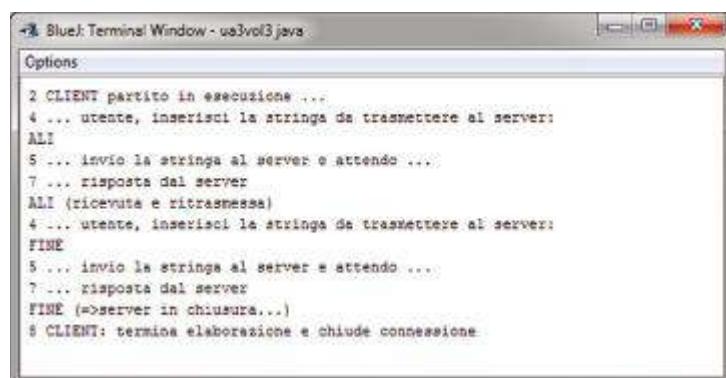
```

-> Blue: Terminal Window - ua3vol3.java
Options
1 Server in attesa ...
3 Server socket  Socket[addr=/127.0.0.1,port=50879,localport=6789]
1 Server in attesa ...
6 Echo sul server :ALI
3 Server socket  Socket[addr=/127.0.0.1,port=50886,localport=6789]
1 Server in attesa ...
6 Echo sul server :BABA
Echo sul server in chiusura :FINE
9 Chiusura socketSocket[addr=/127.0.0.1,port=50879,localport=6789]
Echo sul server in chiusura :FINE
9 Chiusura socketSocket[addr=/127.0.0.1,port=50886,localport=6789]

```

I **client** comunicano al **server** solamente una parola (rispettivamente **ALI** e **BABA**): nel primo (**ALI**) digitiamo **FINE** mentre apriamo una nuova finestra e mandiamo in esecuzione un terzo **client**.

Di seguito, è riprodotto l'echo() sul **client ALI**:

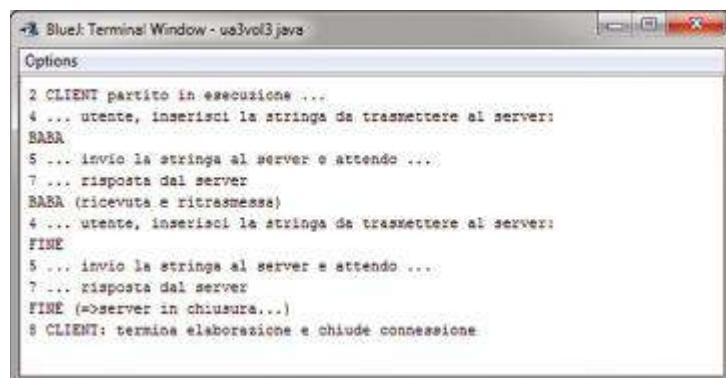


```

-> Blue: Terminal Window - ua3vol3.java
Options
2 CLIENT partito in esecuzione ...
4 ... utente, inserisci la stringa da trasmettere al server:
ALI
5 ... invio la stringa al server e attendo ...
7 ... risposta dal server
ALI (ricevuta e ritrasmessa)
4 ... utente, inserisci la stringa da trasmettere al server:
FINE
5 ... invio la stringa al server e attendo ...
7 ... risposta dal server
FINE (=>server in chiusura...)
8 CLIENT: termina elaborazione e chiude connessione

```

e l'echo() sul **client BABA**:



```

-> Blue: Terminal Window - ua3vol3.java
Options
2 CLIENT partito in esecuzione ...
4 ... utente, inserisci la stringa da trasmettere al server:
BABA
5 ... invio la stringa al server e attendo ...
7 ... risposta dal server
BABA (ricevuta e ritrasmessa)
4 ... utente, inserisci la stringa da trasmettere al server:
FINE
5 ... invio la stringa al server e attendo ...
7 ... risposta dal server
FINE (=>server in chiusura...)
8 CLIENT: termina elaborazione e chiude connessione

```



## Prova adesso!

Realizza un'applicazione che simuli una lotteria di 90 numeri; alcuni giocatori si collegano e "acquistano" alcuni numeri (per esempio 5) estratti causalmente dal **server**: quando si raggiunge un numero adeguato di giocatori (almeno 4) e non avvengono successive connessioni per almeno 60 secondi il **server** provvede alla estrazione di 5 premi, comunicando ai giocatori l'esito della estrazione; a loro volta i **client**, se hanno un numero vincente, lo segnalano al **server** e ritirano la vincita.

## Esercizi proposti

- 1 Realizza l'agenzia stampa PANZA: ogni redazione locale invia a un server una notizia catalogata secondo lo schema Settore/Argomento/Area geografica; l'agenzia server le deve smistare in base a tale classificazione e inviare come risposta tutte le notizie analoghe.
- 2 Realizza un sistema in cui il server sia il banditore di un'asta: accoglie le offerte e comunica se sono accettabili o meno, comunica a richiesta la migliore offerta corrente; i client sono i partecipanti all'asta che possono richiedere quale sia l'offerta migliore (e di chi) e possono effettuare rilanci se il loro budget a disposizione lo consente.
- 3 Realizza un sistema che simuli la Borsa Valori: il server comunica ciclicamente un listino azionario con la quotazione dei singoli titoli a tutti i client a lui connessi, i quali possono comunicare una variazione (verso l'alto o verso il basso di una certa quantità fissa) del valore di un titolo del listino.
- 4 Realizza un sistema per la gestione del gioco della tombola dove il server gestisce il tabellone e assegna a ogni singolo client che si connette una cartella casuale; ogni client controlla i numeri estratti e segnala al server le eventuali vincite.
- 5 Realizza la simulazione di una corsa dei cavalli: quattro cavalli percorrono uno o più giri di un ippodromo ovale fino al termine della gara. Aggiungi successivamente la possibilità di effettuare puntate sui cavalli.
- 6 Realizza un sistema che gestisca il movimento di automobili in rete: il server attende la connessione di N giocatori che muovono la propria auto mediante i tasti freccia.
- 7 Realizza un sistema di estrazioni del lotto: N giocatori scelgano tre numeri e li comunichino alla ricevitoria. Quindi effettua le estrazioni del lotto comunicando eventuali vincite.
- 8 I contatori dell'energia elettrica sono connessi in rete in modo che in giorni ben determinati vengano interrogati per comunicare i consumi che sono stati effettuati e sono raggruppati per aree geografiche: ogni contatore ha un numero di matricola che viene comunicato assieme al valore della lettura.  
Scrivi una applicazione che invii un codice di area e si ponga in attesa delle risposte dei diversi contatori; poi memorizzi i dati ricevuti in un file di record con la seguente struttura:  
`dataLettura, matricolaContatore, .valoreLettura.`  
Al file viene dato come nome il codice dell'area.

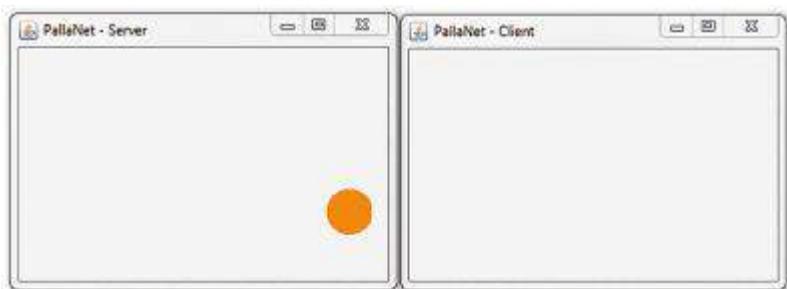
# ESERCITAZIONI DI LABORATORIO 4

## JAVA SOCKET: UN'ANIMAZIONE CLIENT-SERVER

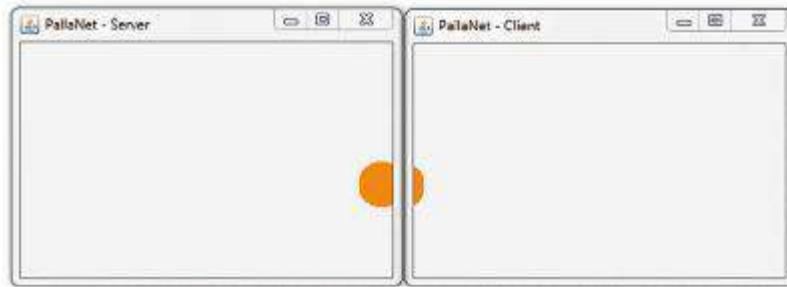
### ■ Una palla che “attraversa” le finestre

Realizziamo ora un'applicazione costituita da due finestre indipendenti affiancate e una pallina in movimento che si sposta nella prima finestra: quando raggiunge il bordo destro, anziché rimbalzare, “esce” dalla prima finestra ed “entra” nella seconda (come mostrato nella sequenza di immagini).

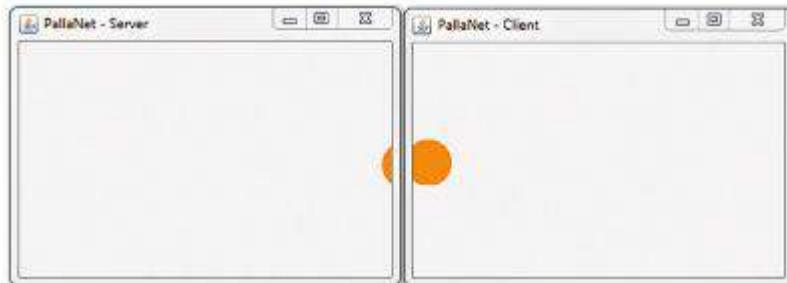
Si avvicina al bordo destro  
della prima finestra: ►



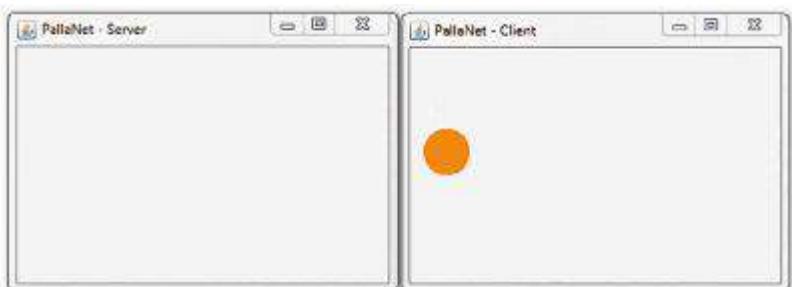
inizia l'attraversamento ►



procede nella seconda finestra ►



e termina l'attraversamento passando completamente nella seconda finestra dove continua il suo movimento, rimbalza sui suoi bordi e ritorna nella prima finestra. ►



## Descrizione della soluzione

Realizziamo il sistema con una applicazione **client-server**, quindi definiamo due classi e facciamole comunicare tra loro.

La pallina viene generata dal **server** non appena si accorge che il **client** si è connesso.

Dovendo gestire un'animazione, abbiamo problemi di sfarfallio che annulliamo con la tecnica del **doppio buffering**: realizziamo l'animazione gestendo l'evento generato dall'oggetto **Timer** e, dato che è proprio il metodo che gestisce tale evento a incaricarsi del movimento della pallina, e quindi riconoscere il momento in cui "sta per lasciare" la finestra, demandiamo a tale metodo la realizzazione della comunicazione tra **client** e **server** per ricevere la segnalazione.

Implementiamo due classi:

```
class PannelloAnimazione extends JPanel implements ActionListener{...}

class PannelloClient extends JPanel implements ActionListener{ ...}
```

rispettivamente per il **server** e per il **client** che si occupano, quindi, di:

- animare la pallina;
- eliminare lo sfarfallio;
- comunicare tra loro le situazioni di transazione.

Nel complesso le due classi sono abbastanza simili: descriviamo solamente la classe **server** dove evidenzieremo le differenze con la classe del **client**.

La classe "starter" è la seguente, cioè **PallaNetServer**, che definisce la finestra e avvia un thread che attende la connessione del **client**:

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import java.io.*;
4 import java.net.*;
5 import javax.swing.*;
6 public class PallaNetServer extends JFrame{
7     private Socket connessione = null;
8     private DataOutputStream out = null;
9     private DataInputStream input = null;
10    public PallaNetServer() {
11        super("PallaNet - Server");
12        this.setSize(500,400);           //setto la grandezza della finestra
13        this.setLocationRelativeTo(null); //setto la posizione al centro dello schermo
14        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
15        //instanzio un oggetto per attendere la connessione di un client
16        ThreadConnessione attendiConnessione = new ThreadConnessione(this);
17        this.setVisible(true);
18    } //fine costruttore
```

Unica osservazione nella definizione di classe è che estende la classe **JFrame**: nulla da sottolineare nel costruttore tranne che per l'utilizzo di un oggetto della classe **ThreadConnessione()** che rimane in attesa sulla porta **6789** mediante il metodo **setConnessione()**, dopo aver ricevuto dal costruttore il riferimento alla finestra.

Nel caso di singolo **client** non è necessario utilizzare i **thread**, ma questo **server** è stato così impostato a titolo di esempio, in modo da poter essere utilizzato intervenendo con "piccole modifiche" anche nel caso di connessione tra più **client**, come viene richiesto nell'esercizio proposto al termine della spiegazione di questo esempio.

Con l'istruzione 73 l'avvenuta connessione col **client** richama il metodo **setConnessione()** che, utilizzando il riferimento alla connessione con il **client**, crea gli stream di input e output e definisce e instanzia un **PannelloAnimazione()** avviando, di fatto, il movimento della pallina.

```

61 class ThreadConnessione implements Runnable{
62     private PallaNetServer finestra;
63     private Thread me;
64     public ThreadConnessione(PallaNetServer finestra){
65         //ottengo il riferimento alla JFrame
66         this.finestra = finestra;
67         me = new Thread(this);           //instanzio il Thread
68         me.start();                   //attivo il Thread
69     }
70     public void run(){
71         try{                         //instanzio un oggetto in ascolto sulla porta 6789
72             ServerSocket server = new ServerSocket(6789);
73             finestra.setConnessione(server.accept()); //attesa connessione
74             server.close();           //chiude il server
75         }catch(Exception e){
76             JOptionPane.showMessageDialog(null,e.toString());
77             System.exit(-1);
78         }
79     }//fine metodo run del Thread
80 }
```

Non appena il **client** si connette viene "chiuso" il **server** oppure si segnala un errore: naturalmente nel caso si dovessero attendere più **client** è necessario intervenire in questa prima parte di codice.

```

49
50     public void setConnessione(Socket connessione){
51         //ricevo il riferimento per la connessione con il client
52         this.connessione = connessione;
53         //ricevo lo stream di output e di input
54         try{
55             out = new DataOutputStream(connessione.getOutputStream());
56             input = new DataInputStream(connessione.getInputStream());
57         }catch(Exception e){
58             JOptionPane.showMessageDialog(null,e.toString());
59             System.exit(-1);
60         }
61         //inizio l'animazione su un oggetto di classe PannelloAnimazione
62         PannelloAnimazione pan = new PannelloAnimazione(this,this.getSize());
63         //lo aggiungo alla JFrame
64         this.add(pan);
65     }
```

Completano la classe due metodi definiti per rispettare l'incapsulamento. ►

```

45     public DataInputStream getInput(){
46         return input;
47     }
48
49     public DataOutputStream getOutput(){
50         return out;
51     }
52 } /* --Fine classe PallaNetServer -- */

```

Le differenze con la classe del **client**, la **PallaNetClient**, si limitano alle istruzioni di connessione. Al posto del thread che effettua la connessione con le seguenti istruzioni:

```

57     ServerSocket server = new ServerSocket(6789);
58     finestra.setConnessione(server.accept());           //attesa connessione
59     server.close();                                    //chiudo il server

```

nel **client** mettiamo semplicemente la seguente istruzione che crea la connessione:

```

61     //mi connetto al server sul socket<chomecomputer>,<porta>
62     connessione = new Socket("localhost",6789);

```

La definizione della classe e del suo costruttore è la seguente:

```

1 import java.awt.*;
2 import java.awt.event.*;
3 import java.io.*;
4 import java.net.*;
5 import javax.swing.*;
6 public class PallaNetClient extends JFrame{
7     private Socket connessione = null;
8     private DataOutputStream out = null;
9     private DataInputStream input = null;
10    public PallaNetClient(){
11        super("PallaNet - Client");                      // costruttore della super classe
12        this.setSize(500,400);                           // grandezza della finestra
13        this.setLocationRelativeTo(null);                // posizione al centro schermo
14        this.setDefaultCloseOperation(EXIT_ON_CLOSE); //modalità di chiusura
15        connetti();                                     //mi connetto al server
16        this.setVisible(true);                          //rendo visibile la finestra
17    }//fine costruttore

```

Il metodo che effettua la connessione e avvia l'animazione della pallina è il seguente:

```

18
19     public void connetti(){
20         try {
21             //mi connetto al server sul socket<chomecomputer>,<porta>
22             connessione = new Socket("localhost",6789);
23             //ottengo lo stream di input dal server e di output verso il server
24             out = new DataOutputStream(connessione.getOutputStream());
25             input = new DataInputStream(connessione.getInputStream());
26         }catch(Exception e){
27             JOptionPane.showMessageDialog(null,e.toString());
28             System.exit(-1);
29         }
30         //inizio l'animazione
31         Container contenitore = this.getContentPane();
32         PannelloClient pan = new PannelloClient(this,contenitore.getSize());
33         contenitore.add(pan);   //lo aggiungo alla JFrame
34     }

```

Conclude la classe la definizione dei due metodi getter:

```

18 public DataInputStream getInput(){
19     return input;
20 }
21 public DataOutputStream getOutput(){
22     return out;
23 }

```

I pannelli che gestiscono l'animazione sono simili: descriviamo quindi solo quello del **server**, cioè la classe **PannelloAnimazione**.

Iniziamo definendo la classe con le variabili necessarie per l'animazione:

```

67 class PannelloAnimazione extends JPanel implements ActionListener{
68     private PallaNetServer finestra;
69     private Dimension dimensioni;
70     private Image bufferVirtuale;           // per il doppio buffering
71     private Graphics offScreen;
72     private Timer tim = null;
73     private int xPallina = 0;               // coordinate iniziali
74     private int yPallina = 0;
75     private int diametroPallina = 40;
76     private int spostamento = 3;
77     private int timerDelay = 10;
78     private boolean destra,basso,pallinaPresente,arrivoComunicato;
79

```

Gli attributi sono “parlanti” e non necessitano di spiegazioni aggiuntive.

Anche il suo costruttore non presenta particolarità degne di nota:

```

93     public PannelloAnimazione(PallaNetServer finestra,Dimension dimensioni){
94         super();
95         this.finestra = finestra;          // riferimento alla finestra corrente
96         this.setSize(dimensioni);        // setto la grandezza
97         destra = true;                  // direzioni iniziali della pallina
98         basso = true;
99         pallinaPresente = true;         //la pallina parte dal questo schermo
100        arrivoComunicato = false;
101        tim = new Timer(timerDelay,this); // instanzio il Timer per il movimento
102        tim.start();                   // attivo il Timer
103    }

```

Completano la classe i due metodi che si occupano del disegno evitando lo sfarfallio mediante la tecnica del **doppio buffering**:

```

102     public void update(Graphics g){      // aggiorno il pannello
103         paint(g);                      // eseguo il disegno in Doppio Buffering
104     }
105     public void paint(Graphics g){
106         super.paint(g);
107         //definisco lo spazio esterno per il doppioBuffering
108         bufferVirtuale = createImage(getWidth(),getHeight());
109         offScreen = bufferVirtuale.getGraphics();
110         Graphics2D screen = (Graphics2D) g;
111         offScreen.setColor(new Color(254,138,22)); // setto il colore della palla
112         if(pallinaPresente){                // disegno la palla
113             offScreen.fillOval(xPallina,yPallina,diametroPallina,diametroPallina);
114         }
115         //disegno l'immagine modificata "bufferVirtuale" sul Component
116         screen.drawImage(bufferVirtuale,0,0,this);
117         offScreen.dispose();
118     }

```

Più articolato è il metodo che gestisce l'evento generato dal **time** e che quindi si occupa del movimento della pallina ed eventualmente della “migrazione” tra le due finestre. Lo analizziamo in tre segmenti:

- aggiornamento dello spostamento della palla in verticale;
- aggiornamento dello spostamento della palla in orizzontale;
- palla assente nella finestra.

Nulla da segnalare quando aggiorniamo il moto verticale in quanto dobbiamo solo controllare quando avviene la collisione con i bordi per invertire la direzione:

```

L13 public void actionPerformed(ActionEvent e) {
L14     if(pallinaPresente){
L15         /*direzione in verticale*/
L16         if(basso){
L17             //si muove verso il basso
L18             if(yPallina > (this.getHeight()-diametroPallina)){// urto inferiore
L19                 basso = false;                                //cambia direzione
L20                 yPallina -= spostamento;
L21             }else{
L22                 yPallina += spostamento;
L23             }
L24         }else{
L25             //si muove verso l'alto
L26             if(yPallina <= 0){                            // urto superiore
L27                 basso = true;                            //cambia direzione
L28                 yPallina += spostamento;
L29             }else{
L30                 yPallina -= spostamento;
L31             }
L32         }
L33     }
L34 }
```

Durante la direzione orizzontale abbiamo due diverse situazioni:

- Ⓐ nel caso di *movimento verso destra*, quando la pallina tocca il bordo destro, inizia la “migrazione” nella finestra del **client**; dobbiamo quindi:
- comunicare al **client** che inizia il trasferimento;
  - comunicare al **client** la coordinata verticale di “passaggio”;
  - comunicare al **client** la direzione verticale.

Quindi impostiamo una variabile booleana (**arrivoComunicato**) utilizzata dal **server** per non ripetere la comunicazione al **client** delle successive posizioni della pallina durante la transazione:

```

L43 /*direzione orizzontale*/
L44 if(destra){
L45     if(!arrivoComunicato)&&(xPallina > (this.getWidth()-diametroPallina)){
L46         /* comunica il verso di direz. verticale e la coordinata verticale
L47         * in cui la pallina si trova attualmente. */
L48         try{
L49             finestra.getOutputStream().writeBoolean(basso);
L50             finestra.getOutputStream().writeInt(yPallina);
L51             arrivoComunicato = true;
L52         }catch(Exception ecc){
L53             JOptionPane.showMessageDialog(null,ecc.toString());
L54             System.exit(-1);
L55         }
L56     }else{
L57         xPallina += spostamento;
L58         if(xPallina > this.getWidth()){
L59             pallinaPresente = false;
L60             arrivoComunicato = false;
L61         }
L62     }
L63 }
```

Durante la transazione avremo due palline presenti, una nel **client** che va “scomparendo” e una nel **server** che si sta “formando”.

Quando la pallina ha lasciato completamente la finestra del **server** vengono modificati i valori delle due variabili booleane **pallinaPresente** e **arrivoComunicato** che vengono posti al valore di falso.

- B Nel caso di *movimento verso sinistra* è sufficiente controllare se si è raggiunto il bordo laterale dello schermo per invertire la direzione:

```

163     }else{
164         if(xPallina <= 0){           // la pallina sta andando a sinistra
165             destra = true;          // fine finestra a sinistra
166             xPallina += spostamento;
167         }else{
168             xPallina -= spostamento;
169         }
170     }

```

Quando la pallina non è presente, il **server** rimane in attesa con l’istruzione 177 di una eventuale comunicazione da parte del **client** dell’inizio di “ritorno pallina”: in questa occasione vengono aggiornate le coordinate e la variabile di stato **pallinaPresente=true**.

```

171     }else{                      // la pallina non è presente
172         // rimango in attesa del ritorno della pallina nel mio schermo
173         boolean direzione = false;
174         int y = 0;
175         try{
176             direzione = finestra.getInput().readBoolean();
177             y = finestra.getInput().readInt(); // attendo che arriva la pallina
178             basso = direzione;            // direzione di ingresso
179             destra = false;              // si muove verso sinistra
180             yPallina = y;                // coord. iniziali della pallina
181             xPallina = this.getWidth();
182             pallinaPresente = true;
183         }
184         catch(Exception ecc){
185             JOptionPane.showMessageDialog(null,ecc.toString());
186             System.exit(-1);
187         }
188     }

```

L’ultima operazione del metodo effettua il ridisegno invocando il metodo **repaint()**.

```

189     }
190     repaint();
191 }

```

Il codice del **client** è simile: sono naturalmente invertite le situazioni di movimento della pallina, cioè se dal **server** esce dal lato destro, nel **client** entrerà dal lato sinistro e da questo lato successivamente uscirà per rientrare nel **server** dallo stesso lato da cui è uscita.

Anche le classi di prova sono simili, per cui riportiamo solo quella del **server** a titolo di esempio. ►

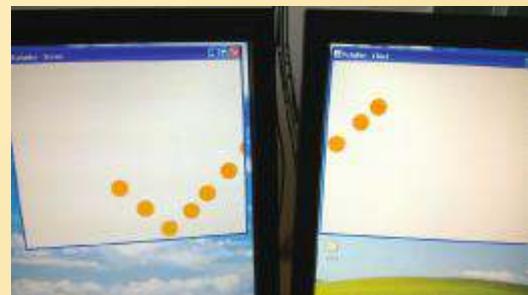
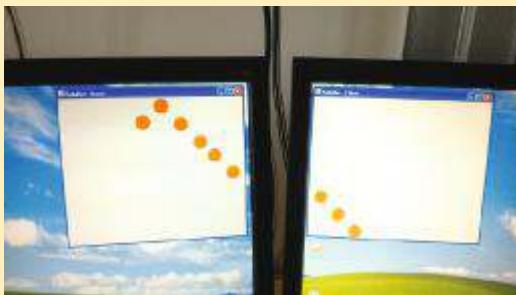
```

1 public class ProvaPallaServer{
2     public static void main(String args[]){
3         PallaNetServer server = new PallaNetServer();
4     }
5 }

```

Maggior effetto scenografico si ottiene mandando in esecuzione i due thread su due macchine diverse, connesse in rete, ponendo i due monitor vicini: unica modifica da apportare alla classe **client** è l'aggiornamento del nome del PC su cui viene mandato in esecuzione il thread **server**.

La fotografia seguente ne riporta una esecuzione



```
class PannelloAnimazione extends JPanel implements ActionListener{...}  
class PannelloClient extends JPanel implements ActionListener{ ...}
```



### Prova adesso!

Modifica il programma precedente aggiungendo un secondo **client**, cioè una terza finestra, facendo in modo che la pallina esca prima a destra e successivamente a sinistra dalla videata centrale.

# ESERCITAZIONI DI LABORATORIO 5

## IL PROTOCOLLO UDP NEL LINGUAGGIO JAVA

### ■ Client e server UDP in Java

Il protocollo **UDP** ha caratteristiche diverse dal protocollo **TCP**: **UDP** non è orientato alla connessione e quindi tra due host non si crea uno stream stabile.

L'avvio di una comunicazione avviene senza handshaking e la consegna non è garantita; al contrario i controlli sull'integrità vengono eseguiti come in **TCP**.

Uno dei principali vantaggi di **UDP** è che, grazie alla sua semplicità offre un servizio molto rapido e non richiede **handshaking** (per questo viene spesso utilizzato per **DNS**).

In questa esercitazione realizzeremo la comunicazione mediante protocollo **UDP** dove un mittente (**UDPClient.java**) trasmette un messaggio (datagram) a un destinatario (**UDPServer.java**).

Ricordiamo che in una comunicazione dati “connectionless o datagram” il canale:

- ▷ trasporta messaggi;
- ▷ non è affidabile;
- ▷ ha il **socket** condiviso;
- ▷ non preserva l'ordine delle informazioni.

Inoltre l'ordine di arrivo dei pacchetti non è necessariamente lo stesso di invio.

Realizzare un sistema **client-server UDP** consiste nell'implementare un programma che effettui i seguenti passi fondamentali.

Per il **client**:

- ▷ crea il **socket**;
- ▷ manda richiesta sul **socket** (composto da *indirizzo*, *porta* e *messaggio*);
- ▷ riceve dati dal **socket**;
- ▷ chiude il **socket**.

Per il **server**:

- ▷ crea il **socket**
- ▷ ripete le seguenti operazioni:
  - si mette in attesa delle richieste in arrivo;
  - riceve il *messaggio* dal **socket**;
  - invia la *risposta* sul **socket** al **client** che ha fatto una richiesta;
- ▷ chiude il **socket**.

La comunicazione **UDP** viene realizzata in **Java** mediante la classe **DatagramSocket**.

## Classe DatagramSocket

La classe Java `DatagramSocket` usa il protocollo UDP per trasmettere dati attraverso i `socket`: permette a un `client` di connettersi a una determinata porta di un host per leggere e scrivere dati impacchettati attraverso la classe `DatagramPacket`.

L'indirizzo dell'host destinatario è sul `DatagramPacket`.

I principali metodi della `DatagramSocket(..)` sono i seguenti.

- Metodi costruttori:

```
void DatagramSocket() throws SocketException
void DatagramSocket(int porta) throws SocketException
```

- Metodi per ricevere/trasmettere:

```
void send(DatagramPacket pacchettoDati) throws IOException
void receive(DatagramPacket pacchettoDati) throws IOException
```

Il metodo `receive()` blocca il chiamante fino a quando un pacchetto è ricevuto

```
void setSoTimeout(int timeout) throws SocketException
```

usando `setSoTimeout()` il chiamante di `receive()` si blocca dopo timeout millisecondi.

Infine, il “solito” metodo per chiudere la connessione:

```
void close()
```

## Classe DatagramPacket

La classe `DatagramPacket` è la classe Java che permette di rappresentare i pacchetti UDP da inviare e ricevere sui `socket` di tipo datagram.

Il costruttore della classe è il seguente:

```
public DatagramPacket(byte buffer[], int lunghezza, InetAddress indirizzo, int porta)
```

Il `client` crea un oggetto di `DatagramPacket` passando al costruttore:

- il contenuto del messaggio: un array di `lunghezza` caratteri;
- l'`indirizzo IP` del `server`;
- il numero di `porta` su cui il `server` è in ascolto.

Il `server`, per ricevere un messaggio, crea un oggetto di `DatagramPacket` definendone la lunghezza massima:

```
public DatagramPacket(byte buffer[], int lunghezza)
```

I principali metodi della classe `DatagramPacket` sono:

- Ⓐ Per la gestione dell'indirizzo IP:

```
void setAddress(InetAddress indirizzo): setta il valore dell'indirizzo IP;
InetAddress getAddress(): restituisce l'indirizzo IP del host da cui il pacchetto è stato ricevuto.
```

**B** Per la gestione della porta:

```
void setPort(int porta): setta il valore della porta;
int getPort(): restituisce la porta della macchina remota da cui il pacchetto è stato ricevuto.
```

**C** Per la gestione dei dati:

```
void setData(byte[] buffer) : inserisce i dati nel pacchetto;
byte[] getData() : restituisce i dati del pacchetto.
```

## Server UDP

Scriviamo il **server UDP**: creiamo un **DatagramSocket()** sulla solita porta **6789** e definiamo i due buffer per i dati, rispettivamente per la ricezione e per la trasmissione, per esempio della medesima dimensione di 1024 byte.

Una variabile booleana viene definita per essere utilizzata come condizione di ripetizione/uscita dal ciclo.

```
1 import java.io.*;
2 import java.net.*;
3
4 class UDPServer{
5     public static void main(String args[]) throws Exception {
6         DatagramSocket serverSocket = new DatagramSocket(6789);
7         boolean attivo = true; // per la ripetizione del servizio
8         byte[] bufferIN = new byte[1024]; // buffer spedizione e ricezione
9         byte[] bufferOUT = new byte[1024];
10
11         System.out.println("SERVER avviato...");
12         while(attivo)
```

Definiamo ora il datagramma e ci poniamo in attesa di ricevere dati da qualche **client**:

```
14     // definizione del datagramma
15     DatagramPacket receivePacket =
16         new DatagramPacket(bufferIN,bufferIN.length);
17     //attesa della ricezione dato dal client
18     serverSocket.receive(receivePacket);
19
```

Quando viene ricevuto un pacchetto, viene analizzato estraendo il messaggio, individuando la lunghezza impostata alla trasmissione per eliminare i caratteri superflui presenti nel buffer, che ricordiamo essere di 1024 byte;

```
20     // analisi del pacchetto ricevuto
21     String ricevuto = new String(receivePacket.getData());
22     int numCaratteri = receivePacket.getLength();
23     ricevuto=ricevuto.substring(0,numCaratteri); //elimina i caratteri in eccesso
24     System.out.println("RICEVUTO: " + ricevuto);
```

Sempre dal pacchetto ricevuto vengono individuati i parametri per la trasmissione della risposta, cioè l'indirizzo e il numero di porta del **client**:

```
25     // riconoscimento dei parametri del socket del client
26     InetAddress IPClient = receivePacket.getAddress();
27     int portaClient = receivePacket.getPort();
28
```

Viene preparata la risposta, che in questo caso non è altro che il messaggio ricevuto trasformato in maiuscolo, viene creato il datagramma in uscita e viene inviato al **client**:

```

17
18 // preparo il dato da spedire
19 String daSpedire = ricevuto.toUpperCase();
20 bufferOUT = daSpedire.getBytes();
21 // invio del Datagramma
22 DatagramPacket sendPacket =
23     new DatagramPacket(bufferOUT, bufferOUT.length, IPClient, portaClient);
24 serverSocket.send(sendPacket);

```

Se il **client** ha inviato la parola “fine”, viene conclusa l’elaborazione del **server**, si esce quindi dal ciclo e si chiude il **socket**.

```

38 // controllo termine esecuzione del server
39 if (ricevuto.equals("fine"))
40 {
41     System.out.println("SERVER IN CHIUSURA. Buona serata.");
42     attivo=false;
43 }
44 }
45 serverSocket.close();
46 }

```

## Client UDP

Scriviamo ora il **client UDP**: come per il **server** definiamo i buffer per la comunicazione e indichiamo i riferimenti del **socket** del **server**:

```

1 import java.io.*;
2 import java.net.*;
3
4 class UDPClient
5 {
6     public static void main(String args[]) throws Exception
7     {
8         int portaServer = 6789;           // porta del server
9         InetAddress IPServer = InetAddress.getByName("localhost");
10
11         byte[] bufferOUT = new byte[1024]; // buffer di spedizione e ricezione
12         byte[] bufferIN = new byte[1024];
13         BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

```

Creiamo un **socket** e invitiamo l’utente a inserire un dato:

```

16 // creazione del socket
17 DatagramSocket clientSocket = new DatagramSocket();
18 System.out.println("Client pronto - inserisci un dato da inviare:");

```

Leggiamo la stringa inserita dall’utente e predisponiamo il buffer di uscita:

```

19 // preparazione del messaggio da spedire
20 String daSpedire = input.readLine();
21 bufferOUT = daSpedire.getBytes();

```

Creiamo ora un **DatagramPacket** con il messaggio da inviare e i parametri del **server** e lo inviamo a destinazione:

```

23 // trasmissione del dato al server
24 DatagramPacket sendPacket =
25     new DatagramPacket(bufferOUT, bufferOUT.length, IPserver, portaServer);
26 clientSocket.send(sendPacket);

```

Quindi ci mettiamo in attesa della risposta, predisponendo il **DatagramPacket** con il buffer per la ricezione:

```

27 // ricezione del dato dal server
28 DatagramPacket receivePacket = new DatagramPacket(bufferIN, bufferIN.length);
29 clientSocket.receive(receivePacket);
30 String ricevuto = new String(receivePacket.getData());

```

Alla ricezione del messaggio da parte del **server**, questo viene analizzato, vengono tolti i caratteri eccedenti presenti nel buffer di ricezione e viene visualizzato il risultato:

```

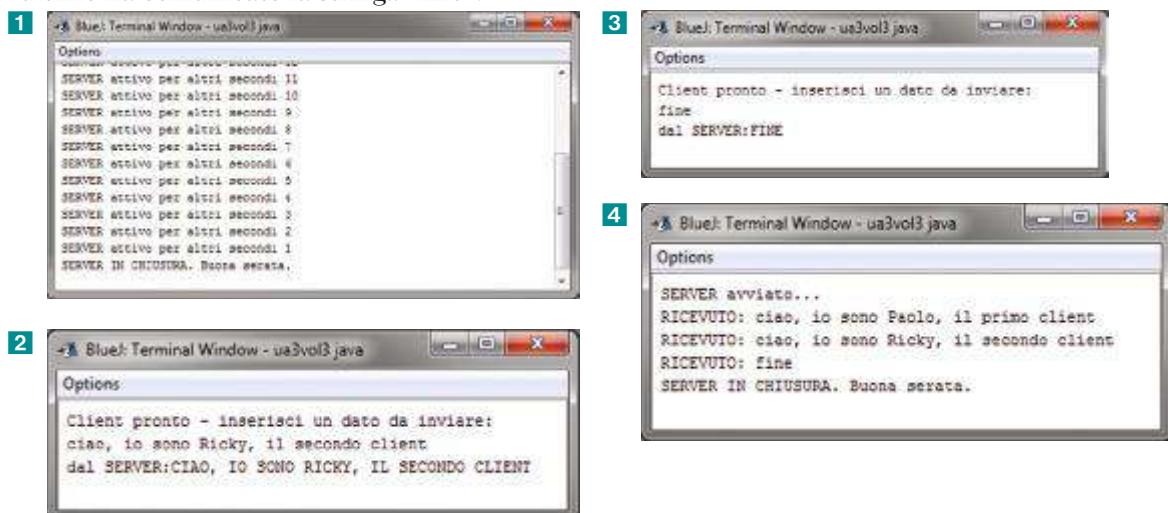
// elaborazione dei dati ricevuti
int numCaratteri = receivePacket.getLength();
ricevuto=ricevuto.substring(0,numCaratteri); //elimina i caratteri in eccesso
System.out.println("dal SERVER:" + ricevuto);

// termine elaborazione
clientSocket.close();

```

Il **client** termina l'elaborazione chiudendo la connessione.

Un'esecuzione della applicazione è la seguente, dove sono stati mandati in esecuzione tre **client** e l'ultimo ha comunicato la stringa "fine":



## Prova adesso!

Il servizio daytime (udp/13) fornisce data e ora correnti in formato comprensibile da un essere umano: scrivi un **server** che invia data e ora in un pacchetto **UDP** a ogni **client** che gli invia un qualsiasi pacchetto **UDP**, anche vuoto.

Successivamente effettua il conteggio delle volte che un medesimo **client** si è connesso memorizzandolo su di un file: alla decima connessione segnala al **client** che ha "esaurito i bonus" gratuiti e da questo momento il servizio "orario" è a pagamento.

# ESERCITAZIONI DI LABORATORIO 6

## APPLICAZIONI MULTICAST IN JAVA

### ■ Client e server multicast

La comunicazione tra gruppi di processi realizzata mediante **multicasting** (one to many communication):

- ▶ un insieme di processi formano un **gruppo di multicast**;
- ▶ un messaggio **spedito** da un **processo** a quel gruppo viene recapitato a tutti gli **altri** partecipanti appartenenti al gruppo.

L'implementazione del **multicast** richiede:

- ▶ uno schema di indirizzamento dei gruppi;
- ▶ un supporto che registri la corrispondenza tra un gruppo e i partecipanti;
- ▶ un'implementazione che ottimizzi l'uso della rete nel caso di invio di pacchetti a un gruppo di multicast.

Le **API Multicast** contengono le primitive per far unire un **client** a un gruppo di **multicast (join)** e per lasciare il gruppo (**leave**) quando non è più interessato a ricevere i messaggi relativi a quel gruppo.

La libreria **java.net** mette a disposizione la classe **MulticastSocket** per inviare messaggi **multicast** e con i **socket datagram** si può usare il **multicast** per mandare un pacchetto a un insieme di processi con una sola invocazione a **send()**.

I principali metodi della classe **MulticastSocket** sono:

<b>joinGroup(InetAddress addr) throws IOException</b>	: per unirsi al gruppo
<b>leaveGroup(InetAddress addr) throws IOException</b>	: per lasciare il gruppo
<b>send(DatagramPacket p)</b>	: per inviare un pacchetto
<b>setTimeToLive(int tlive) throws IOException</b>	: per settare il tempo di vita dei pacchetti inviati sul <b>socket</b>
<b>int getTimeToLive() throws IOException</b>	: ritorna il tempo restante (tra 0 e 255)

### Client multicast

I **client** che vogliono ricevere messaggi **multicast** devono unirsi a un gruppo specifico e per far questo devono conoscere:

- ▶ la porta del **socket** usata dal **server** (per noi la solita **6789**);
- ▶ l'indirizzo del gruppo a cui vengono inviati messaggi (per esempio "**225.4.5.6**").

Vediamo nello specifico come realizzare una classe **client** che si connette a un gruppo e riceve i messaggi a esso destinati:

```

1 import java.net.*;
2 import java.io.*;
3
4 public class MulticastClient
5 {
6     public static void main(String[] args) throws IOException
7     {
8         byte[] bufferIN = new byte[1024];           // buffer di ricezione
9         //parametri del server
10        int porta = 6789;
11        String gruppo = "225.4.5.6";

```

Creiamo un **socket multicast** e uniamo il **client** nello specifico gruppo definito sul **server**:

```

12 // creazione del socket sulla porta
13 MulticastSocket socket = new MulticastSocket(porta);
14 // mi aggiungo al gruppo Multicast
15 socket.joinGroup(InetAddress.getByName(gruppo));

```

Creiamo un oggetto datagramma e ci poniamo in attesa dei messaggi del **server**:

```

16 // creo il DatagramPacket e mi metto in ricezione
17 DatagramPacket packetIN = new DatagramPacket(bufferIN, bufferIN.length);
18 socket.receive(packetIN);

```

Sullo schermo visualizziamo il messaggio ricevuto e alcuni parametri della porta del **socket**:

```

21 // Visualizzo i parametri ed i dati ricevuti
22 System.out.println("Ho ricevuto dati di lunghezza: " + packetIN.getLength()
23   + " da : " + packetIN.getAddress().toString()
24   + " porta :" + packetIN.getPort());
25 System.out.write(packetIN.getData(),0,packetIN.getLength());
26 System.out.println();

```

Al termine delle operazioni il **client** si scollega dal gruppo e chiude il **socket**.

```

29 // al termine della ricezione lascio il gruppo
30 socket.leaveGroup(InetAddress.getByName(gruppo));
31 // chiudo il socket
32 socket.close();

```

Naturalmente un **client** può continuare a ricevere quanti pacchetti vuole: è sufficiente includere la **receive()** in un ciclo ed eseguirla periodicamente.

## Server multicast

Il processo **server** crea il **socket** per una porta e invia, quando vuole, pacchetti **datagram** al gruppo: nel nostro esempio invieremo la data e l'ora a intervalli di un secondo per un determinato tempo prefissato nella variabile conta (per esempio per 20 secondi).

Definiamo “225.4.5.6” come indirizzo di gruppo e apriamo un **socket** sulla porta 6789:

```

1 import java.net.*;
2 import java.io.*;
3 import java.util.*;
4
5 public class MulticastServer
6 {
7     public static void main(String[] args) throws IOException
8     {
9         boolean attivo = true;           // per la ripetizione del servizio
10        byte[] bufferOUT = new byte[1024]; // buffer di spedizione e ricezione
11        int conta = 20;                // secondi di attività del server
12        int porta = 6789;
13        InetSocketAddress gruppo = new InetSocketAddress("225.4.5.6");

```

Creiamo il **socket**

```

15 // creo il socket multicast
16 MulticastSocket socket = new MulticastSocket();
17

```

e prepariamo la variabile che conterrà il dato da trasmettere.

```

18 // contenitore per il dato da trasmettere
19 String dString = null;
20

```

Il ciclo di trasmissione genera il dato, crea un datagramma e lo spedisce al gruppo:

```

21 // ciclo di trasmissione
22 while (attivo)
23 {
24     // Come messaggio viene inviata la data e l'ora di sistema
25     dString = new Date().toString();
26     bufferOUT = dString.getBytes();
27     // creo il DatagramPacket
28     DatagramPacket packet;
29     packet = new DatagramPacket(bufferOUT, bufferOUT.length, gruppo, porta);
30     // invio il dato
31     socket.send(packet);

```

Il ciclo di trasmissione viene ripetuto ogni secondo grazie all'attesa del metodo **sleep()** e a ogni iterazione aggiorna il contatore che modifica la condizione di terminazione:

```

32 // introduco un ciclo di attesa di 1 secondo.
33 try {
34     Thread.sleep(1000);    //attesa di 1000 millisecondi
35 } catch (InterruptedException ie) {
36     ie.printStackTrace();
37 }
38 conta--;
39 if (conta==0){
40     System.out.println("SERVER IN CHIUSURA. Buona serata.");
41     attivo=false;
42 }else{
43     System.out.println("SERVER attivo per altri secondi "+conta);
44 }

```

Alla fine, come ultima istruzione, il **server** chiude la connessione.

```

46 |     // alla fine chiudo il socket
47 |     socket.close();
48 |

```

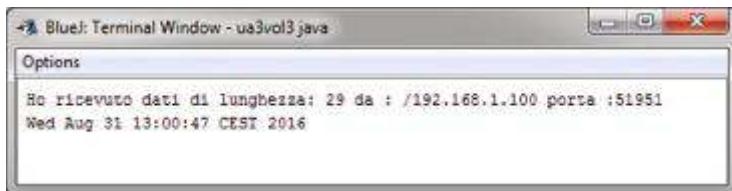
Una possibile esecuzione è la seguente:

**A** per il **server**:



The terminal window title is "Blue!: Blue!: Terminale - ua3vol3.java". The content is a list of messages from the server: SERVER attivo per altri secondi 19, SERVER attivo per altri secondi 18, SERVER attivo per altri secondi 17, SERVER attivo per altri secondi 16, SERVER attivo per altri secondi 15, SERVER attivo per altri secondi 14, SERVER attivo per altri secondi 13.

**B** per il **client**:



The terminal window title is "Blue!: Terminal Window - ua3vol3.java". The content shows a message from the client: Ho ricevuto dati di lunghezza: 29 da : /192.168.1.100 porta :51951. Below it is the date and time: Wed Aug 31 13:00:47 CEST 2016.



## Prova adesso!

Realizza una applicazione di diffusione in **multicast** della "Divina Commedia": il **server** inizia a trasmettere a intervalli di 1 secondo pacchetti contenenti ciascuno una riga di un file di testo così strutturata:

*cantica # numero canto # numero riga # una riga del canto*

Il **client** che riceve i datagrammi aspetta l'inizio di un nuovo canto e successivamente scrive un file per ogni canto, con la seguente struttura:

*cantica\_nrcanto.txt* (per esempio *inferno\_canto5.txt*).

Il file da trasmettere è anche disponibile nella cartella materiali della sezione del sito [www.hoepiscuola.it](http://www.hoepiscuola.it) dedicata a questo volume.

# ESERCITAZIONI DI LABORATORIO 7

## UN ESEMPIO COMPLETO CON LE JAVA SOCKET: "LA CHAT"

Come esempio completo di applicazione Web realizziamo una semplice chat tra un **server** e un numero N di **client** (per esempio 10 **client**).

Per la gestione della ricezione e dell'invio dei messaggi utilizziamo quindi il meccanismo **multithread**, in quanto non si può prevedere il momento in cui ogni singolo **client** fa richiesta di invio di un nuovo messaggio.

Definiamo un **thread** con il compito di rimanere in ascolto su una determinata porta (la nota 6789) in attesa di connessioni da parte di un **client**; il **server**, a ogni nuova connessione, provvede a istanziare un nuovo oggetto dedicato con il compito di mantenere la connessione **client-server** per tutta la durata del suo collegamento e di occuparsi di ricevere e spedire i messaggi a quel **client**.

Vediamo ora il **thread** che si occupa di ricevere le connessioni dei **client**:

```

1  public class ThreadGestioneServizioChat implements Runnable
2  {
3      private int nrMaxConnessioni;
4      private List lista;
5      private ThreadChatConnessioni[] listaConnessioni;
6      Thread me;
7      private ServerSocket serverChat;
8
9      public ThreadGestioneServizioChat(int numeroMaxConnessioni, List lista)
10     {
11         this.nrMaxConnessioni = nrMaxConnessioni-1;
12         this.lista = lista;
13         this.listaConnessioni = new ThreadChatConnessioni[this.nrMaxConnessioni];
14         me = new Thread(this);
15         me.start();
16     }
17 }
```

Al costruttore passiamo come parametro il numero massimo di connessioni e la lista dei messaggi. Vediamo ora il **thread** che si occupa di ricevere le connessioni dei **client**: per prima cosa viene creato un **socket** sulla porta 6789:

```

11  public void run()
12  {
13      boolean continua = true;
14      //Instanzio la connessione del server per la chat
15      try{
16          serverChat = new ServerSocket(6789);
17      }catch(Exception e){
18          JOptionPane.showMessageDialog(null,"Impossibile instanziare il server");
19          continua = false;
20      }
21 }
```

Se questa operazione va a buon fine, si procede con la creazione di un **thread** per ogni connessione:

```

22     if(continua){
23         //accetto le connessioni chat
24         try {
25             for(int xx=0;xx<nrMaxConnessioni;xx++){
26                 Socket tempo=null;
27                 tempo = serverChat.accept();
28                 listaConnessioni[xx] = new ThreadChatConnessioni(this,tempo);
29             }
30             serverChat.close();
31         }catch(Exception e){
32             JOptionPane.showMessageDialog(null,"Impossibile instanziare server chat");
33         }
34     }
35 }
36 //fine metodo "run"

```

L'elenco delle connessioni viene memorizzato nell'array **listaConnessioni**.

La classe viene completata con un metodo che invia un messaggio a tutti i **client**:

```

47     public void spedisceMessaggio(String mex)
48     {
49         //scrivo il mex nella mia lista
50         lista.add(mex);
51         lista.select(lista.getItemCount()-1);
52         //mando il messaggio agli altri
53         for(int xx=0; xx<this.nrMaxConnessioni;xx++){
54             if(listaConnessioni[xx] != null){
55                 listaConnessioni[xx].spedisceMessaggioChat(mex);
56             }
57         }
58     }
59 }
60 //fine classe ThreadGestioneServizioChat

```

Descriviamo ora la classe che viene associata a ogni **client** connesso con il **server**: si ottengono gli stream di I/O e si crea un metodo che permette di spedire i messaggi al **client**.

(Le variabili non sono commentate in quanto i loro identificativi “parlanti” indicano a che cosa servono.)

```

4  public class ThreadChatConnessioni implements Runnable
5  {
6      private ThreadGestioneServizioChat gestoreChat;
7      private Socket client = null;
8      private BufferedReader input = null;
9      private PrintWriter output = null;
10     Thread me;
11
12     public ThreadChatConnessioni(ThreadGestioneServizioChat gestoreChat,Socket client)
13     {
14         this.gestoreChat = gestoreChat;
15         this.client= client;
16         try{
17             this.input = new BufferedReader(new InputStreamReader(client.getInputStream()));
18             this.output = new PrintWriter(this.client.getOutputStream(),true);
19         }catch(Exception e){
20             output.println("Errore nella lettura.");
21         }
22         me = new Thread(this);
23         me.start();
24     }

```

Le operazioni eseguite nel costruttore sono comunque note:

- creazione e inizializzazione dello stream input come oggetto `BufferedReader`;
- creazione e inizializzazione dello stream output come oggetto `PrintWriter`;
- inizializzazione di `gestoreChat` ricevuto come parametro;
- inizializzazione di `Socket client` ricevuto come parametro.

Quindi viene creato un `thread` e mandato in esecuzione.

Vediamo di seguito l'implementazione del metodo `run()`.

```

16 public void run()
17 {
18     while(true){
19         try{
20             String mex=null;
21             //rimango in attesa dei messaggi inviati dal client
22             while((mex = input.readLine())!=null)
23             {
24                 //invoco il metodo del gestoreChat per ripetere a tutti il messaggio ricevuto
25                 gestoreChat.spedisceMessaggio(mex);
26             }catch(Exception e)
27             {
28                 output.println("Errore nella spedizione del messaggio a tutti.");
29             }
30         }
31     }
32 }
```

Completa la classe un semplice metodo che spedisce un messaggio a un singolo `client`.

```

44 public void spedisceMessaggioChat(String messaggio)
45 {
46     try{
47         output.println(messaggio);
48     }catch(Exception e){
49         output.println("Errore nella spedizione del singolo messaggio.");
50     }
51 }
```

Vediamo ora l'implementazione del servizio `client` della chat.

Il `client` ha solamente il compito di connettersi con il `server`, di rimanere in attesa di messaggi inviati dal `server` e, su richiesta dell'utente, deve inviare nuovi messaggi verso il `server`.

Analizziamo in dettaglio il codice: la classe definisce le variabili private utilizzate da ogni singolo `thread` ► e riceve nel costruttore la lista dei `thread` e i parametri del `server` al quale collegarsi. ▼

```

1 public class ThreadChatClient implements Runnable
2 {
3     private List lista;
4     Thread me;
5     private Socket client;
6     private BufferedReader input=null;
7     private PrintWriter output=null;
```

```

14     public ThreadChatClient(List lista, String ipServer, int porta)
15     {
16         this.lista = lista;
17         try{
18             client = new Socket(ipServer,porta);
19             this.input = new BufferedReader(new InputStreamReader(client.getInputStream()));
20             this.output = new PrintWriter(client.getOutputStream(),true);
21         }catch(Exception e){
22             JOptionPane.showMessageDialog(null,"Impossibile connettersi al server");
23         }
24         me = new Thread(this);
25         me.start();
26     }
```

Il metodo **run()** rimane in attesa che l'utente scriva un messaggio, quindi invoca il metodo del **server** affinché il messaggio venga inoltrato a tutti coloro che sono connessi alla chat.

```

16 public void run()
17 {
18     while(true){
19         try
20         {
21             String mex=null;
22             //rimango in attesa dei messaggi mandati dai client
23             while((mex = input.readLine())!=null)
24             {
25                 //invoco il metodo del gestoreChat per ripetere a tutti il messaggio ricevuto
26                 gestoreChat.spedisciMessaggio(mex);
27             }catch(Exception e)
28             {
29                 output.println("Errore nella spedizione del messaggio a tutti.");
30             }
31         }
32     }
33 }
```

Conclude la classe un metodo che invia il messaggio al singolo **client**: ►

```

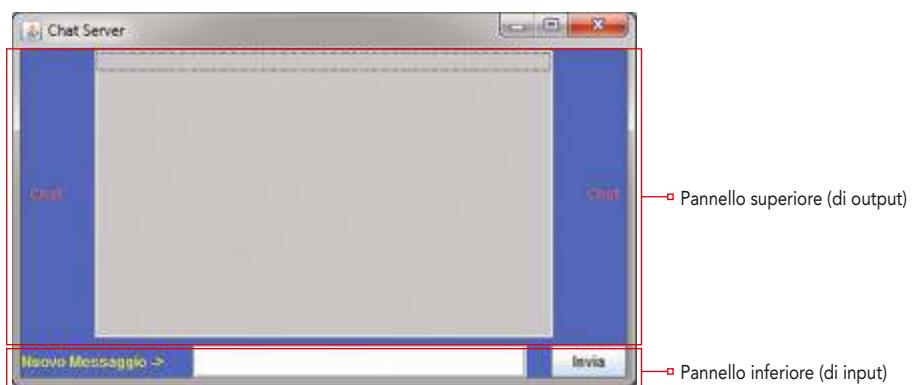
44 public void spedisciMessaggioChat(String messaggio){
45     try{
46         output.printin(messaggio);
47     }catch(Exception e){
48     }
49 }
```

Riportiamo il codice che realizza l'interfaccia grafica solamente per il **server**, poiché è pressoché identica a quella del **client**:

```

1 public class ChatServer extends JFrame
2 {
3     public ChatServer()
4     {
5         super("Chat Server");
6         this.setSize(new Dimension(500,300)); // setto la grandezza della finestra
7         this.setLocationRelativeTo(null); // la metto al centro dello schermo
8         this.setEnabled(true); // setto la proprietà enable
9         this.setBackground(color.blue); // setto il colore di sfondo
10        //creo il pannello per l'inserimento e la visualizzazione dei messaggi
11        PannelloChatServer pan = new PannelloChatServer();
12        this.getContentPane().add(pan);
13        this.setVisible(true);
14    }
15 }
```

Nella classe viene definito un oggetto della classe:



L'interfaccia è costituita da due parti:

- un pannello superiore (di output) dove vengono visualizzati tutti i messaggi;
- un pannello inferiore (di input) dove l'utente scrive il proprio messaggio per inviarlo al **server** (e quindi agli altri utenti collegati).

La parte superiore del pannello è realizzata con questo segmento di codice: ►

```

17 public PannelloChatServer()
18 {
19     super();
20     this.setBackground(new Color(50, 100, 255));
21     // pannello superiore: lista messaggi
22     JPanel panLista = new JPanel(new BorderLayout(20,5));
23     panLista.setBackground(new Color(50, 100, 255));
24     lista = new List();
25     lista.setBackground(Color.lightGray);
26     lista.setSize(100,50);
27     lista.setVisible(true);
28     // scritte laterali
29     JLabel chat1 = new JLabel(" Chat ",JLabel.CENTER);
30     chat1.setForeground(new Color(200,100,100));
31     JLabel chat2 = new JLabel(" Chat ",JLabel.CENTER);
32     chat2.setForeground(new Color(200,100,100));
33     // aggiungiamo gli oggetti sul pannello
34     panLista.add(chat1,BorderLayout.WEST);
35     panLista.add(lista,BorderLayout.CENTER);
36     panLista.add(chat2,BorderLayout.EAST);

```

La parte inferiore del pannello è realizzata con questo segmento di codice: ►

```

42 //pannello inserimento nuovo messaggio
43 JPanel nuovoMex = new JPanel(new BorderLayout(20,5));
44 nuovoMex.setBackground(new Color(50, 100, 255));
45
46 JLabel labNuovo = new JLabel("Nuovo Messaggio -> ",JLabel.CENTER);
47 labNuovo.setForeground(new Color(255,255,0));
48
49 JTextField textNuovo = new JTextField("");
50
51 JButton buttonInvia = new JButton("Invia");
52 buttonInvia.addActionListener(this);
53 // aggiungiamo gli oggetti sul pannello
54 nuovoMex.add(labNuovo,BorderLayout.WEST);
55 nuovoMex.add(textNuovo,BorderLayout.CENTER);
56 nuovoMex.add(buttonInvia,BorderLayout.EAST);
57
58 this.setLayout(new BorderLayout(0,5));
59 add(panLista,BorderLayout.CENTER);
60 add(nuovoMex,BorderLayout.SOUTH);
61
62 connetti();
63 } //fine costruttore classe PannelloChat

```

Concludono la classe due metodi, rispettivamente per avviare il **server** e predisporre la lista delle possibili connessioni (10 utenti massimo) e la gestione dell'evento sul pulsante di invio che inoltra il messaggio. Infatti anche l'utente che avvia il **server** può, di fatto, comunicare con gli altri utenti. ►

```

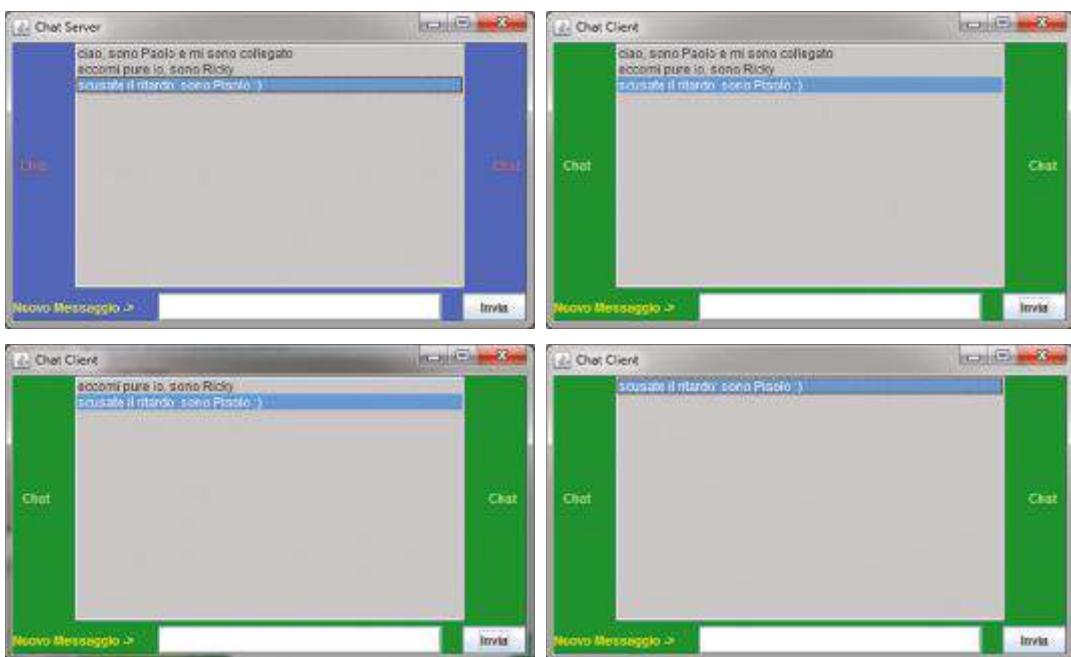
71 public void connetti()
72 {
73     //instanzio il Thread per le connessioni : numero massimo giocatori = 10
74     gestioneServizio = new ThreadGestioneServizioChat(10,lista);
75 }
76
77 public void actionPerformed(ActionEvent e)
78 {
79     String bottone = e.getActionCommand();
80     if(bottone.equals("Invia"))
81     {
82         gestioneServizio.spedisceMessaggio(textNuovo.getText());
83         textNuovo.setText("");
84     }
85 }

```

Un esempio di esecuzione è il seguente:



In questa prima situazione abbiamo avviato il **server** e un **client** e inviato un primo messaggio. Aggiungiamo ora altri due **client** e inviamo un messaggio da ciascuno di essi: possiamo osservare come la comunicazione viene inviata a tutte le finestre.



L'unico problema che possiamo evidenziare è che non sono presenti i nomi dei mittenti, e quindi "non si sa di chi sono i messaggi"!



### Prova adesso!

Modifica l'esempio sopra descritto aggiungendo una form di login in modo che l'utente inserisca il proprio nome e questo venga visualizzato nella sezione di output vicino a ogni messaggio, in modo da riconoscerne la "paternità".

Quindi fai le opportune modifiche per poter utilizzare questa applicazione in rete, creando uno "sniffer" che ricerca sulla rete l'indirizzo IP del **server** che offre questo servizio sulla porta 6789.

# ESERCITAZIONI DI LABORATORIO 8

## I SOCKET NEL LINGUAGGIO C

### ■ Il dominio Internet e gli indirizzi IP

La gestione dei **socket** in linguaggio **C** è più complessa rispetto a **Java**: sappiamo che un **socket** è solitamente legato a un indirizzo e il formato dell'indirizzo dipende dal **dominio di comunicazione**.

Ciascun dominio ha un suo nome simbolico che inizia con **PF\_**, iniziali di **protocol family**, altro nome con cui si indicano i **domini di comunicazione**: a ciascun dominio viene inoltre associata una seconda costante, che inizia invece **AF\_** (da **address family**), che identifica il formato degli indirizzi usati in quel **dominio**.

I principali **domini** implementati sono descritti in **<sys/socket.h>** e sono:

- il dominio **UNIX (AF\_UNIX)**;
- il dominio **Internet (AF\_INET, AF\_INET6)**;
- il dominio **Novell (AF\_IPX)**;
- il dominio **AppleTalk (AF\_APPLETALK)**.

I processi per comunicare nello stesso **dominio** usano lo stesso formato di indirizzi.

Nel **dominio internet** abbiamo due possibili situazioni per un **indirizzo IP**:

- **AF\_INET**: è costituito da quattro decimali interi separati da punti e consente di individuare univocamente la posizione di una sottorete e di un host all'interno di quest'ultima.  
Viene memorizzato in un record così definito nell'header file **<netinet/in.h>**:

```
struct in_addr
{
    u_long s_addr;           /* 4 byte / 32 bit IP V4 */
}
```

- **AF\_INET6**: utilizza il formato IP V6 per gli indirizzi e quindi saranno necessari per questo dominio record con 6 byte.

Dato che i valori sono memorizzati come numeri, nella libreria **<arpa/inet.h>** sono disponibili alcune funzioni per la conversione di **indirizzi IP** sia nella forma **dotted-decimal ASCII string** (cioè nella forma 192.168.0.1) che nella forma **32-bit network byte ordered** (formato binario).

### **inet\_addr()**

Ritorna un indirizzo IP come numero a 32 bit a partire dal formato stringa **dotted decimal**.

```
int inet_addr (const char *str)
```

## inet\_aton()

Converte la stringa dell'indirizzo **dotted decimal** in un indirizzo IP e lo memorizza.

```
int inet_aton (const char *str, struct in_addr *addrptr);
```

Scrive nel record puntato da **addrptr** il valore numerico a 32 bit ottenuto dalla conversione della stringa presente in **str**. Restituisce zero in caso di errore, 1 se tutto va bene.

## inet\_ntoa()

Converte un indirizzo IP presente nel record **addrptr** in una stringa **dotted decimal**.

```
char* inet_ntoa(struct in_addr addrptr)
```

Restituisce il puntatore alla stringa che contiene l'espressione in formato dotted decimal. Per effettuare l'assegnazione a una variabile è comodo utilizzare la funzione **sprintf()** che memorizza in una stringa quello che è destinato all'output formattato sullo schermo:

```
char s_IP_Remoto[15];           // indirizzo formato stringa
struct in_addr remoto_addr     // indirizzo formato numerico
...
sprintf((char*)s_IP_Remoto,"%s", inet_ntoa(remoto_addr));
```

Se volessimo effettuare una assegnazione diretta dovremmo necessariamente eseguire complesse operazioni di casting.

## inet\_ntop()

Anche se ufficialmente **inet\_ntoa()** non è deprecata se si vuole salvare l'indirizzo in formato stringa nelle variabili è conveniente utilizzare **inet\_ntop()** che risulta essere maggiormente compatibile nelle versioni di **Linux** più recenti.

```
char *inet_ntop (AF_INET, const void *src, char *dest, sizeof(dest));
```

Restituisce il puntatore alla stringa che contiene l'espressione in formato dotted decimal che assume il valore NULL in caso di errore.

### ESEMPIO

```
char s_IP_Remoto[15];           // indirizzo formato stringa
struct in_addr remoto_addr     // indirizzo formato numerico
inet_ntop(AF_INET, &remoto_addr, s_IP_Remoto, sizeof(s_IP_Remoto));
```

## ■ Strutture dati per i socket

Come abbiamo detto i **socket** di tipo **AF\_INET/PF\_INET** vengono usati per la comunicazione attraverso Internet; la struttura predisposta per memorizzare i dati per un socket Internet è definita nell'header file **<netinet/in.h>** come:

```
struct sockaddr_in
{
    short sin_family;          /* valore AF_INET */
    u_short sin_port;          /* numero porta TCP/UDP (16 bit) */
    struct in_addr sin_addr;   /* indirizzo IP */
    char sin_zero[8];          /* riempimento (non usato) */
};
```

dove :

- ▷ **sin\_family** deve essere sempre impostato a **AF\_INET**, altrimenti si avrà un errore di **EINVAL**;
- ▷ **sin\_addr** contiene un indirizzo Internet e ad esso si può accedere sia come struttura che direttamente come numero intero; se viene impostato a **INADDR\_ANY** si connette automaticamente all'indirizzo locale che ha l'interfaccia di rete dell'host al momento dell'esecuzione del programma;
- ▷ **sin\_port** specifica il numero di porta, che deve essere superiore a 1024.

### ESEMPIO

Vediamo ad esempio per un **server** come potrebbe essere composto questo record:

```
/*creazione di un elemento di tipo sockaddr_in
struct sockaddr_in datiSocket;
/* inizializzazione dell'elemento creato
datiSocket.sin_family = AF_INET;
datiSocket.sin_addr.s_addr = inet_addr( "127.0.0.1" );
datiSocket.sin_port = htons( 27015 )
```

dove la funzione **htons()** trasforma un numero dal formato interno a quello della rete (Big-Endian).

### AREA digitale

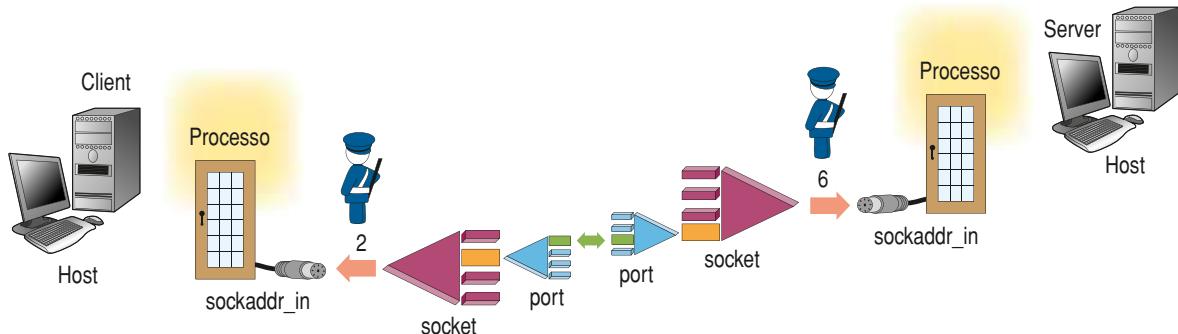


Funzioni di conversione Big-Endian/Little-Endian

Per la creazione di un **socket server** è anche disponibile la definizione di una struttura ridotta:

```
struct sockaddr
{
    unsigned short sa_family; /* valore address family (AF_INET)      */
    char sa_data[  ];           /* per memorizzare indirizzo famiglia */
};
```

La visione generale dell'architettura **client-server** è la seguente.



## ■ Comunicazione mediante socket

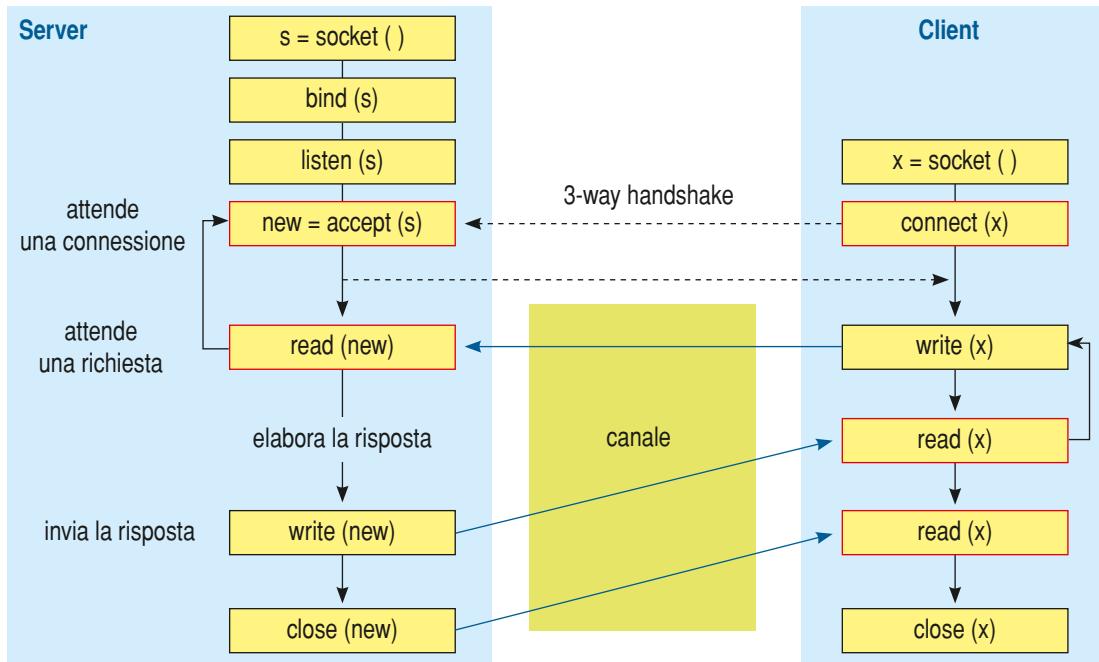
Nella realizzazione della comunicazione utilizzando i **socket** come primo attore viene avviata l'esecuzione del **server** che si predisponde alla connessione e solo successivamente viene avviato il **client** che chiede la connessione al **server** che, essendo in esecuzione, la può attivare.

Il “principio di funzionamento che utilizza i socket” è molto simile a quello utilizzato per la gestione dei file:

- ▷ apertura del canale;
- ▷ lettura/scrittura dei messaggi;
- ▷ chiusura del canale.

L'esempio tipico di funzionamento è quello descritto di seguito dove il **client** invia una *richiesta di connessione* al **server** alla quale quest'ultimo risponde trasmettendo alcuni dati necessari per instaurare la connessione: quindi si procede con una sequenza di trasmissioni bidirezionali che termina quando uno dei due interlocutori decide di interrompere la connessione inviando un apposito comando (mediante la primitiva `close()`).

La sequenza completa delle operazioni è descritta nel diagramma di figura dove sono indicate le diverse funzioni chiamate rispettivamente dal **server** e dal **client**.



Descriviamo una per una le singole funzioni.

### Funzione `socket()`

I **socket** costituiscono gli estremi del canale di comunicazione e prendono il nome di **transport endpoint**: i due processi che vogliono comunicare attraverso la rete devono definire i rispettivi **socket** tramite una chiamata di sistema che restituisce un numero (in una variabile intera indicata con identificatore `mysocketfd`) maggiore o uguale a zero mentre in caso di errore restituisce (-1) e setta **errno** con uno dei seguenti valori in modo da riconoscere le cause del fallimento:

- ▷ **EPROTONOSUPPORT**: il tipo di socket o il protocollo scelto non sono supportati nel dominio;
- ▷ **ENFILE**: il kernel non ha memoria sufficiente a creare una nuova struttura per il socket;
- ▷ **EMFILE**: si è ecceduta la tabella dei file;
- ▷ **EACCES**: non si hanno privilegi per creare un socket nel dominio/protocollo specificato;
- ▷ **EINVAL**: protocollo sconosciuto o dominio non disponibile;
- ▷ **ENOBUFS**: non c'è sufficiente memoria per creare il socket (può essere anche **ENOMEM**).

Quindi la prima azione per “avviare la comunicazione” in rete è la chiamata alla funzione `socket()` con la specifica del tipo di protocollo da utilizzare, ad esempio **TCP** con **IPv4**, **UDP** con **IPv6** ecc.:

```
int socket(int domain, int type, int protocol);
```

dove:

- ▷ **domain** indica il dominio del **socket**, che per internet è **AF\_INET**;
- ▷ **type** indica la modalità di comunicazione, cioè:

- **SOCK\_STREAM** se verrà usato il paradigma **connection oriented**;
- **SOCK\_DGRAM** se verrà utilizzato il paradigma **connection less**;
- **protocol** specifica il protocollo da utilizzare e nel caso che venga lasciato al valore 0 il sistema automaticamente sceglie il protocollo più adatto in base ai primi due parametri.

### ESEMPIO

```
#include <sys/socket.h>
/* creazione e definizione del socket di tipo stream TCP */
mysocketfd = socket(AF_INET, SOCK_STREAM, 0);
```

## Funzione bind()

Per legare l'indirizzo dell'host a un **socket identifier** in modo da poter essere contattato da un client si utilizza la funzione **bind()** che “crea un legame” tra il **socket identifier** e il record che contiene i dati del **socket**:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int mysocketfd, const struct sockaddr *address, size_t add_len);
```

dove:

- **mysocketfd** è il valore restituito dalla system call **socket()**;
- **address** è il puntatore alla struttura dati che contiene l'indirizzo e la porta del socket;
- **add\_len** è la lunghezza in byte dell'indirizzo.

La funzione restituisce 0 in caso di successo, altrimenti -1.

La funzione **bind()** viene utilizzata dal server per “effettuare il collegamento” a una porta specifica: se non viene eseguita la chiamata alla funzione **bind()**, il sistema operativo assegnerà al **socket** una porta qualunque e uno degli indirizzi IP dell'host: inoltre nel caso di protocollo TCP/IP se nel record la porta è specificata come zero, il fornitore di servizi assegna una porta tra 1024 e 5000.

Successivamente alla **bind()** l'applicazione server può usare la funzione **getsockname()** per apprendere l'indirizzo IP e la porta assegnati.

## Funzione listen()

Il **server** si mette in ascolto in attesa dei client per mezzo della seguente system call:

```
#include <sys/socket.h>
int listen(int mysocketfd, int queue_size);
```

dove:

- **mysocketfd** è il socket creato sul server;
- **queue\_size** è il numero massimo di client che si possono accodare in attesa della connessione.

La funzione restituisce 0 in caso di successo e -1 in caso di errore con i seguenti valori restituiti in **errno**:

- **EBADF** l'argomento **mysocketfd** non è un file descriptor valido;
- **ENOTSOCK** l'argomento **mysocketfd** non è un socket;
- **EOPNOTSUPP** il socket è di un tipo che non supporta questa operazione.

### ESEMPIO

Utilizzando quanto detto sino a ora, iniziamo a vedere la struttura di un programma **server** che utilizza i socket:

```
#include <sys/socket.h>
void main(){
    int mysocketfd, ris;                                // definizione variabili
    struct sockaddr_in socketServer;                   // record dati del server
    /* inizializzazione con i dati del server */
    socketServer.sin_family = AF_INET;
    socketServer.sin_addr.s_addr = inet_addr("127.0.0.1");
    socketServer.sin_port = htons(27015)

    /* creazione e definizione del tipo di socket (stream TCP) */
    mysocketfd = socket(AF_INET, SOCK_STREAM, 0);

    /* assegnazione dell'indirizzo al socket */
    ris = bind(mysocketfd, (struct sockaddr*)&socketServer, sizeof(socketServer));

    /* pone il server in ascolto sul socket */
    ris = listen(mysocketfd, 10);
}
```

## Funzione accept()

Il server accetta una connessione tramite la chiamata di sistema `accept()`:

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int mysocketfd, struct sockaddr* address, size_t* add_len);
```

La funzione estrae la prima connessione dalla coda di pendenza delle connessioni sul socket in input: crea un nuovo socket che verrà usato per comunicare con il client ed il suo valore viene passato come parametro di ritorno di `accept()`.

Il socket in input non cambia e continua a restare in ascolto per nuove richieste.

I due argomenti `address` e `add_len` sono usati dal server per ottenere l'indirizzo del client da cui proviene la richiesta di connessione.

La funzione restituisce un numero di `socket descriptor` positivo in caso di successo e -1 impostando `errno` ai seguenti valori:

- **EBADF** l'argomento `mysocket` non è un file descriptor valido;
- **ENOTSOCK** l'argomento `mysocket` non è un socket;
- **EOPNOTSUPP** il socket è di un tipo che non supporta questa operazione;
- **EAGAIN** o **EWOULDBLOCK** il socket è stato impostato come non bloccante e non ci sono connessioni in attesa di essere accettate.

Aggiungiamo all'esempio precedente il comando `accept()`:

```
/* definizione delle variabili */
int clientSocketfd;           // identificatore del socket client
struct sockaddr_in client;    // struttura del client address
int clientLen;                // dimensione del client address
...
clientLen = sizeof(client);
clientSocketfd = accept(mysocketfd, (struct sockaddr*)&client, &clientLen);
```

Prima della chiamata della funzione `accept()` deve essere inizializzato `clientLen` alle dimensioni della struttura il cui indirizzo è passato come argomento.

### Funzione `close()`

Un `socket` viene distrutto chiudendo il file descriptor associato alla connessione.

La funzione standard unix `close()` che si usa sui file può essere usata con lo stesso effetto anche sui file descriptor associati a un socket.

```
#include <unistd.h>
int close(int mysocketfd);
```

L'azione di questa funzione quando applicata a un socket è di marcarlo come chiuso e ritornare immediatamente al processo: dopo la sua esecuzione il socket descriptor non è più utilizzabile dal processo e non può essere usato come argomento per le operazioni di lettura e/o scrittura.

#### ESEMPIO

```
/* chiusura*/
close(mysocketfd);
```

## ■ L'ambiente di sviluppo e di esecuzione

Come già descritto nel volume 2 (esercitazione di laboratorio 1, unità 1), se non si dispone di una partizione con `Linux` è possibile utilizzare l'emulatore `Cygwin` che è installabile seguendo le istruzioni presenti alla pagina <http://www.cygwin.com>.

Nei nostri esempi faremo riferimento a entrambe le situazioni.

# ESERCITAZIONI DI LABORATORIO 9

## SERVER TCP IN C

### ■ Funzioni per inviare e ricevere dati

Nella libreria `<sys/socket.h>` sono disponibili due gruppi di funzioni per effettuare le operazioni di comunicazione con il server: noi le useremo entrambe per testarne le funzionalità.

#### Funzioni `read()` e `write()`

##### `read()`

La struttura di una istruzione di `read()` è la seguente:

```
int read (int socketfd, void *buffer, int tanti);
```

Cerca di leggere `tanti` byte dal file descriptor `socketfd`, scrivendoli nel buffer puntato da `buffer`: se `tanti` è zero la `read()` restituisce zero altrimenti viene effettuata la lettura e viene restituito il numero di byte letti, e se il valore di ritorno è:

- ▶ maggiore di zero, significa che tutto è OK;
- ▶ uguale a 0 (zero), significa end-of-file (fine stream), cioè l'altro end system ha volutamente chiuso la connessione e il `socket` non può essere più utilizzato per leggere;
- ▶ uguale a -1, significa che si è verificato un errore e viene settata la variabile globale `errno` definita in `<errno.h>` con un valore che indica quale errore è avvenuto.

La funzione `read()`, applicata a un `socket`, presenta una particolarità: può accadere che la `read()` restituisca meno byte di quanti richiesti, anche se lo stream è ancora aperto e questo si verifica nel caso in cui il buffer a disposizione del `socket` nel kernel è stato esaurito.

In questo caso è necessario ripetere la `read()` richiedendo il numero dei byte mancanti.

##### `write()`

La struttura di una istruzione di `write()` è la seguente:

```
int write (int socketfd, void *buffer, int tanti);
```

Cerca di scrivere fino a `tanti` byte nel buffer di sistema corrispondente al file descriptor `socketfd` perché siano poi trasmessi leggendoli dal buffer puntato da `buffer`.

#### Funzioni `send()` e `recv()`

Esistono altre due funzioni per trasmettere e ricevere dati attraverso i socket che oltre a effettuare le medesime operazioni delle coppia `read()/write()` offrono la possibilità di configurare in modo più specifico il comportamento in casi particolari utilizzando un ulteriore parametro (`int flag`): sono le system call `send()` e `recv()`.

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int mysockfd, const char *buffer, int length, int flag);
int recv(int mysockfd, void *buffer, int length, int flag);
```

dove:

- **buffer** contiene la stringa da trasmettere;
- **length** è la dimensione in byte del buffer;
- **flag** assume valori diversi per modificare il funzionamento della funzione:
  - 0: **send()** e **recv()** equivalgono, rispettivamente alle system call **write()** e **read()**;
  - altri valori per la **send()**:
    - **MSG\_OOB**: il processo invia dati “out of band”;
    - **MSG\_DONTROUTE**: vengono ignorate le condizioni di routing dei pacchetti sottostanti al protocollo utilizzato;
  - altri valori per la **recv()**:
    - **MSG\_PEEK**: i dati vengono letti, ma non “consumati” in modo che una successiva **recv()** riceverà ancora le stesse informazioni;
    - **MSG\_OOB**: legge soltanto i dati “out of band”;
    - **MSG\_WAITALL**: la **recv()** ritorna soltanto quando la totalità dei dati è disponibile.

## ■ Realizzazione di un server

Per poter collaudare un sistema **client-server** deve naturalmente essere scritto il codice di entrambi gli attori: come primo esempio completo creiamo una applicazione **connection oriented** dove si vuole realizzare un sistema con un **server** che riceve un carattere dal **client** e lo restituisce trasformato in maiuscolo fino a che gli arriva il carattere “**x**”, che chiude la trasmissione.

In questa lezione realizzeremo il **server** mentre nella prossima descriveremo il **client**.

Dopo aver incluso le librerie indichiamo come costante la porta che viene utilizzata nel server: per tutti i nostri esempi utilizzeremo sempre la porta **1313**.

```
serverMaiu.c
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <stdio.h>
5
6 #define SERVER_PORT 1313           // numero di porta del server
7 #define SOCKET_ERROR   ((int)-1)    // codice di errore
8 #define FINE_RICE 88              // valore ASCII carattere X
```

Il **main()** definisce alcune variabili di utilità e la struttura del **socket**:

```
24
25 int main (unsigned argc, char **argv) {
26     int socketAttesa, client_len, socketComunica;
27     int asciiletto;                      // valore ascii ultimo char ricevuto
28     struct sockaddr_in server, client;    // record con i dati del server e del client
29 }
```

Viene quindi effettuata la sua inizializzazione:

```
30     // impostazione del transport endpoint del server
31     printf ("socket()\n");
32     if((socketAttesa = socket(AF_INET, SOCK_STREAM, 0)) == -1){
33         perror("chiamata della system call socket() fallita");
34         return(1);
35     }
```

```

36 // definizione dei dati del socket
37 server.sin_family = AF_INET;
38 server.sin_addr.s_addr = htonl(INADDR_ANY);
39 server.sin_port = htons(SERVER_PORT);

```

Viene effettuato il `bind()` passando come parametro il `socket` e la sua dimensione:

```

41 // leghiamo l'indirizzo al transport endpoint
42 printf ("bind()\n");
43 if (bind(socketAttesa, (struct sockaddr*)&server, sizeof server) == -1){
44     perror("chiamata della bind() fallita");
45     return(2);
46 }

```

Quindi ci si pone in attesa di un client:

```

48 // poniamo il server in ascolto delle richieste dei client
49 printf ("listen()\n");
50 listen(socketAttesa, 10);           // può attendere fino a 10 client
51

```

Alla richiesta di connessione da parte di un client questa viene valutata:

```

52 // gestione delle connessioni dei client
53 printf ("accept()\n");
54 // trasferimento della comunicazione su un nuovo socket dedicato per il client
55 client_len = sizeof(client);

```

e quando l'esito è positivo si esce dal ciclo di attesa e inizia la comunicazione;

```

61 // inizio dialogo tra server e client
62 fprintf(stderr, "Aperta connessione col server.\n"); // echo nel client
63 send(socketComunica, "Hello dal ServerMaiu: trasformo caratteri in maiuscolo!\n", 56, 0);
64
65 // ciclo di attesa/comunicazione di un carattere alla volta col client
66 do // funzione che riceve un carattere, lo trasforma in maiuscolo, lo ritrasmette
67     asciletto = upper1Char(socketComunica, socketComunica);
68 while (asciletto != FINE_RICE); // se arriva x termina l'elaborazione
69
70

```

Quando riceve una “x” la comunicazione termina e il server chiude la connessione:

```

71 // fine della comunicazione
72 close(socketComunica);
73 fprintf(stderr, "Chiusa connessione col server.\n"); // echo nel client
74 }

```

Mancava ora da riportare la funzione che effettua la ricezione dei dati e li trasforma in maiuscolo; una possibile codifica è la seguente:

```

10 // funzione che riceve un carattere, lo trasforma in maiuscolo, lo ritrasmette
11 int upper1Char(int in, int out) {
12     char ch, chMaiu;
13     int len, ascival;
14     if(recv(in, &ch, 1, 0) > 0) {          // riceve un carattere
15         chMaiu = toupper(ch);            // lo trasforma in maiuscolo
16         ascival= (unsigned int)ch;       // prende il valore ASCII
17         printf("\nil carattere ricevuto e': %c con valore ASCII %d ", ch, ascival);
18         printf(" maiuscolo %d \n", (unsigned int)chMaiu);
19         send(out, &chMaiu, 1, 0);        // invia il carattere maiuscolo
20         ascival= (unsigned int)chMaiu;
21     }
22 }
23

```

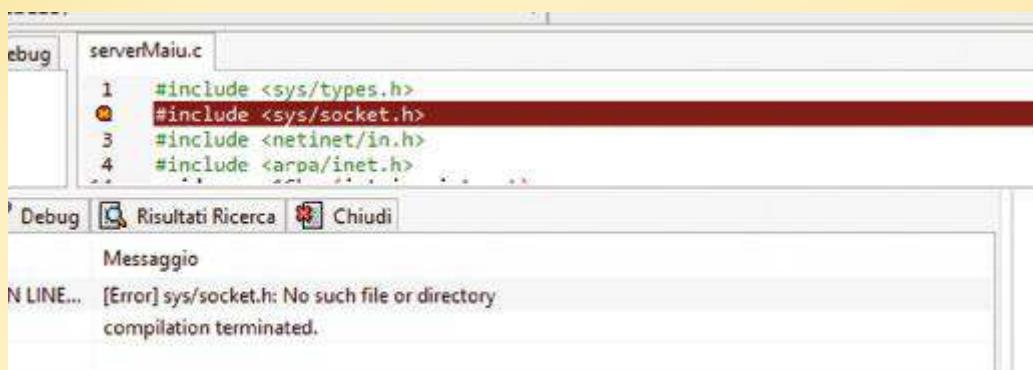
Il codice completo del server è memorizzato nel file `serverMaiu.c`.

## Avvio del server

Verifichiamo il funzionamento del programma: per prima cosa ci posizioniamo nella directory **socket** dove è presente il codice sorgente **serverMaiu.c** e lo compiliamo.

```
Paolo@PCwin8 ~
$ cd ua3vo13
Paolo@PCwin8 ~/ua3vo13
$ gcc -o serverMaiu serverMaiu.c
Paolo@PCwin8 ~/ua3vo13
$ |
```

Se ci viene segnalata la mancanza di qualche libreria con un messaggio simile a quello riportato:



significa che abbiamo erroneamente avviato la compilazione in ambiente **Dev-cpp** dove le librerie indicate non sono presenti dato che riguardano **socket** in ambiente **Linux**.

Avviamo il server e come **echo** ci riporta la sequenza delle funzioni che ha eseguito ponendosi infinite in attesa della richiesta da parte di un client:

```
Paolo@PCwin8 ~/ua3vo13
$ ./serverMaiu
socket()
bind()
listen()
accept()
```



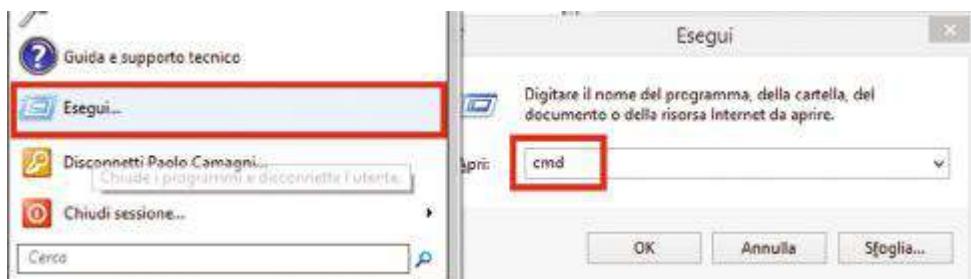
### Prova adesso!

- Server TCP
- Funzioni **read()** e **write()**

- 1 Modifica l'esempio descritto utilizzando per la comunicazione le funzioni **read()** e **write()** prima descritte.
- 2 Confronta il tuo codice con quello riportato nel file **serverMaiuRW.c**.

## Telnet come client

Per verificare il funzionamento del **server**, dato che non abbiamo ancora realizzato il codice del **client**, possiamo servirci del programma **telnet** posizioniamoci in una finestra di comando digitando: a tale scopo digitiamo **cmd** dalla finestra di esecuzione dei programmi:



e successivamente: **telnet 127.0.0.1 1313**

```
Microsoft Windows [Versione 6.3.9600]
(c) 2013 Microsoft Corporation. Tutti i diritti riservati.
C:\Windows\System32>telnet 127.0.0.1 1313...
```

È anche possibile richiamare e avviare **telnet** direttamente dalla riga di ricerca:



## AREA digitale

Abilitare telnet in Windows 8 e Windows 8.1

Confermiamo con invio e se la connessione va a buon fine avremo i messaggi seguenti: sul **server**:

```
Paolo@PCwin8 ~
$ cd ua3vol3
Paolo@PCwin8 ~/ua3vol3
$ ./serverMain
socket()
bind()
listen()
accept()
Aperta connessione col server.
```

sul **client**:

```
Hello al ServerMain: trasformo caratteri in maiuscolo!
```

Qualunque carattere che ora digitiamo ci viene “rispedito” convertito in maiuscolo:

```
Hello al ServerMain: trasformo caratteri in maiuscolo! aBbBcCdDs$
```

e sul server avremo l’echo dei caratteri inviati dal client.

Per terminare la sessione premere **[CTRL] +** e imparire nella nuova riga il comando **quit**:

```
Microsoft Telnet Client
Il carattere di Escape è 'CTRL+'
Microsoft Telnet> quit
```

Nella finestra dove abbiamo in esecuzione il **server** viene visualizzato un messaggio sia nel momento dell'inizio della connessione **telnet** che alla chiusura oltre agli **echo** delle singole istruzioni che vengono eseguite:

```
Paolo@PCwin8 ~
$ cd ua3vol3
Paolo@PCwin8 ~/ua3vol3
$ ./serverMaiu
socket()
bind()
listen()
accept()
Aperta connessione col server.

il carattere ricevuto e': a con valore ASCII 97 maiuscolo 65
il carattere ricevuto e': b con valore ASCII 98 maiuscolo 66
il carattere ricevuto e': c con valore ASCII 99 maiuscolo 67
il carattere ricevuto e': d con valore ASCII 100 maiuscolo 68
il carattere ricevuto e': s con valore ASCII 115 maiuscolo 83
Chiusa connessione col server.
```



## Prova adesso!

- Realizzare un server
- Realizzare un client
- Comunicare tra server e client

- 1 Modifica l'esempio descritto in modo che la terminazione avvenga quando il server riceve un qualunque carattere non alfabetico.
- 2 Successivamente bufferizza i caratteri che vengono ricevuti dal server per esempio in un vettore di caratteri:

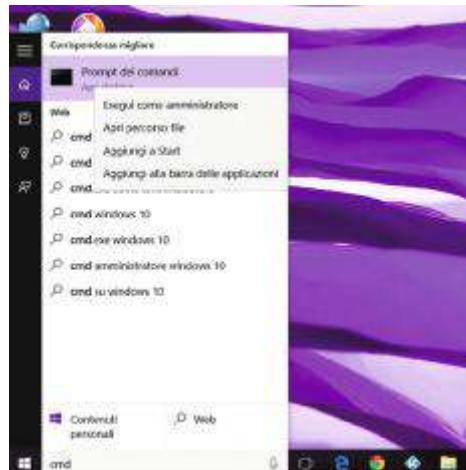
```
#define LINESIZE 80 // dimensione della linea di caratteri
char inputline[LINESIZE];
```

Questo viene spedito interamente convertito al client al termine della ricezione.

## ■ Abilitare Telnet su Windows 10

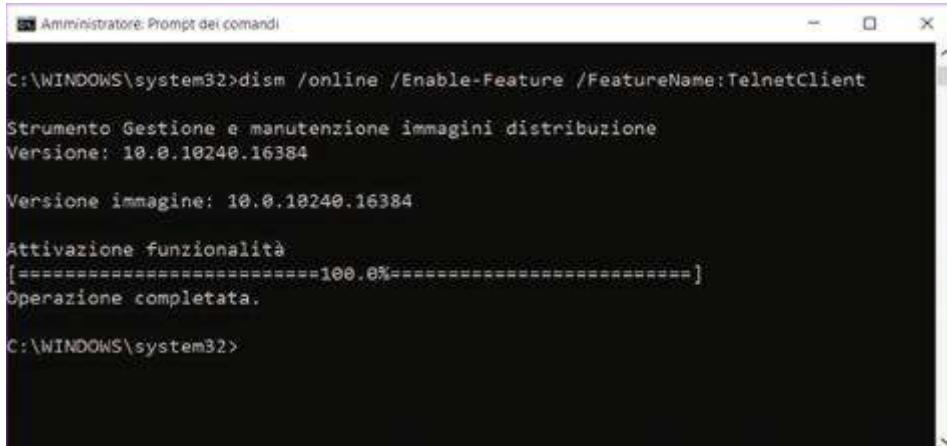
Per impostazione predefinita, nei sistemi operativi Microsoft il **client Telnet** è disattivato; possiamo attivarlo direttamente dalla riga di comando con tre semplici operazioni:

- 1 Nella casella di ricerca di **Cortana** digitiamo **cmd**.
- 2 Successivamente individuiamo il risultato e col tasto destro del mouse, clicchiamo sulla voce “**Esegui come amministratore**”.
- 3 Chiudiamo con un “**Sì**” la finestra “**Controllo dell'account utente**”:



- 4 Nella nuova finestra “Amministratore: Prompt dei comandi”, digitiamo:  
dism /online /Enable-Feature /FeatureName:TelnetClient

- 5 Premiamo [Invio] per confermare l’installazione del client.



```
C:\WINDOWS\system32>dism /online /Enable-Feature /FeatureName:TelnetClient
Strumento Gestione e manutenzione immagini distribuzione
Versione: 10.0.10240.16384

Versione immagine: 10.0.10240.16384

Attivazione funzionalità
[=====100.0%=====]
Operazione completata.

C:\WINDOWS\system32>
```

- 6 Dopo pochi secondi, come per magia, ecco pronto il nostro Telnet.



```
Telnetarbornet.org...
Microsoft Telnet Client
Il carattere di Escape è 'CTRL++'
Microsoft Telnet> open arbornet.org

FreeBSD/i386 (m-net.arbornet.org) (pts/2)

login: *
```

# ESERCITAZIONI DI LABORATORIO 10

## CLIENT TCP IN C

### ■ Struttura di un client

Nelle lezioni precedenti abbiamo visto come avviene la procedura di comunicazione tra **server** e **client**. Per il **client** parte delle operazioni sono identiche a quelle effettuate sul **server** nelle lezione precedente, come l'inizializzazione del **socket** con la definizione del **transport endpoint** del **client**.

### Operazioni preliminari alla connessione

Nel **client** devono però essere definiti due record di tipo **sockaddr\_in** in quanto uno deve contenere i dati del **client** e l'altro i dati del **server** al quale deve connettersi.

Per prima cosa includiamo le librerie e definiamo le costanti:

```
clientMaiulx1.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7 #include <unistd.h>
8 #include <arpa/inet.h>
9 #include <errno.h>
10
11 #define SOCKET_ERROR ((int)-1)
12 #define SERVER_PORT 1313 // numero di porta del server
13 #define MAXSIZE 10 // dimensione del messaggio
14
```

Successivamente nel **main()** definiamo le variabili di servizio:

```
15 int main(int argc, char *argv[]){
16     char indirizzoServer[] = "127.0.0.1"; // indirizzo del server
17     char messaggio[] = "ciao"; // messaggio da inviare
18     char buffer[MAXSIZE]; // messaggio ricevuto
19     char ch, chMaiu; // singolo carattere inviato e ricevuto
20     int ris, nwrite, len;
21 }
```

Definiamo ora le variabili necessarie per la connessione e ne inseriamo i valori:

```

22 int socketfd;                                // identificatore della socket
23 struct sockaddr_in locale, remoto;           // dati dei socket
24
25 // settaggio del socket locale
26 locale.sin_family = AF_INET;
27 locale.sin_addr.s_addr = htonl(INADDR_ANY);
28 locale.sin_port = htons(0);
29 // assegnazione parametri del server
30 remoto.sin_family = AF_INET;
31 remoto.sin_addr.s_addr = inet_addr(indirizzoServer);
32 remoto.sin_port = htons(SERVER_PORT);

```

Naturalmente i valori di `indirizzoServer` e `SERVER_PORT` devono essere noti al **3** per poter richiedere la connessione al **server**.

Creiamo il **socket** sul **client**:

```

34 // impostazione del transport endpoint
35 printf("creazione del socket()\n");
36 socketfd = socket(AF_INET, SOCK_STREAM, 0);
37 if (socketfd == SOCKET_ERROR){
38     printf ("fallito socket(), errore: %d \"%s\"\n", errno, strerror(errno));
39     return(1);
40 }

```

Effettuiamo quindi le operazioni di **bindig**:

```

41 // Leghiamo l'indirizzo al transport endpoint
42 printf("bind()\n");
43 ris = bind(socketfd, (struct sockaddr*) &locale, sizeof(locale));
44 if (ris == SOCKET_ERROR){
45     printf ("fallito il bind(), errore: %d \"%s\"\n", errno, strerror(errno));
46     fflush(stdout);
47     return(3);
48 }

```

Siamo quindi pronti per effettuare la connessione col **server**.

## Connessione col server: la funzione `connect()`

Mentre il codice del **server** a questo punto del programma si mette in attesa che “un qualche client” lo contatti, il **client** deve effettuare la richiesta di connessione e questa viene eseguita mediante la funzione `connect()`; il **client** si connette al **server** in corrispondenza del **socket** indicato dalla struttura che contiene gli indirizzi del **server remoto**:

```

#include <sys/types.h>
#include <sys/socket.h>
int connect(int socketfd, const struct sockaddr *remoto, sizeof(remoto));

```

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori:

- ▷ **ECONNREFUSED**: non c’è nessuno in ascolto sull’indirizzo remoto;
- ▷ **ETIMEDOUT**: si è avuto timeout durante il tentativo di connessione;
- ▷ **ENETUNREACH**: la rete non è raggiungibile;
- ▷ **EINPROGRESS**: il **socket** è non bloccante e la connessione non può essere eseguita;

- **EALREADY**: il **socket** è non bloccante e un tentativo precedente non si è ancora concluso;
- **EAGAIN**: non ci sono più porte locali libere;
- **EAFNOSUPPORT**: l'indirizzo non ha una famiglia di indirizzi corretta nel relativo campo;
- **EACCES, EPERM**: si è tentato di eseguire una connessione a un indirizzo broadcast senza che il **socket** fosse stato abilitato per il broadcast.

Il **client** del nostro esempio effettua la seguente richiesta:

```

51 // richiesta di connessione al server
52 printf("connect()\n");
53 ris = connect(socketfd, (struct sockaddr*) &remoto, sizeof(remoto));
54 if (ris == SOCKET_ERROR){
55     printf("fallita la connect(), errore: %d \"%s\"\n", errno, strerror(errno));
56     fflush(stdout);
57     return(4);
58 }

```

Se tutto è andato a buon fine, i due programmi procederanno con lo scambio di dati fino alla chiusura della connessione, che viene eseguita mediante la seguente istruzione:

```

74
75 // chiusura socket
76 close(socketfd);
77 return(0);
78 }

```

## ■ Operazioni di lettura e scrittura

Vediamo ora ad esempio il segmento di codice che ci permette di inviare un messaggio al **server**, di riceverlo elaborato dal **server**, e quindi di memorizzarlo nel **buffer**.

Come primo esempio inviamo il messaggio un carattere alla volta; prima di trasmettere un nuovo carattere attendiamo che il precedente ci arrivi convertito in maiuscolo dal **server**:

```

60 // ciclo che invia la stringa (1 char alla volta) al server
61 len = strlen(messaggio);
62 printf("\nstringa lunga: %d \n", len);
63 for (nwrite = 0; nwrite < len; nwrite++){
64     write(socketfd, &(messaggio[nwrite]), 1); // invia un carattere
65     if (read(socketfd, &chMaiu, 1) > 0){ // lettura della risposta
66         printf("carattere ricevuto: %c\n", chMaiu); // stampa singolo carattere
67         buffer[nwrite] = chMaiu;
68     }
69 }

```

Il codice non necessita di ulteriori chiarimenti: l'unica osservazione sta nel fatto che se nel messaggio che stiamo inviando è presente il carattere "x" il server cessa la sua elaborazione dato che questo viene interpretato come "fine trasmissione!".

Il codice completo del **client** è memorizzato nel file **clientMaiu1x1.c**.

Per verificare il funzionamento del programma è necessario modificare il **server** descritto nel precedente laboratorio togliendo la frase di saluto inviata alla riga 65 in quanto verrebbe contata dal client come carattere inviato al **client**: il codice del **server** è memorizzato nel file **serverMaiu1x1.c**.



## Prova adesso!

- Comunicazione client-server
- Funzioni connect()

- 1 Completa il codice descritto inserendo l'invio di un carattere di chiusura "x" da parte del client in modo da far terminare il programma anche nel server una volta che il messaggio è stato completamente trasformato in maiuscolo.
- 2 Modifica **client** e **server** facendo in modo che possa inviare una qualunque parola digitata dall'utente e ne attenda la conversione in maiuscolo.  
Confronta la tua soluzione con quella presente nei file **clientMaiuParola.c** e **serverMaiuParola.c**.
- 3 Modifica il **client** facendo in modo che riceva come parametro sia l'indirizzo del **server** che la porta alla quale connettersi.  
Confronta la tua soluzione con quella presente nel file **clientMaiuParola2.c**.
- 4 Successivamente modifica il programma **serverMaiu1x1.c** in modo che accetti più connessioni contemporaneamente (utilizzando la **fork**).

## Esercizi proposti

- 1 Modifica il client in modo che legga da terminale un carattere digitato dall'utente e lo invii direttamente al server utilizzando la funzione **getche()**, in modo da ottenere una situazione simile a quella riportata in figura:

```

Paolo@PCwin8 ~/ua3vol3
$ gcc -o cms clientMaiuSol.c

Paolo@PCwin8 ~/ua3vol3
$ ./cms
Programma che invia un carattere letto con getche() e lo riceve in maiuscolo
Aperta la connessione col server:

Inserisci il carattere da inviare: a -> A
Inserisci il carattere da inviare: b -> B
Inserisci il carattere da inviare: f -> F
Inserisci il carattere da inviare: x -> X

chiusura socket: termine elaborazione.

```

Confronta la tua soluzione con quella presente nel file **clientMaiuSol.c**.

- 2 Realizzare un **client** (e il corrispondente **server**) con queste specifiche.  
Accetta all'avvio una riga di comando con la seguente sintassi:

```
clienteA.exe [IP] [porta] [messaggio]
```

### ESEMPIO

```
clienteA.exe 139.204.72.5 1313 messaggiodiprova).
```

Si connette a IP+porta .

Legge una **stringa A** di 10 byte trasmessi dal **server** (es.: 0000001313).

Invia una **stringa B** di 100 bytes concatenando il proprio nome, cognome, telefono alla **stringa A**, riempiendo con gli spazi i caratteri mancanti e ricevendo un saluto col proprio cognome e nome con le iniziali in maiuscolo.

**ESEMPIO**

```
invio
0000001313 giuseppe, garibaldi, 333987654
ricezione
Hello 1313, Giuseppe Garibaldi
```

Invia un **intero C** di 4 bytes (header) indicando quanti caratteri N spedirà in futuro.  
N=strlen(messaggio)

**ESEMPIO**

```
int N = strlen(messaggiodiprova)
```

Invia una **stringa D** di N caratteri con la parola presa come parametro.

Legge una **stringa E** di 2 bytes:

se E="OK" il **client** ha avuto successo;

se E="ER" il **client** ha commesso un errore nel protocollo.

Stampa a console la **stringa E**.

Chiude la connessione.

- 3** Realizza un sistema **client-server** che implementa una calcolatrice:

- ▷ il **client** si connette al **server**;
- ▷ il **server** ITERATIVO accetta la connessione e scrive nel file **connessioni.log** i dati del **client** (IP, PORTA e ora della connessione);
- ▷ il **client** invia al **server** 2 interi e l'operazione da effettuare (struttura);
- ▷ il **server** effettua l'operazione e invia il risultato al **client**;
- ▷ il **server** scrive nel file connessioni .log il tipo di operazione effettuata;
- ▷ il **client** stampa a video il risultato ricevuto dal **server** e si pone in attesa di inviare altri dati... quindi esce con ^D;
- ▷ quando il **client** esce, il **server** scrive nel file **connessioni.log** l'ora di uscita del **client**;
- ▷ prima di terminare del tutto, il **server** stampa a video lo storico delle operazioni effettuate fino a quel momento.

- 4** Scrivi un'applicazione **client-server** per raccogliere le iscrizioni a una gara di atletica (per esempio per una maratona): l'utente (**client**) richiede di effettuare l'iscrizione e il **server** gli comunica il numero di pettorina.

Si ipotizzi di avere tre categorie:

- ▷ junior: età inferiore a 16 anni;
- ▷ senior: età superiore ai 60 anni;
- ▷ medium: tutti gli altri.

Il concorrente comunica oltre al suo nome anche la sua età e il numero che gli viene assegnato è un progressivo del tipo J23, S23 oppure M345 a seconda della categoria alla quale viene iscritto.

# ESERCITAZIONI DI LABORATORIO 11

## IL PROTOCOLLO UDP NEL LINGUAGGIO C

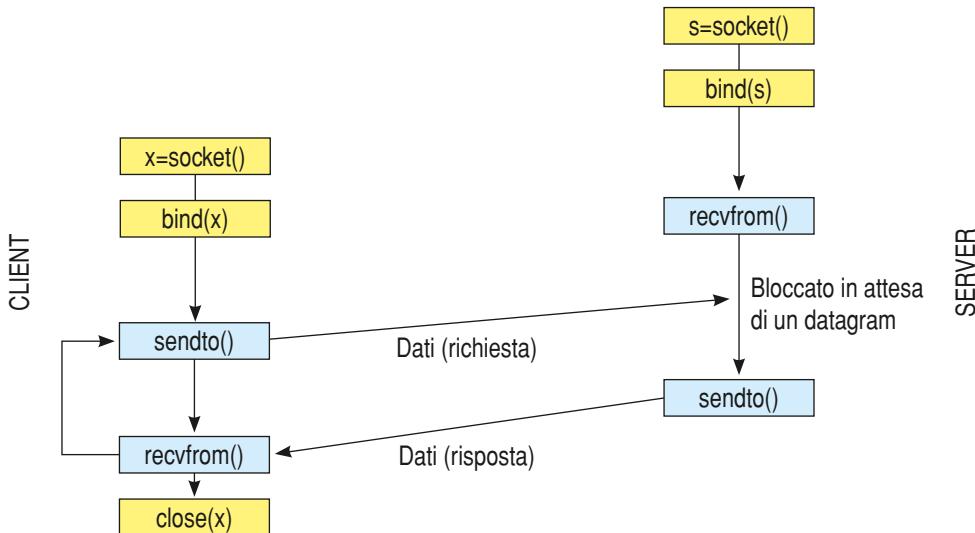
### ■ Struttura del protocollo UDP

Realizziamo una comunicazione mediante protocollo UDP dove un mittente ([inviaUDP.c](#)) trasmette un messaggio ([datagram](#)) a un destinatario ([riceviUDP.c](#)).

Ricordiamo che in una comunicazione dati **connectionless** o **datagram** il canale:

- ▷ trasporta messaggi;
- ▷ non è affidabile;
- ▷ il **socket** è condiviso;
- ▷ non preserva l'ordine delle informazioni.

La struttura di una applicazione UDP è la seguente:



Per leggere o scrivere su un **socket UDP** si utilizzano funzioni di sistema differenti da **TCP** che devono specificare l'indirizzo del **server** e la lunghezza del messaggio; descriviamole nel dettaglio.

### ■ Funzioni `sendto()` e `recvfrom()`

Per usare UDP è necessario utilizzare le funzioni `sendto()` e `recvfrom()` in quanto in UDP non esiste il concetto di connessione e quindi non è possibile usare direttamente `read()` e `write()` non avendo già stabilito quale sia la sorgente e quale la destinazione dei dati.

Le funzioni `sendto()` e `recvfrom()` sono utilizzabili per qualunque tipologia di `socket` e sono le sole utilizzabili da UDP: hanno tre argomenti aggiuntivi attraverso i quali è possibile specificare la destinazione dei dati trasmessi oppure ottenere l'origine dei dati ricevuti.

## ■ Funzione `sendto()`

La prima di queste funzioni, `sendto()`, serve per trasmettere un messaggio a un altro `socket`, e ha il seguente prototipo:

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t sendto(int socketfd, const void *buffer, size_t len, int flags, const
               struct sockaddr *to, socklen_t tolen)
```

ha come parametro di ritorno il numero di byte inviati che deve sempre corrispondere alla dimensione totale specificata dal parametro `len` in quanto i dati non possono essere spezzati in invii successivi ma vengono sempre inviati in forma di pacchetto.

I primi tre argomenti sono identici a quelli della funzione `write()` e definiscono il `socket` `socketfd` a cui si fa riferimento, il buffer `*buffer` che contiene i dati da inviare e la relativa lunghezza `len`. L'argomento `flags` è un intero usato come maschera binaria che permette di impostare una serie di modalità di funzionamento avanzate della comunicazione attraverso il `socket`: nei casi standard si lascia sempre uguale a 0.

I due argomenti `to` e `tolen` servono a specificare la destinazione del messaggio da inviare, e indicano rispettivamente la struttura contenente l'indirizzo di quest'ultima e la sua dimensione.

Questi argomenti vanno specificati nella stessa forma in cui si sarebbero usati con `connect()`: `to` punta alla struttura contenente l'indirizzo IP e la porta di destinazione dei dati da inviare.

La `sendto()` termina dopo aver completato la spedizione e non garantisce che i dati arrivino a destinazione: restituisce il numero di caratteri inviati in caso di successo e -1 per un errore, nel qual caso `errno` viene impostata al rispettivo codice di errore:

- ▶ **EAGAIN**: il `socket` è in modalità non bloccante, ma l'operazione richiede che la si blocchi;
- ▶ **ECONNRESET**: l'altro capo della comunicazione ha resettato la connessione;
- ▶ **EDESTADDRREQ**: il `socket` non è di tipo connesso e non si è specificato un indirizzo di destinazione;
- ▶ **EISCONN**: il `socket` è già connesso, ma si è specificato un destinatario;
- ▶ **EMSGSIZE**: il tipo di `socket` richiede l'invio dei dati in un blocco unico, ma la dimensione del messaggio lo rende impossibile;
- ▶ **ENOBUFS**: la coda di uscita dell'interfaccia è già piena (di norma Linux non usa questo messaggio ma scarta silenziosamente i pacchetti);
- ▶ **ENOTCONN**: il socket non è connesso e non si è specificata una destinazione;
- ▶ **EOPNOTSUPP**: il valore di `flags` non è appropriato per il tipo di socket usato;
- ▶ **EPIPE**: il capo locale della connessione è stato chiuso, si riceverà anche un segnale di `SIGPIPE`, a meno di non aver impostato `MSG_NOSIGNAL` nel parametro `flags`.

## ■ Funzione `recvfrom()`

La funzione utilizzata per ricevere i dati nella comunicazione fra `socket UDP` è `recvfrom()`; il suo prototipo è:

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recvfrom (int socketfd, const void *buffer, size_t len, int flags,
                  const struct sockaddr *from, socklen_t fromlen)
```

I primi tre argomenti sono identici a quelli della funzione di `read()`: dal `socket` vengono letti `len` byte che vengono salvati nel buffer `buffer`.

I due argomenti `from` e `fromlen` permettono di ottenere l'indirizzo del mittente del pacchetto che è stato ricevuto, e devono essere inizializzati con i puntatori alle variabili della struttura dove saranno scritti l'indirizzo IP e la sua lunghezza.

La funzione restituisce il numero di byte ricevuti in caso di successo e `-1` in caso di errore; nel qual caso `errno` assumerà il valore:

- **EAGAIN**, il socket è in modalità non bloccante, ma l'operazione richiede che la funzione si blochi, oppure si è impostato un timeout in ricezione e questo è scaduto;
- **ECONNREFUSED**, l'altro capo della comunicazione ha rifiutato la connessione (in genere perché il relativo servizio non è disponibile);
- **ENOTCONN**, il socket è di tipo connesso, ma non si è eseguita la connessione;
- ...oltre ai soliti **EFAULT**, **EBADF**, **EINVAL**, **EINTR**, **ENOMEM**, **ENOTSOCK** e gli eventuali altri errori relativi ai protocolli utilizzati.

A differenza delle `read()` e `write()` che abbiamo usato con i `socket TCP` in questo caso è possibile inviare con `sendto()` un pacchetto vuoto specificando un valore nullo per `len`: la ricezione con `recvfrom()` di un valore di ritorno di 0 byte viene interpretato come una chiusura della connessione o come una cessazione delle comunicazioni.

## ■ Destinatario UDP

Realizziamo un host che si pone in attesa di un messaggio sulla porta locale 1313: la prima parte del codice è simile a quella descritta nelle esercitazioni precedenti, dove vengono definite le variabili e viene creato il `socket`: una differenza è il parametro utilizzato, cioè `SOCK_DGRAM` mediante il quale si definisce la modalità `UDP`.

A differenza di `Java`, che ha una struttura completamente diversa tra `TCP` e `UDP`, in `C` per passare da un protocollo all'altro basta solo sostituire un parametro nella definizione del `socket`:

- `TCP` viene settato a `SOCK_STREAM`;
- `UDP` viene settato a `SOCK_DGRAM`.

Dopo aver incluso le librerie, aggiungiamo la definizione delle seguenti costanti.

```
riceveUDP.c inviaUDP.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7 #include <arpa/inet.h>
8 #include <errno.h>
9
10 #define SOCKET_ERROR ((int)-1)
11 #define SIZEBUF 10000
12 #define SERVER_PORT 1313 // numero di porta del server
13
```

Il codice del `main()` inizia con la definizione delle variabili:

```

15
14 int main(int argc, char *argv[]){
15     struct sockaddr_in locale, remoto;
16     int socketfd, msglen, len, ris;
17     char msg[SIZEBUF];
18

```

Nella definizione dei parametri dell'host destinatario inizializziamo il campo dell'IP con valore `INADDR_ANY` che indica la disposizione a ricevere/effettuare connessioni qualunque sia l'indirizzo IP della macchina. Come numero di porta si deve invece specificare il numero di porta sulla quale si desidera porsi in ascolto. La configurazione è la seguente:

```

19 /* configurazione parametri porta locale */
20 locale.sin_family      = AF_INET;           // internet
21 locale.sin_addr.s_addr= htonl(INADDR_ANY);   // qualunque IP
22 locale.sin_port         = htons(SERVER_PORT);
23

```

Come al solito creiamo il `socket` indicando come parametro `SOCK_DGRAM`:

```

24 /* creazione del socket */
25 socketfd = socket(AF_INET, SOCK_DGRAM, 0);
26 if (socketfd == SOCKET_ERROR) {
27     printf ("socket() fallito err: %d \"%s\"\n", errno, strerror(errno));
28     exit(1);
29 }
30 printf ("socket() ok\n");
31 fflush(stdout);

```

Effettuiamo ora l'operazione di `bind()`:

```

33     ris = bind(socketfd, (struct sockaddr*) &locale, sizeof(locale));
34 if (ris == SOCKET_ERROR) {
35     printf ("bind() fallita err: %d \"%s\"\n", errno, strerror(errno));
36     exit(1);
37 }
38 printf ("bind() sul server ok, ora mi pongo in attesa:\n");
39 fflush(stdout);
40

```

Quindi il `server` si pone in attesa dell'arrivo di messaggi da parte dei `client`:

```

40
41     /* attesa della ricezione di un datagramma */
42     len = sizeof(struct sockaddr_in);
43     msglen = recvfrom(socketfd, msg, (int)SIZEBUF, 0, (struct sockaddr*)&remoto, &len);
44 if (msglen<0) {
45     char msgerror[1024];
46     sprintf(msgerror,"recvfrom() failed [err %d] ", errno);
47     perror(msgerror);
48 }

```

Man mano che arriva un messaggio questo viene visualizzato con i parametri dell'host mittente:

```

50 // Lettura datagramm e parametri del mittente
51 printf ("ho ricevuto un messaggio: \n");
52 printf("ricevuto msg: \"%s\" lunghezza: %d \n", msg, msglen);
53 // salvataggio utilizzando inet_ntop()
54 char s_IP_Remoto[15];           // indirizzo dell'host remoto
55 short int portaRemota;         // porta usata dall'host remoto
56 inet_ntop(AF_INET, &remoto.sin_addr, s_IP_Remoto, sizeof(s_IP_Remoto));
57 portaRemota = ntohs(remoto.sin_port); // trasforma in formato Big Endian
58 printf("host mittente: %s:%d\n", s_IP_Remoto, portaRemota);

```

Una possibile esecuzione è la seguente:

```
Utente@PC-Win7 ~/ua3vol3
$ ./riceveUDP
socket() ok
bind() sul server ok, ora mi pongo in attesa:
ho ricevuto un messaggio:
ricevuto msg: "ciao" lunghezza: 4
host mittente: 127.0.0.1:3377

Utente@PC-Win7 ~/ua3vol3
```

## Mittente UDP

Il codice del mittente è molto simile a quello del client **TCP** descritto nelle lezioni precedenti; dopo l'inclusione delle librerie definiamo le variabili del **main()**:

```
15
14 [-] int main(int argc, char *argv[]){
15     struct sockaddr_in locale, remoto;
16     char indirizzoServer[]="127.0.0.1";           // indirizzo del server
17     int socketfd, optVal, len;
18     int ris;
19 }
```

Per poter inviare il **datagram** il **client** oltre a definire il proprio **socket** deve specificare l'indirizzo del **server**; viene così inizializzato anche il record che contiene i dati del **server**.

```
20 // configurazione parametri porta locale
21 locale.sin_family      = AF_INET;
22 locale.sin_addr.s_addr = htonl(INADDR_ANY);
23 locale.sin_port         = htons(0);
24
25 // assegnazione parametri del destinatario
26 remoto.sin_family      = AF_INET;
27 remoto.sin_addr.s_addr = inet_addr(indirizzoServer); // indirizzoServer
28 remoto.sin_port         = htons(SERVER_PORT);          // porta remota
29 }
```

Si procede con la creazione del **socket** con l'accortezza di indicare come parametro **SOCK\_DGRAM**:

```
-- 33 // creazione del socket
34     socketfd = socket(AF_INET, SOCK_DGRAM, 0);
35 [-] if (socketfd == SOCKET_ERROR){
36     printf ("socket() fallita, err: %d \"%s\"\n", errno, strerror(errno));
37     exit(1);
38 }
39 printf ("socket() ok \n");
```

Il **socket** viene inizializzato e viene effettuato in **bind()**, segnalando eventuali errori:

```
41
42 // bind del socket
43 ris = bind(socketfd, (struct sockaddr*) &locale, sizeof(locale));
44 [-] if (ris == SOCKET_ERROR){
45     printf ("bind() fallita, err: %d \"%s\"\n", errno, strerror(errno));
46     exit(1);
47 }
48 printf ("bind() ok \n");
```

Ora è possibile inviare un messaggio; nel nostro esempio inviamo l'argomento letto come parametro nella linea di comando all'avvio del client:

```
-- 51 // invia una parola a destinazione
52 len = sizeof(struct sockaddr_in);
53 ris = sendto(socketfd, argv[1], strlen(argv[1]), 0, (struct sockaddr*)&remoto, len);
```

```

54 if (ris < 0) {
55     printf ("errore sendto(): %d \"%s\"\n", errno,strerror(errno));
56     exit(1);
57 }
58 else
59     printf("datagram UDP \"%s\" inviato a %s:%d\n",argv[1],indirizzoServer,SERVER_PORT);
60

```

Una possibile esecuzione è la seguente:

```

Utente@PC-Win7 ~/ua3vol3
$ ./inviaUDP ciao
avvio esecuzione client
socket() ok
bind() ok
datagram UDP "ciao" inviato a 127.0.0.1:1313

Utente@PC-Win7 ~/ua3vol3

```



## Prova adesso!

- 1** Realizza un’“applicazione echo UDP” dove il **server** replica tutti i messaggi inviatigli dal **client**: il **client** legge linee di testo dallo standard input e le invia al **server**:
  - il **server** legge linee di testo dal **socket** e le rimanda al **client**;
  - il **client** legge righe di testo dal **socket** e le invia allo standard output.
 Confronta la tua soluzione con quella presente nel file [inviaUDPSol.c](#).
- 2** Modifica il **client** facendo in modo che riceva come parametro sia l’indirizzo del **server** che quello della porta alla quale connettersi e dopo aver inviato un messaggio si metta in attesa di una risposta.  
Modifica il codice del **server** creando una applicazione che è in ascolto su una porta **UDP**, accetta tutti i datagram ricevuti ma deve scartare quelli che provengono da indirizzi **IP** non noti: ad esempio, vengono accettati i datagrammi che sono trasmessi degli host con un numero dispari nell’ultimo ottetto dell’indirizzo **IP** e a questi si risponde con “OK” mentre agli altri si invia un “KO”.  
Dopo aver accettato tre messaggi, il **server** termina la sua esecuzione.
- 3** Realizza un’applicazione multicast di datagrammi **UDP** dove il **server** invia a un indirizzo multicast e a intervalli di 1 secondo pacchetti **UDP** contenenti ciascuno una riga di un file di testo contenente *La Divina Commedia*; l’indirizzo e la porta di comunicazione vengono letti dalla linea di comando:  
**UDPserver** IPaddress UDPport (ad esempio **UDPserver** 127.0.0.1 6916)  
Il **client** riceve i datagrammi e scrive un file per ogni canto, con la seguente struttura: **cantica\_nrcanto.txt** (ad esempio **inferno\_canto5.txt**).

## AREA digitale



Funzioni **GETSOCKOPT()** e **SETSOCKOPT()**

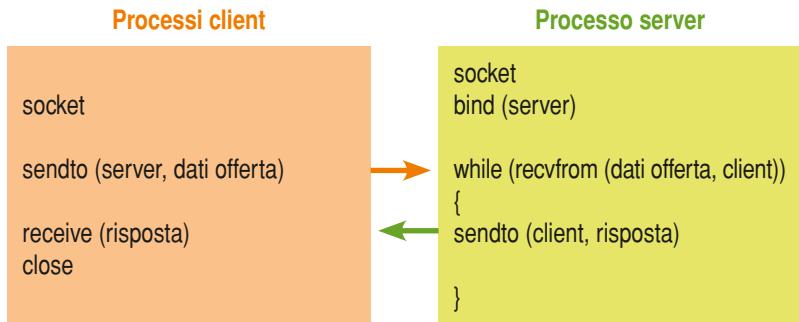
# ESERCITAZIONI DI LABORATORIO 12

## UN ESEMPIO COMPLETO CON I SOCKET: ASTA ONLINE

Scriviamo un'applicazione **client-server UDP** per gestire le offerte pervenute a un'asta: i **client** inviano al **server** l'importo dell'offerta e l'identificativo dell'offerente e il **server** mantiene in memoria solamente l'offerente e l'offerta più alta, rispondendo ai diversi **client** se l'offerta è stata accettata oppure l'importo da superare per potersi aggiudicare l'asta.

La struttura del sistema, composta da un processo **server** e da  $n$  processi **client**, è la seguente:

- A il processo **server**, dopo aver inizializzato il sistema, si pone in attesa delle offerte costituite da due dati (nome offerente e importo offerta) e a ogni nuova ricezione analizza i dati, controllando se l'importo è superiore alla migliore offerta sino ad allora pervenuta, comunicando l'esito al nuovo offerente; l'asta termina per esempio dopo che sono pervenute 10 offerte consecutive;
- B i singoli **client** effettuano una offerta e si pongono in attesa dell'esito: quindi terminano la loro esecuzione.



### Codifica del server

Nella prima sezione del codice, oltre alle librerie, vengono definite alcune costanti che non necessitano di spiegazioni particolari:

<code>astaServer.c</code>	<code>astaClient.c</code>
---------------------------	---------------------------

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7 #include <netdb.h>
8 #include <arpa/inet.h>
9 #include <unistd.h>
10 #include <errno.h>
11
12 #define MAXOFFERTE 9           // numero massimo di offerte per sessione
13 #define MAXLUNGO 40           // lunghezza nominativo offerente
14 #define FILENAME "asta.txt"   // file con i dati del miglior offerente
15 #define IPADDRESS "127.0.0.1" // indirizzo del server
16 #define PORTA 1313            // numero di porta del server

```

Per la realizzazione del **server** definiremo tre funzioni:

```

18 struct sockaddr_in offerente;
19 int socketlen;
20
21 // creazione della socket sul server
22 int creaServer (char *ip_address, int porta){
23     ...
24     return socketfd;
25 }
26
27 // funzione che trasmette un messaggio al client
28 int inviaMsg (int socketfd, char *buffer){
29     ...
30     return esito;           // 1 = ok, 0 = errore
31 }
32
33 // funzione che riceve un messaggio dal client
34 int riceviMsg (int socketfd, char *buffer){
35     ...
36     return numCaratteri;
37 }
```

La prima funzione crea il **socket**, la seconda invia un messaggio che riceve come parametro nel buffer e la terza rimane in attesa di una comunicazione da parte di un **client**.

La codifica viene lasciata come esercizio al lettore: una soluzione è presente nel file **astaServer.c** contenuto in **ua3vol3 c.rar** prelevabile dal **DVD** allegato, cartella **source\UA3**, oppure scaricabile dal sito [www.hoeplicsuola.it](http://www.hoeplicsuola.it) nella cartella **materiali**.

Il programma principale può essere organizzato in tre sezioni:

```

39 // programma principale
40 int main (int argc, char *argv[]){
41     // definizione delle variabili locali
42     ...
43     // 1 - creazione del socket sul server
44     ...
45     // 2 - recupero dati da file
46     ...
47     // 3 - ricezione di una offerta
48     ...
49     // chiusura del socket
50     ...
51 }
```

La prima parte di codice è la seguente:

```

72 // programma principale
73 int main (int argc, char *argv[]){
74     int socketfd, numOfferte;           // definizione delle variabili locali
75     char buffer [BUFSIZ + 1];
76     char nomeOfferente [MAXLUNGO], nomeVince [MAXLUNGO];
77     int nuovaOfferta, offertaVince;
78     FILE *fileOfferte;
79
80     numOfferte = 0;
81 }
```

Sul file **fileOfferte** vengono memorizzati solo i dati del miglior offerente, e in particolare solo il nome e l'importo dell'offerta, che vengono successivamente letti e inseriti rispettivamente nelle variabili **nomeVince** e **offertaVince**.

I dati della nuova offerta pervenuta vengono temporaneamente memorizzati in `nomeOfferente` e `nuovaOfferta` e quindi vengono elaborati.

La creazione del `socket` sul `server` avviene semplicemente con la chiamata alla funzione prima definita:

```

82 // creazione della socket sul server
83 if ((socketfd = creaServer (IPADDRESS, PORTA)) == -1){
84     fprintf (stderr, "impossibile creare la socket sul server\n");
85     exit (EXIT_FAILURE);
86 }
```

Viene ora letto dal file `fileOfferte` il valore della offerta vincente, se presente, altrimenti le rispettive variabili vengono azzerate:

```

88 // recupero dati da file
89 if ((fileOfferte = fopen (FILENAME, "r" )) == NULL){
90     nomeVince [0] = '\0';
91     offertaVince = 0;
92     printf ("L'asta e' aperta: nessuna offerta pervenuta\n");
93 }
94 else{
95     fscanf (fileOfferte, "%s %d", nomeVince, &offertaVince);
96     fclose (fileOfferte);
97     printf ("L'asta e' aperta: fate le vostre offerte\n");
98     printf ("Migliore offerta pervenuta: %s %d\n", nomeVince, offertaVince);
99 }
```

Mediante `riceviMsg()` il `server` si mette ora in attesa di ricevere una nuova offerta da parte di un `client`:

```

101 // ricezione di una offerta
102 while (riceviMsg (socketfd, buffer) > 0){
103     nuovaOfferta = 0;
104     sscanf (buffer, "%s %d", nomeOfferente, &nuovaOfferta);
105     if (nuovaOfferta <= 0)
106         inviaMsg (socketfd, "Offerta non valida\n");
107     else{
108         numOfferte++;
109         printf ("Nuova offerta # %d di Euro %d\n", numOfferte, nuovaOfferta);
110         if (nuovaOfferta > offertaVince) { // la nuova offerta diviene vincente
111             < aggiorna il file con la nuova offerta ricevuta >
112         }
113     }
114     else{ // l'offerta e' inferiore
115         inviaMsg (socketfd, "Spiacente, e' stata fatta un'offerta superiore\n");
116     }
117 }
118 if (numOfferte>MAXOFFERTE){ // l'asta e' terminata
119     printf ("Termine asta : si aggiudica il sig. %s con l'offerta di Euro %d \n",
120            nomeVince, offertaVince);
121     exit (0);
122 }
123 }
```

Quando gli perviene un messaggio, lo analizza:

- A** se l'importo è minore di zero, l'offerta è errata, e segnala questa situazione al mittente;
- B** se l'importo è corretto:
  - se è superiore alla offerta che sta “vincendo” deve essere aggiornato il file con il nuovo importo e il nuovo offerente;
  - se è inferiore alla offerta che sta “vincendo” viene solamente inviato un messaggio al `client`.

Il codice che aggiorna i dati è il seguente:

```

110 if (nuovaOfferta > offertaVince) { // la nuova offerta diviene vincente
111     offertaVince = nuovaOfferta;
112     strcpy (nomeVince, nomeOfferente);
113     if ((fileOfferte = fopen (FILENAME, "w" )) == NULL) { // aggiorna il file
114         inviaMsg (socketfd, "Errore di apertura del file\n");
115         printf ("errore di apertura del file\n");
116     }
117     else{
118         fprintf (fileOfferte, "%s %d", nomeVince, offertaVince);
119         fclose (fileOfferte);
120     }
121     inviaMsg (socketfd, "Offerta accettata\n");
122     printf ("Nuova offerta da superare: %s %d\n", nomeVince, offertaVince);
123 }
124 else{ // l'offerta è inferiore

```

Per ogni esecuzione del server vengono accettate solo 10 offerte: se si supera tale valore l'asta si considera terminata, viene indicato il vincitore e si procede con la chiusura del **socket** e con la terminazione del programma.

```

136 // chiusura del socket
137 if (close (socketfd) != 0) {
138     printf ("errore nella chiusura della socket");
139     return 0;
140 }
141
142 return EXIT_SUCCESS;
143 }

```

Una possibile esecuzione è la seguente:

```

Utente@PC-Win7 ~/ua3vol3
$ ./astaServer
L'asta e' aperta: fate le vostre offerte
Migliore offerta pervenuta: ciro 6000
Nuova offerta # 1 di Euro 7000
Nuova offerta da superare: mario 7000
Nuova offerta # 2 di Euro 6600
Nuova offerta # 3 di Euro 7200
Nuova offerta da superare: paolo 7200
Nuova offerta # 4 di Euro 7200
Nuova offerta # 5 di Euro 7200
Nuova offerta # 6 di Euro 7400
Nuova offerta da superare: piero 7400
Nuova offerta # 7 di Euro 7500
Nuova offerta da superare: paolo 7500
Nuova offerta # 8 di Euro 7700
Nuova offerta da superare: mauro 7700
Nuova offerta # 9 di Euro 8000
Nuova offerta da superare: paolo 8000
Nuova offerta # 10 di Euro 8200
Nuova offerta da superare: mario 8200
Terminata asta : si aggiudica il sig. mario con l'offerta di Euro 8200

```

## ■ Codifica del client

Come per il **server** definiamo due funzioni, la prima per ricevere dati dal **server** e la seconda per trasmettere la nostra offerta:

La codifica viene lasciata come esercizio al lettore: una soluzione è presente nel file **astaClient.c** contenuto in **ua3vol3 c.rar** scaricabile dal sito [www.hoepliscuola.it](http://www.hoepliscuola.it) nella cartella materiali.

```

15 struct sockaddr_in client;
16 int socketlen;
17
18 // funzione che riceve un messaggio al server
19 int riceviMsg (int socketfd, char *buffer){
20     ...
21     return numCaratteri;
22 }
23
24 // funzione che trasmette un messaggio al server
25 int inviaMsg (int socketfd, char *buffer, char *ip_address, int porta){
26     ...
27     return esito;           // 1 = ok, 0 = errore
28 }

```

Il programma principale è abbastanza semplice; riceve in ingresso come argomento l'indirizzo IP del **server** al quale inviare l'offerta:

```

47
48 int main (int argc, char *argv[]){
49     // variabili locali
50     char buffer[BUFSIZ + 1];
51     int socketfd;
52
53 if (argc != 2){                  // IP del server come parametro
54     fprintf (stderr, "richiesto un argomento: indirizzo IP del server\n");
55     exit (EXIT_FAILURE);
56 }

```

e quindi definisce il **socket** del **client**:

```

57
58     // creazione della socket del client
59     if ((socketfd = socket (AF_INET, SOCK_DGRAM, 0)) < 0){
60         printf ("errore fatale nella creazione della socket()");
61         return 0;
62     }

```

L'utente può ora inserire il nome e l'offerta che viene inoltrata al **server** che provvederà a rispondere con un messaggio che contiene l'esito dell'operazione, cioè se si è superata o meno l'offerta degli altri competitori:

```

63
64     // creazione della offerta
65     printf ("Inserire nome e offerta: ");
66     if (fgets (buffer, BUFSIZ, stdin) != NULL){
67         inviaMsg (socketfd, buffer, argv[1], PORTA);      // inoltro messaggio
68         if (riceviMsg (socketfd, buffer) > 0)            // attesa risposta
69             fputs (buffer, stdout);
70     }

```

Il codice del **client** termina con la chiusura del **socket**:

```

71
72     // chiusura socket del client
73     if (close (socketfd) != 0){
74         printf ("errore fatale nella chiusura della socket()");
75         return 0;
76     }
77
78     return EXIT_SUCCESS;
79 }

```

Una esecuzione del **client** è riportata nella figura seguente:

```
Inserire nome e offerta: paolo 6600
Spiacente, e' stata fatta un'offerta superiore

Utente@PC-Win7 ~/ua3vol3
$ ./astaClient 127.0.0.1
Inserire nome e offerta: paolo 7200
Offerta accettata

Utente@PC-Win7 ~/ua3vol3
$ ./astaClient 127.0.0.1
Inserire nome e offerta: mario 7200
Spiacente, e' stata fatta un'offerta superiore

Utente@PC-Win7 ~/ua3vol3
$ ./astaClient 127.0.0.1 piero 7300
richiesto un argomento: indirizzo IP del server

Utente@PC-Win7 ~/ua3vol3
$ ./astaClient 127.0.0.1
Inserire nome e offerta: piero 7200
Spiacente, e' stata fatta un'offerta superiore

Utente@PC-Win7 ~/ua3vol3
$ ./astaClient 127.0.0.1
Inserire nome e offerta: piero 7400
Offerta accettata

Utente@PC-Win7 ~/ua3vol3
$ ./astaClient 127.0.0.1
Inserire nome e offerta: paolo 7500
Offerta accettata

Utente@PC-Win7 ~/ua3vol3
$ ./astaClient 127.0.0.1
Inserire nome e offerta: mauro 7700
Offerta accettata
```



## Prova adesso!

- 1** Modifica il sistema facendo in modo che, nel caso in cui una nuova offerta superi il valore della migliore offerta presente, il nuovo valore massimo venga comunicato a tutti i partecipanti attivi, cioè coloro che abbiano effettuato almeno un'offerta.
- 2** Modifica il sistema precedente offrendo la possibilità di effettuare un'offerta per un oggetto scelto da un elenco dove sono presenti diversi articoli, ciascuno identificato tramite un proprio codice identificatore di lunghezza 8 caratteri (per esempio ABC12345): il sistema deve controllare che il codice corrisponda effettivamente a un articolo posto all'asta, verificare il valore dell'offerta e comunicare l'esito sia al nuovo offerente che al precedente detentore dell'offerta massima nel caso in cui questa venga superata.
- 3** Completa il sistema precedente aggiungendo una scadenza temporale per ciascun oggetto messo all'asta: quando si avvia al termine dell'asta si deve comunicare a tutti i partecipanti l'esito e richiedere il pagamento al vincitore.

# 4

# Applicazioni lato server in Java: servlet

L1 Le servlet

L2 JDBC: Java DataBase Connectivity

## Esercitazioni di laboratorio

- 1 XAMPP e il server engine Tomcat; 2 L'inizializzazione delle servlet;
- 3 L'interazione tra client e servlet get/post con le servlet;
- 4 La permanenza dei dati con le servlet: i cookie; 5 La permanenza dei dati con le servlet: le sessioni; 6 JDBC e MySQL; 7 Servlet e database MDB con parametri

### Conoscenze

- Acquisire le caratteristiche delle servlet
- Conoscere il ciclo di vita di una servlet
- Conoscere le caratteristiche di web.xml
- Acquisire le caratteristiche dell'interfaccia JDBC
- Conoscere i tipi di driver per la connessione ai database

### Competenze

- Installare e utilizzare XAMP
- Realizzare una applicazione web
- Riconoscere i componenti di una pagina lato server
- Generare un file .WAR

### Abilità

- Realizzare un'applicazione WEB dinamica con servlet
- Utilizzare cookie e sessioni con le servlet
- Connettere le applicazioni web con MySQL e Access
- Scrivere, installare e configurare una servlet

## AREA *digitale*



Esercizi



Schema UML della gerarchia delle classi

Installazione della libreria UCanAccess

Installazione di Tomcat su Apache

Netbeans, Eclipse e il file web.xml

Metodi principali classe Cookie

Elenco metodi dell'oggetto HttpSession

Codifica del Session ID nei link

Configurazione manuale di JDBC

Registrazione database in Windows mediante ODBC

Installazione librerie UCanAccess



Esempi proposti

Consulta il DVD in allegato al volume



Soluzioni

Puoi scaricare il file anche da hoepliScuola.it

# Le servlet

In questa lezione impareremo...

- ▶ le caratteristiche delle servlet
- ▶ il ciclo di vita di una servlet
- ▶ a scrivere, installare e configurare una servlet

## ■ Servlet e CGI

L'avvento di **Internet** ha reso possibile lo sviluppo di applicazioni che permettono ad utenti connessi da **host** localizzati in ogni parte del globo di accedere a informazioni presenti su **server** remoti.

### ESEMPIO

Tramite pagine **HTML** e mediante un semplice pulsante **submit** l'utente compila i campi presenti in una **form** e li invia a un **server** che li elabora e risponde inviandogli l'esito della computazione.

Le applicazioni remote sfruttano sostanzialmente due tecniche:

- ▶ **Common Gateway Interface (CGI);**
- ▶ **SERVLET.**

### CGI

Una prima tecnica per realizzare questa modalità operativa è quella di invocare **procedure remote** indicando cioè l'indirizzo di un programma **Common Gateway Interface (CGI)** che viene eseguito sul **server** e al quale vengono passati i dati letti nella **form**.

Il programma **CGI** è scritto in un qualunque linguaggio di programmazione che può leggere da standard **input** e scrivere su standard **output**, come **Perl**, **Python**, **C**, **C++**.

In questa situazione il **web server** ha il solo compito di invocare il programma **CGI** corretto che esegue tutte le elaborazioni utilizzando gli stream standard per input e output.

Per comunicare con l'utente il programma **CGI** produce una pagina **HTML**, la passa al **web server** che la invia all'utente in modo che possa essere visualizzata dal suo browser.

## Servlet

Un'alternativa di **CGI** può essere il linguaggio **Java** per realizzare le **servlet**, che sono delle classi particolari con molte affinità alle **applet**.

- La principale differenza fra **servlet** e **CGI** è che gli script **CGI** vengono eseguiti dal sistema operativo e quindi sono potenzialmente meno portabili delle **servlet** che, essendo scritte in **Java**, vengono eseguite dalla **JVM** integrata nel **web server**: tutti i moderni **web server** hanno integrata la **JVM** e, quindi, le **servlet** sono di fatto portatili su tutte le macchine.
- Una seconda differenza è legata all'efficienza: gli script **CGI** vengono caricati ed eseguiti una volta per ogni richiesta, quindi richiedendo sempre un notevole tempo di latenza, mentre le **servlet** vengono caricate solo una volta e viene creato un **thread** per ogni richiesta (una **servlet** una volta caricata rimane in memoria e può ottimizzare l'accesso alle risorse attraverso caching, pooling ecc.).
- A vantaggio degli script **CGI** c'è la possibilità di utilizzare pressoché qualsiasi linguaggio e quindi il programmatore può scegliere volta per volta il linguaggio più adatto alla particolare applicazione o compito che deve realizzare mentre le **servlet** sono classi **Java** e quindi devono essere scritte in **Java**, con tutti i vantaggi (platform independence, OO, GC ecc. e gli svantaggi (bytecode interpretato) intrinseci di questo linguaggio.

Le **servlet** possono essere utilizzate qualunque sia il servizio espletato dal **server**, ovvero qualunque sia il protocollo di interazione **client-server** (per esempio sia **HTTP** che **FTP**), e facendo parte integrante dello stesso processo del **web server** si può immaginare che sia una procedura dello stesso **server**; questo perché il **web server**, dopo aver effettuato un caricamento dinamico della **classe Servlet** alla prima richiesta alla **servlet** da parte di un client, la mantiene residente in memoria a disposizione delle successive richieste.

## ■ Struttura di una servlet

La **servlet** è una classe "speciale", così definita direttamente dalla **Sun**:



### SERVLET

Una **servlet** è un componente software scritto in **Java**, gestito da un "container", che produce contenuto web dinamico (**Java Servlet Specification, v. 2.4**).

Ogni istanza di una **servlet** è un oggetto **Java** che viene caricato ed eseguito dal **web server** all'interno del processo di richiesta/risposta di servizi. Il suo controllo viene effettuato da un'altra applicazione **Java**, il **container**, che regola il ciclo di vita della **servlet** stessa. In realtà è il **web server** stesso che svolge la funzione di **container** occupandosi della gestione del ciclo di vita delle **servlet**, passandogli i dati del **client** e restituendo ai **client** i dati prodotti dall'esecuzione delle **servlet**.

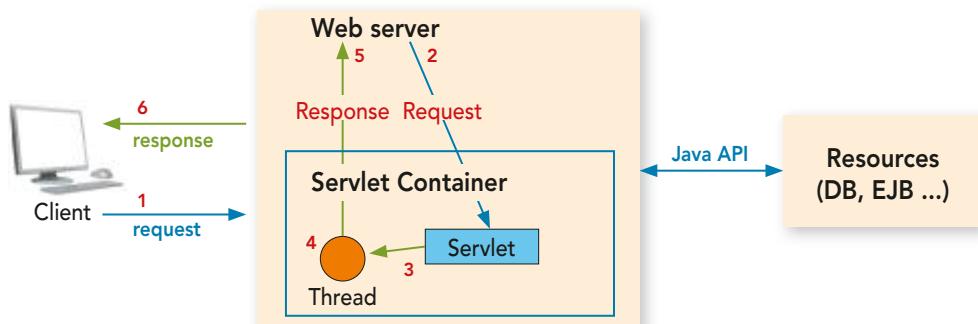
Dato che la **servlet** viene caricata in memoria alla sua prima richiesta di esecuzione, le inizializzazioni necessarie vengono eseguite una volta sola e consentono una più semplice condivisione dei dati fra le istanze di una **servlet**.

## La gestione delle richieste del client: il web container

L'ambiente di esecuzione di una **servlet** si è detto che prende il nome di **container** e, come vedremo, sarà anche l'ambiente di esecuzione delle **JSP (Java Server Page)** trattate in alcune prossime lezioni. Il **container** gestisce completamente la **servlet** durante tutto il suo ciclo di vita svineolando il programmatore dalla loro gestione: gestisce la comunicazione con i **client**, la sicurezza, il multithreading oltre alla sua attivazione e alla sua terminazione.

Questo fa sì che lo sviluppatore si possa concentrare esclusivamente sullo sviluppo della logica dell'applicazione.

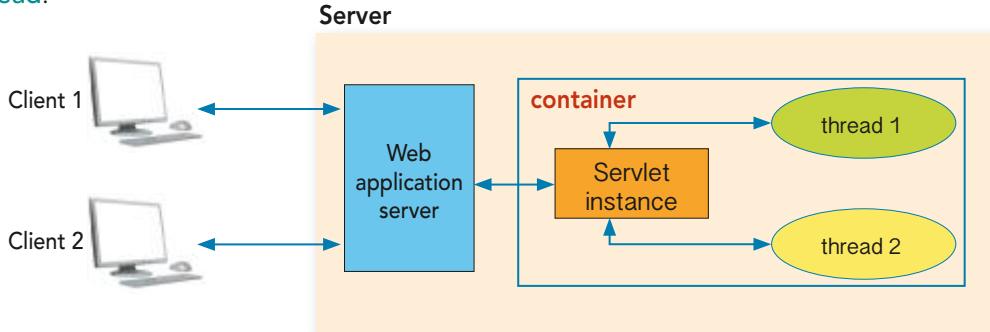
Una **Servlet** interagisce con un **web client** attraverso il paradigma di comunicazione **request/response** schematizzato nella figura seguente:



Descriviamo sinteticamente il flusso di esecuzione:

- ① un **client** invia una richiesta (**request**) per una **Servlet** a un **web application server**;
- ② se si tratta di una prima richiesta allora il **server** istanzia e carica la **Servlet**;
- ③ si attiva un **thread** che gestisce la comunicazione tra il **client** e la **Servlet** stessa;
- ④ il **server** invia al **thread-Servlet** la richiesta pervenutagli dal **client**;
- ⑤ il **thread-Servlet** costruisce e imposta la risposta (**response**) e la inoltra al **server**;
- ⑥ il **server** invia la risposta al **client**.

Nel caso si verifichi una richiesta contemporanea di più **client** vengono attivati nel **Servlet container** più **thread**:



Generalmente le applicazioni web oltre a eseguire semplici algoritmi contenuti all'interno della **Servlet** per generare la risposta al **client** interagiscono con i **Data Base** oppure con altre **Servlet**.



Il **Servlet container** open source più diffuso e utilizzato è **Tomcat**, del “gruppo” **Apache** (del quale riportiamo il logo a lato).

**Tomcat** è un **Servlet container Open Source** e **Free Software** interamente scritto in **Java** disponibile per diverse piattaforme che si integra con i più diffusi **web server**, come **Apache** e **IIS**.

È scaricabile direttamente da: <http://tomcat.apache.org/>: la sua installazione e configurazione sarà descritta nella lezione di laboratorio 1 a esso appositamente dedicata.

L'esecuzione delle **servlet** richiede inoltre che sulla macchina **server** sia installata una versione di **Java** (anche semplicemente la **standard edition**).

## Realizzazione di una servlet

Esistono diverse tipologie di **servlet**, che ereditano da un supertipo presente nel package **javax.servlet**: la loro estensione dipende dal protocollo di comunicazione utilizzato. Le **servlet HTTP** sono il tipo più comune di **servlet** e possono processare richieste (**requestHTTP**) producendo risposte (**responseHTTP**).

Il codice con l'intestazione di una generica **servlet** ha la seguente struttura:



```

EsempioTesto - ua4 servlet
Class Edit Tools Options
Compile Undo Cut Copy Paste Find... Close
Source Code
1 import java.io.*;
2 import javax.servlet.*;
3 import javax.servlet.http.*;

```

Quindi le **servlet** sono in relazione con classi e interfacce contenute nei **package**:

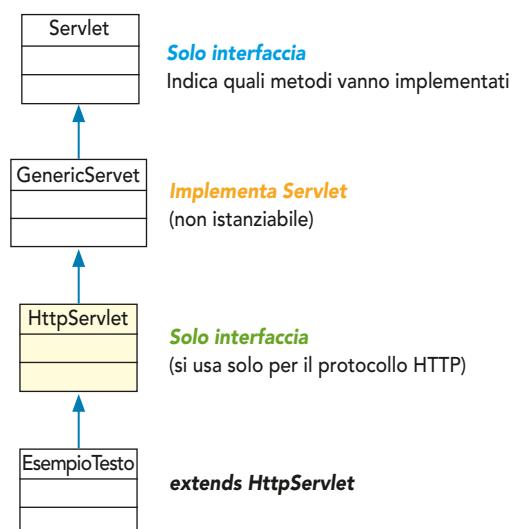
- ▷ **javax.servlet** // indipendenti dal protocollo di comunicazione
- ▷ **javax.servlet.http** // specifiche per il protocollo http

Il **package javax.servlet** è la base per le **API Servlet** e contiene la definizione dell'interfaccia **servlet** e le classi utili alla comunicazione fra **client** e **server**, a partire dai prototipi di tutti i metodi necessari alla gestione del ciclo di vita di una **servlet** e alla esecuzione delle operazioni implementate dalla **servlet**.

Il **package javax.servlet.http** fornisce classi che estendono le funzionalità di base di una **servlet** adottando le caratteristiche del protocollo **HTTP** come la gestione dei metodi **GET** e **POST** o degli header **HTTP**.

Tutte le **servlet** estendono **HttpServlet** o un'altra classe e sono quindi classi che implementano l'interfaccia **servlet**: la gerarchia completa è rappresentata in figura.

Naturalmente è necessario che la libreria **javax** (file **javax.servlet.jar**) sia installata sulla macchina e sia configurata correttamente la sua visibilità all'interno delle applicazioni **Java**: se non venisse automaticamente installata con **Tomcat** è possibile scaricarla dal sito [www.hoepiScuola.it](http://www.hoepiScuola.it), nella cartella **materiali** della sezione riservata a questo libro e copiarla nelle librerie di **Java**.

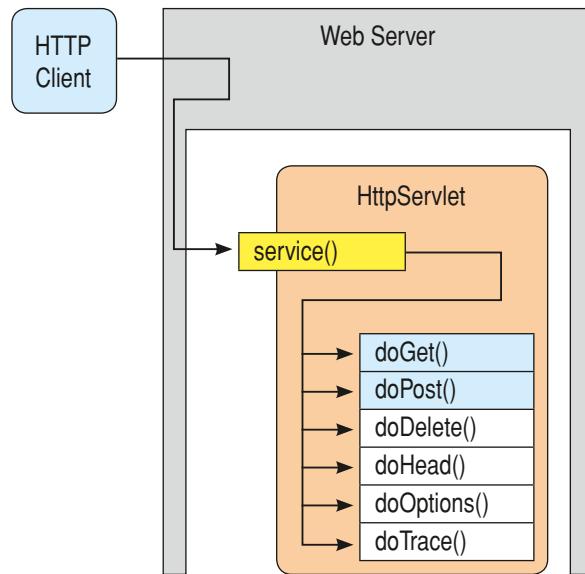


## AREA digitale

 Schema UML della gerarchia delle classi

## ■ La classe HttpServlet

La classe **HttpServlet** implementa **service()** e invoca un insieme di metodi, tra i quali quelli necessari per servire le richieste dal web. Il metodo **service()** viene invocato ogni volta che un client accede a una servlet ed è in grado di interpretare il tipo di richiesta **HTTP** e invocare il metodo opportuno. Per le applicazioni che utilizzano il protocollo **HTTP** non è necessario sovrascrivere il metodo **service()**. La classe che stiamo scrivendo per la nostra applicazione eredita quindi tale metodo definito all'interno della classe **HttpServlet** oltre che i metodi mostrati in figura:



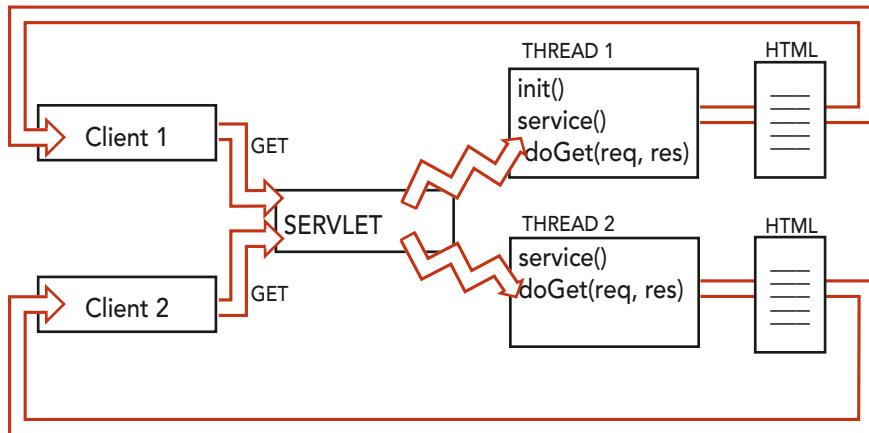
I due metodi principali sono:

- **doGet()**: processa le richieste di tipo **GET** effettuate dalla pagina **HTML**;
- **doPost()**: processa le richieste di tipo **POST** effettuate dalla pagina **HTML**.

Questi due metodi vengono automaticamente chiamati in corrispondenza delle rispettive richieste effettuate dalla pagina **HTML**.

### ESEMPIO

Se **service()** riconosce una richiesta **GET** viene richiamato automaticamente **doGet()** passandogli due oggetti (ad esempio **req** e **res**), rispettivamente il primo del tipo **HttpServletRequest**, che contiene i dati passati dal **client** e **HttpServletResponse** nel quale verranno “inserite” le risposte per il **client**.



## HttpServletRequest

Un oggetto `HttpServletRequest` estende `ServletRequest`, viene passato da `service()` e contiene la richiesta del `client`: permette di ottenere i parametri inviati dal `client`, il riferimento alla sessione utente e il flusso dei dati inviati dal `client` oltre che riconoscere l'utente autenticato.

I metodi più utilizzati su questi oggetti sono:

- ▷ `getRequestURI()` che restituisce l'`URI` richiesto;
- ▷ `getMethod()` che fornisce il metodo `HTTP` utilizzato per ri-inoltrare la richiesta;
- ▷ il metodo `getParameter(String)` che consente di accedere per nome ai parametri contenuti nella query string del client;
- ▷ `getInputStream()` e `getReader()` che permettono di creare gli stream di input e la possibilità quindi di ricevere i dati della richiesta `HTTP`.

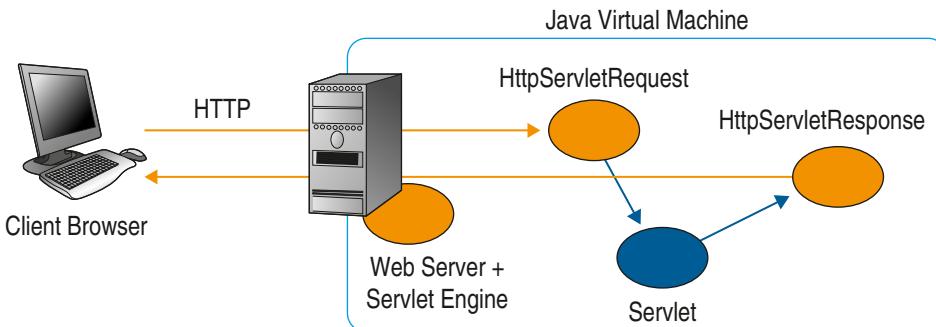
## HttpServletResponse

Un oggetto `HttpServletResponse` estende `ServletResponse`, viene passato da `service()` e contiene la risposta per il `client`: permette di inviare dati al `client` in formato `HTML` oppure come flusso binario, di inviare codici di errore e codici di controllo nell'intestazione della response `HTTP` per controllare il comportamento del browser.

I metodi per creare gli stream di output e la possibilità quindi di inviare i dati della risposta sono:

- ▷ `getOutputStream()` per l'invio di dati in forma binaria;
- ▷ `getWriter()` per l'invio attraverso il canale `System.out`.

Inoltre la classe `HttpServletResponse` fornisce dei metodi per accedere e fissare gli header della risposta, come il `setContentType()`, che deve essere definito prima di poter inviare il body al `client`.



### ESEMPIO

Iniziamo a scrivere una `servlet` di esempio dove utilizziamo `doGet()` ipotizzando che il `client` invii i dati dalla form `HTML` mediante `GET`: lo completiamo con i suoi due parametri e il “rilancio” delle eventuali eccezioni `IOException` e `ServletException`:

```

19 public void doGet(HttpServletRequest req, HttpServletResponse res)
20     throws IOException, ServletException
  
```

## ■ Ciclo di vita di una servlet

Prima di procedere con il codice della `servlet` è necessario analizzare il ciclo di vita di una `servlet` stessa, che può essere riassunto in tre fasi: `init() → service() → destroy()`.

### init()

Al momento del caricamento della `servlet` in memoria il `container` delle `servlet` invoca il metodo `init()` che provvede all'inizializzazione della `servlet` e al settaggio delle sue variabili globali in modo da renderla disponibile per rispondere alla prima richiesta.

Il metodo `init()` riceve come parametro un oggetto di tipo `ServletConfig` che contiene la configurazione iniziale di una `servlet` e per salvare la configurazione richiama il metodo `super.init()`.

Il metodo `init()` viene chiamato una sola volta, all'atto del caricamento che avviene alla prima chiamata: se la `servlet` non riesce a completare l'`inizializzazione` viene generata un'eccezione.

Nel nostro primo esempio non riscriviamo il metodo `init()` ma vedremo come utilizzarlo in seguito.

### service()

Le funzionalità della `servlet`, come ricevere e rispondere alle richieste dei `client`, sono effettuate dal metodo `service()` (o dai suoi sostituti nel protocollo `HTTP doGet()` e `doPost()`), che gestiscono le richieste mediante i due oggetti `ServletRequest` e `ServletResponse` prima descritti.

### destroy()

La terminazione di una `servlet` avviene tipicamente solo quando termina l'esecuzione del web server, che si occupa anche di rilasciare le risorse occupate: questa operazione avviene tramite la chiamata al metodo `destroy()`.

Può anche essere utilizzato per salvare tutte quelle informazioni che possono essere utili al successivo riavvio della `servlet` stessa.

## ■ Output sul client

Nei nostri esempi invieremo sempre ai `client` come risposta una pagina `HTML` utilizzando il metodo `getWriter()`: prima di utilizzarlo è però necessario settare nell'oggetto `response` il tipo `MIME` della risposta con il metodo `setContentType()` e solo successivamente possiamo creare un oggetto `PrintWriter` sul quale scrivere i tag `HTML` da inviare al `client`:

```
12     res.setContentType("text/html");           // content type
13     PrintWriter output = res.getWriter();    // get writer
```

Sull'oggetto `PrintWriter` creiamo la pagina `HTML` da inviare al `client`: creiamo l'oggetto `out` di comunicazione dopo aver indicato la tipologia con il metodo:

```
14     // crea una pagina HTML e la invia al client
15     output.println("<html>");
16     output.println("<head>");
17     output.println("<title>Primo esempio di servlet </title>");
18     output.println("</head>");
19     output.println("<body bgcolor=\"white\">");
20     output.println("<h1>Buongiorno, questa è la prima servlet!</h1>");
21     output.println("</body>");
22     output.println("</html>");
```

La pagina `HTML` può essere creata anche definendo un oggetto buffer del tipo `StringBuffer` e "aggiungendo" su di esso le diverse righe per poi visualizzarlo sull'oggetto `PrintWriter`:

```
5  public class EsempioBuffer extends HttpServlet{
6      public void doGet(HttpServletRequest reqt, HttpServletResponse res)
7          throws IOException, ServletException {
8          res.setContentType("text/html");           // content type
9          PrintWriter output = res.getWriter();    // get writer
10         // crea una pagina HTML e la invia al client
11         StringBuffer buffer = new StringBuffer();
12         buffer.append( "<HTML><HEAD><TITLE>\n" );
13         buffer.append( "Secondo esempio di servlet\n" );
```

```

14     buffer.append( "</TITLE></HEAD><BODY bgcolor=\"white\">\n" );
15     buffer.append( "<h1>Buongiorno, questa è la seconda servlet!</h1>" );
16     buffer.append( "</BODY></HTML>" );
17     output.println( buffer.toString() );
18     output.close(); // chiusura del PrintWriter stream
19 }
20 }
```

I codici di questi due esempi sono nei file **EsempioTesto.java** ed **EsempioBuffer.java**.

A questo punto compiliamo la classe e procediamo con le operazioni successive di configurazione necessarie per poterla eseguire su di un **server**: questa operazione prende il nome di **deployment**.

## ■ Deployment di un'applicazione web

Per poter far funzionare il nostro esempio è necessario effettuare tutti i passaggi previsti dal **deployment**, e cioè:

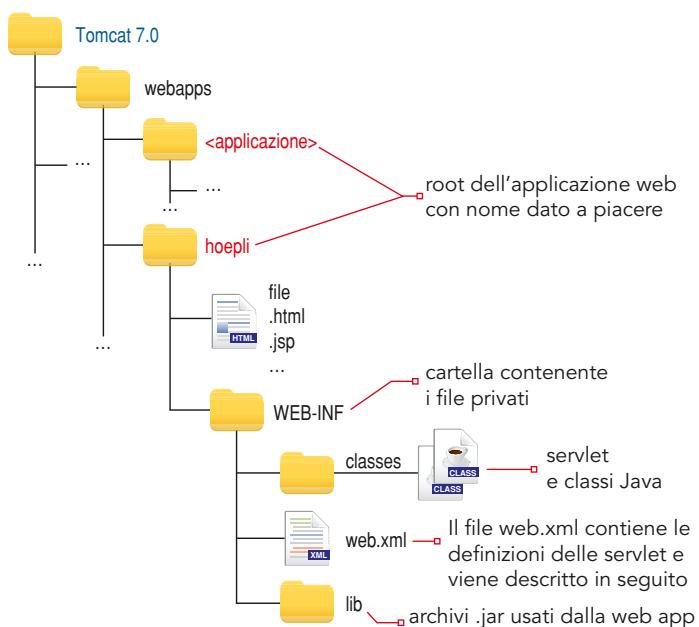
- ▶ la definizione del *run time environment* di una **web application** (WAP);
- ▶ il posizionamento di tutti i file della applicazione nelle opportune directory;
- ▶ la mappatura delle **URL** sulle **servlet**;
- ▶ la definizione delle impostazioni di default di un'applicazione (per esempio la *welcome page* e la *error pages*);
- ▶ la configurazione dei vincoli di sicurezza dell'applicazione.

Nel nostro calcolatore deve essere già stato installato il **web server**: il procedimento viene descritto nella prima esercitazione di laboratorio di questa unità didattica.

Supponiamo di avere già installato **Tomcat** come **web server**: questo richiede per ogni **WAP** una particolare strutturazione delle directory collocate a partire dalla sua cartella **WEBAPPS**, che è la **top level directory** di tutte le applicazioni, che deve essere rispettata dal programmatore:

- ▶ nella root directory dell'applicazione, nel nostro caso **hoepli**, vengono copiati i file: **\*.html**, **\*.jsp** ecc. viene creata una subdirectory **WEB-INF** con due sottocartelle **classes** e **lib**:
  - nella **WEB-INF** si memorizza il file **web.xml** (**Deployment Descriptor**);
  - in **WEB-INF/classes** i file **\*.class** delle **servlet** (bytecode);
  - in **WEB-INF/lib** eventuali file di libreria **\*.jar**.

L'albero completo di una **web application** è raffigurato a fianco: ▶



Predisponiamo la struttura destinata a raccogliere la nostra prima applicazione: nel nostro disco ci posizioniamo nella directory **programmi** dove individuiamo la cartella che contiene il software del **web server** (per esempio **Apache Software Foundation/Tomcat**); in quella posizione creiamo sotto **webapps** la directory **esempioTesto**, che sarà la root della nostra applicazione.

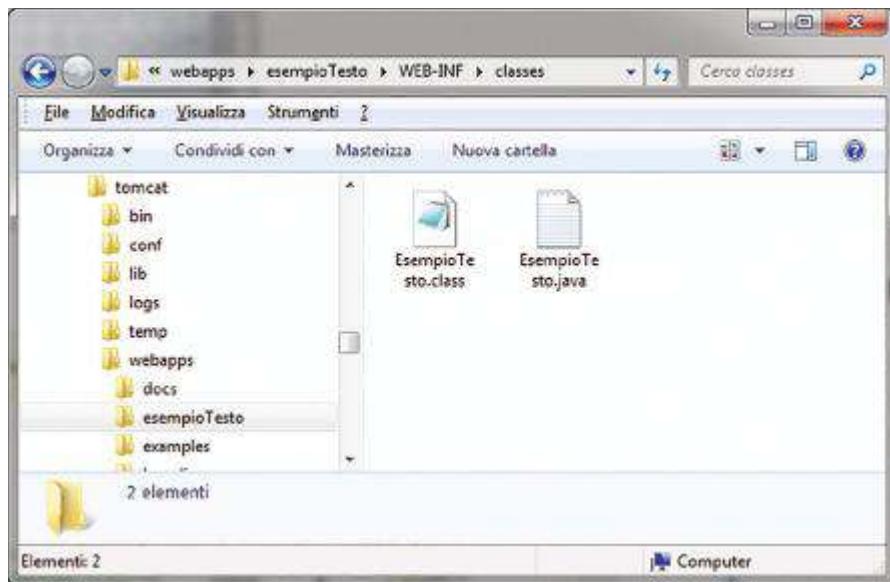
Replichiamo la struttura richiesta da **Tomecat**, cioè in sequenza:

- 1 nella cartella **webapps** creiamo la cartella col nome della nostra applicazione (noi realizzeremo due applicazioni, **esempioTesto** e **hoepli**);
- 2 creiamo la cartella **WEB-INF**;
- 3 ci posizioniamo al suo interno e creiamo la cartella **classes**;
- 4 ci posizioniamo nella cartella **classes** e vi copiamo la nostra classe **EsempioTesto.class**.

È comodo (ma non necessario) copiare nella cartella anche il file sorgente **EsempioTesto.java**, in modo da poterlo consultare ed eventualmente modificare.

È doveroso ricordare di fare particolare attenzione alla scrittura dei nomi in quanto siamo in una situazione di **case-sensitive** e spesso il funzionamento delle applicazioni è compromesso dalla errata digitazione delle iniziali minuscole e/o maiuscole.

A questo punto siamo nella seguente situazione:



Per mandare in esecuzione l'applicazione abbiamo due modalità:

- 1 digitare nel browser il nome della pagina **HTML** di inizio della applicazione che al suo interno richiama la **servlet**:

```
http://localhost:8080/<applicazione>/index.html
```

- 2 digitare nel browser il nome della servlet connessa alla applicazione:

```
http://localhost:8080/<applicazione>/servlet/<nome_servlet.class>
```

### ESEMPIO

Realizziamo una semplice pagina **HTML**, la **index.html**, con un hyperlink che non fa altro che richiamare la nostra **servlet** presente in **servlet/EsempioTesto**.

```

1 <html>
2 <head>
3 <title>Primo esempio di servlet</title>
4 </head>
5 <body>
6 <p><strong><font color="#0000FF" size="4" face="Arial, Helvetica, sans-serif">Primo
7 esempio di utilizzo delle servlet</font></strong></p>
8 <p><a href="servlet/EsempioTesto">Per eseguire la servlet clicca qui!</a> </p>
9 <p>&nbsp;</p>
10 </body>
11 </html>
12
13

```

La posizioniamo nella **cartella principale** della nostra applicazione web:



A questo punto abbiamo stabilito il contesto che deve essere registrato in un apposito file **XML**, il file **web.xml**, dove vengono riportate le path dei singoli file in modo che il web server li possa ritrovare al momento della richiesta di esecuzione nella applicazione: questo file prende il nome di **deployment descriptor o Context XML descriptor**.

## ■ Il Context XML descriptor o Deployment descriptor

Il file **web.xml**, chiamato anche **Deployment descriptor**, serve per configurare l'applicazione web in modo che **Tomcat** possa gestire le richieste da parte dei client: contiene l'elenco dei **servlet** e per ogni servlet permette di definire una serie di parametri come coppie nome-valore che costituiscono la descrizione del **Context** associato a una **web application**.

Nel file **web.xml** vengono specificati diversi parametri come:

- il nome della classe che definisce la **servlet**;
- i percorsi che causano l'invocazione della **servlet** da parte del **Container**;
- una serie di parametri di configurazione (coppie nome-valore).

Contiene anche la mappatura fra **URL** e **servlet** che compongono l'applicazione.

La struttura completa è la seguente

- <web-app> definisce la configurazione di tutte le servlet dell'applicazione web
- <display-name> specifica un nome che viene utilizzato dall'amministratore del server
- <description> specifica una descrizione dell'applicazione che viene visualizzata nel pannello del manager
- <servlet-name> definisce un nome per la servlet all'interno di questo file
- <description> fornisce una descrizione di questa servlet
- <servlet-class> specifica il nome completo della classe servlet compilata
- <servlet-mapping> specifica l'associazione tra il nome dato alla servlet nell'interno del file web.xml (<servlet-name>) e l'<url-pattern> con cui la servlet potrà essere invocata

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
    <!--descrizione generale della applicazione web -->
    <display-name>
        Primo esempio di utilizzo delle servlet
    </display-name>

    <description>
        Richiama l'esecuzione della classe Esempio0
    </description>

    <!<definizione delle servlet -->>
    <servlet>
        <servlet-name>Esempio0</servlet-name>
        <description>Visualizza una riga</description>
        <servlet-class>Esempio0</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Esempio0</servlet-name>
        <url-pattern>/servlet/Esempio0</url-pattern>
    </servlet-mapping>
</web-app>

```

## ESEMPIO

Nel nostro esempio il file **web.xml** può essere semplicemente il seguente:

```

File Edit Selection Find View Goto Tools Project Preferences Help
web.xml
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <web-app xmlns="http://java.sun.com/xml/ns/javaee"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://
5      java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
6      version="2.5">
7
8      <display-name>Primo esempio di utilizzo delle servlet</display-name>
9      <description>Richiama l'esecuzione della classe Esempio1</description>
10
11     <servlet>
12         <servlet-name>EsempioTesto</servlet-name> nome servlet
13         <servlet-class>EsempioTesto</servlet-class>
14     </servlet>
15
16     <servlet-mapping>
17         <servlet-name>EsempioTesto</servlet-name> mappatura
18         <url-pattern>/servlets/servlet/EsempioTesto</url-pattern>
19     </servlet-mapping>
20
21 </web-app>

```

L'URL può anche essere scritto in formato assoluto:

<http://localhost:8080/EsempioTesto/servlet/EsempioTesto>

Lo scriviamo con un editor e lo collichiamo nella cartella **WEB-INF**.

## ■ Esecuzione di una servlet

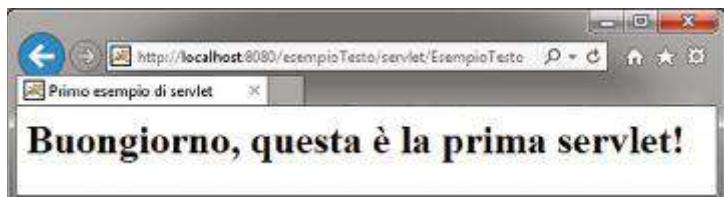
Al momento dell'invocazione di una **servlet**, Apache Tomcat esegue diverse operazioni:

- legge il file **web.xml**;
- trova un'entry **<servlet-mapping>** da cui capisce che una **servlet** deve essere eseguita in risposta all'**URL** richiesto;
- trova un'entry **<servlet>** che indica il nome della classe **Java**;
- controlla se esiste in memoria il file **.class** corrispondente o se deve essere ricaricato;
- viene creato un **thread** e mandato in esecuzione.

Siamo ora pronti per mandare in esecuzione la nostra servlet: come prima esecuzione richiamiamo la pagina **index.html** presente nella cartella **esempioTesto** digitandone nel browser semplicemente il nome della applicazione: ►



Cliccando sul collegamento hyperlink viene manda in esecuzione la **servlet** che produce il seguente risultato: ►



Al medesimo risultato si giungeva anche richiamando direttamente la **servlet** dalla linea di comando del browser:



Quando viene ricompilata una **servlet** e copiata nella cartella **classes** è necessario riavviare **Tomcat** per rendere effettiva la sostituzione con quella precedentemente caricata nella **RAM** del **server**.

L'installazione di **Tomcat** e la descrizione completa dell'utilizzo dell'ambiente di sviluppo viene descritto nella esercitazione di laboratorio 1 di questa unità di apprendimento.

## ■ Servlet concorrenti

Le **servlet** hanno il compito di servire le richieste contemporanee dei client e il **web server** per definizione utilizza una sola istanza di **servlet** condividendola fra le varie richieste: a fronte di ogni richiesta da parte di client della stessa **servlet** viene creato un **threads** differente da parte del web server capace di eseguire la **servlet** in maniera concorrente.

È però necessario tenere conto dei meccanismi di accesso concorrente alle **servlet** e ai loro dati in fase di programmazione, mediante l'operatore **synchronized**, come studiato nei classici problemi di concorrenza.

## ■ Vantaggi e svantaggi delle servlet

### Vantaggi delle servlet

Come già detto in precedenza, ancora oggi le tecnologie **CGI** sono molto utilizzate. Ricapitoliamo in breve i vantaggi più evidenti delle **servlet** rispetto alla chiamate **CGI**.

- **Efficienza:** si è detto che le **servlet** vengono istanziate e caricate una volta soltanto, alla prima invocazione e alle successive richieste da parte di nuovi client vengono creati nuovi thread e quindi non deve essere ricaricato ogni volta tutto il codice della classe: con il meccanismo delle **CGI** a ogni richiesta da parte di un **client** il **server** effettua il caricamento completo del processo **CGI** con un notevole degrado delle prestazioni.
- **Portabilità:** la tecnologia **Java** è facilmente portatile su ogni piattaforma senza problematiche connesse alla compatibilità in quanto sono intrinsecamente risolte dalla **JVM**.
- **Persistenza:** quando una servlet viene caricata rimane in memoria mantenendo intatte determinate informazioni anche alle successive richieste.
- **Gestione delle sessioni:** il protocollo **HTTP** è un protocollo **stateless** e quindi non è in grado di ricordare i dettagli delle precedenti richieste provenienti da uno stesso client: con le **servlet** è possibile superare queste limitazioni.

### Svantaggi delle servlet

I principali inconvenienti che denunciamo gli sviluppatori di applicazioni web che utilizzano le **servlet** sono i seguenti:

- con le **servlet** si “mescola” la logica della applicazione con la presentazione;
- se il codice **HTML** è complesso a volte è “particolarmente oneroso” doverlo “inglobare” all’interno del codice **Java** per mezzo di istruzioni di **println()**;
- non è possibile la “prototipazione rapida”: ogni modifica del codice richiede una ricompilazione esplicita e ogni aggiunta il cambiamento del file **web.xml** con il riavvio dell’applicazione o del **server**.

#### Servlet

A servlet is a small program that runs on a server. The term was coined in the context of the Java applet, a small program that is sent as a separate file along with a Web (HTML) page. Java applets, usually intended for running on a client, can result in such services as performing a calculation for a user or positioning an image based on user interaction.

Some programs, often those that access databases based on user input, need to be on the server. Typically, these have been implemented using a Common Gateway Interface (CGI) application.

However, with Java running on the server, such programs can be implemented with the Java program-



ming language. The advantage of a Java servlet on servers with lots of traffic is that they can execute more quickly than CGI applications. Rather than causing a separate program process to be created, each user request is invoked as a thread in a single daemon process, meaning that the amount of system overhead for each request is slight.

The “8080” port number in the URL means the request is intended directly for the Web server itself. The “servelt” would indicate to the Web server that a servlet was being requested.

Add-on modules allow Java servlets to run in Netscape Enterprise, Microsoft Internet Information Server (IIS), and Apache servers.

## Verifichiamo le conoscenze

### 1. Risposta multipla

**1** Quale di queste affermazioni sulle servlet è falsa?

- a. sono molto simili alle applet
- b. sono eseguite dalla JVM sul server
- c. richiedono un basso tempo di latenza
- d. a ogni servlet è associato un solo Thread
- e. sono più efficienti delle CGI

**2** Quale dei seguenti compiti non è relativo al container?

- a. gestisce la comunicazione con i client
- b. gestisce la sicurezza
- c. realizza il multithreading
- d. gestisce la attivazione della servlet
- e. effettua la compilazione della servlet
- f. gestisce la terminazione della servlet

**3** Quale tra i seguenti metodi non viene utilizzato nel protocollo HTTP?

- |             |                |               |
|-------------|----------------|---------------|
| a. doGet()  | c. doTrace()   | e. doDelete() |
| b. doPost() | d. doService() | f. doHead()   |

**4** Nella subdirectory WEB-INF non è presente:

- |                        |                    |
|------------------------|--------------------|
| a. la cartella webapps | c. la cartella lib |
| b. la cartella classes | d. il file web.xml |

**5** Quale tra i seguenti non è un vantaggio delle servlet rispetto alle CGI?

- |                |                            |
|----------------|----------------------------|
| a. efficienza  | d. prototipazione rapida   |
| b. portabilità | e. gestione delle sessioni |
| c. persistenza |                            |

### 2. Ordinamento

**1** Ordina la gerarchia completa di una servlet:

- a. ..... classe <applicazione>
- b. ..... classe astratta GenericServlet
- c. ..... classe astratta HttpServlet
- d. ..... interfaccia Servlet

### 3. Vero o falso

**1** Il programma CGI è scritto in un linguaggio di programmazione server oriented.

V F

**2** Gli script CGI vengono eseguiti dal sistema operativo.

V F

**3** Le servlet vengono eseguite dal sistema operativo.

V F

**4** Il web server svolge la funzione di container occupandosi della gestione del ciclo di vita delle servlet.

V F

**5** Una servlet interagisce con un web client attraverso il paradigma request/response.

V F

**6** Le servlet usate nel web sono estensioni della classe javax.servlet.Servlet.

V F

**7** Con deployment si intendono le operazioni eseguite per collaudare una servlet.

V F

**8** La cartella WEBAPPS è la top level directory di tutte le applicazioni su Tomcat.

V F

**9** Il file web.xml è chiamato anche deployment descriptor.

V F

# JDBC: Java DataBase Connectivity

In questa lezione impareremo...

- ▶ le caratteristiche dell'interfaccia JDBC
- ▶ i tipi di driver
- ▶ la connessione con MySQL e Access

## ■ JDBC

Sin dalle prime versioni di Java, la SUN ha messo a disposizione le API che permettono alle applicazioni scritte in questo linguaggio di avere un livello di astrazione uniforme verso i database: le JDBC. JDBC racchiude una serie di classi per poter accedere alle basi di dati con metodi e schemi di funzionamento molto intuitivi, in linea con lo stile di programmazione tipico di Java: sfruttando un apposito driver JDBC costituito da una classe Java è possibile connettersi a un particolare database e oggi tutti i principali DBMS dispongono di uno specifico driver JDBC.

Spesso JDBC viene interpretato come un acronimo (Java DataBase Connectivity) ma la Sun non ne rivendica la paternità e sostiene che JDBC è semplicemente il nome di una interfaccia, cioè di uno "strato di programmazione" che si interpone tra il codice Java e i database.

A corredo di JDBC vi è inoltre un particolare driver, chiamato ponte JDBC-ODBC, che permette l'utilizzo di qualsiasi fonte di dati per la quale è disponibile un driver ODBC, quindi mediante JDBC e il ponte ODBC è possibile interagire con tutti i DBMS presenti sul mercato da codice Java, sia dalle applicazioni stand-alone, che dalle servlet e dalle JSP.

La Sun ha incluso JDBC nel JDK a partire dalla versione 1.1, rimarcando la filosofia multipiattaforma che ha caratterizzato questo linguaggio non solo nei confronti dei sistemi operativi ma anche verso i database: lo scopo è quello di utilizzare una medesima modalità per connettersi a tutti i più diffusi e utilizzati database.

Non ci addentriamo nei dettagli ma dal punto di vista dello sviluppatore sono messi a disposizione tre strumenti che permettono di interagire omogeneamente con tutti i diversi database:

- ▶ gli strumenti per la connessione al database desiderato;
- ▶ i metodi per inviare i comandi SQL;
- ▶ alcuni potenti meccanismi per gestire le risposte ricevute dalla query effettuate sui database.

Dal punto di vista del programmatore è completamente trasparente il tipo di DB che sta utilizzando in quanto interagisce con tutti mediante la stessa interfaccia, cioè JDBC, che si connette al DB fisico semplicemente cambiando il "nome del driver" da utilizzare nella sua applicazione.



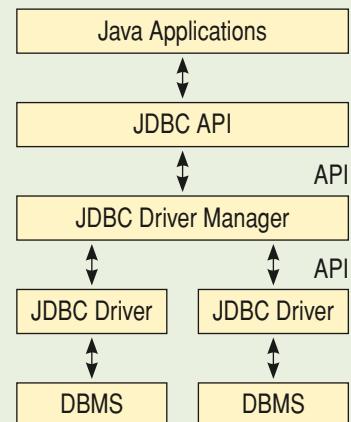
## Zoom su...

### ODBC

ODBC (Open DataBase Connectivity) è lo standard proposto da Microsoft nel 1991. È supportato da quasi tutti i DBMS relazionali e offre all'applicazione un'interfaccia che consente l'accesso a un DB indipendente da:

- il particolare dialetto di SQL;
- il protocollo di comunicazione da usare con il DBMS;
- la posizione del DBMS (locale o remoto).

Mediante un DSN (Data Source Name), che contiene tutti i parametri necessari alla connessione con il DB, permette di connettersi a un particolare DB semplicemente indicando il protocollo di comunicazione e il tipo di sorgente dati (lo descriveremo in dettaglio nella prima esercitazione di laboratorio dedicata a JDBC): ►



## ■ Tipi di driver JDBC

I driver JDBC possono essere di quattro diverse tipologie.

### Tipo 1. Bridge JDBC-ODBC

È il driver che permette l'accesso al DB nel caso in cui non sia stata realizzata una vera e propria implementazione JDBC verso un particolare database JDBC: si utilizza quindi un JDBC-ODBC bridge cioè un "driver-ponte" che converte le chiamate JDBC in ODBC.

Il codice ODBC binario e il codice del DBMS client devono essere caricati su ogni macchina client che usa JDBC-ODBC bridge.

È consigliabile utilizzare questa strategia solo quando non esistono soluzioni alternative dato che richiede l'installazione delle librerie sul client violando la filosofia write once, run anywhere (◀ Wora ▶) alla base delle applicazioni Java.

Dalla versione Java 8 viene deprecato il bridge JDBC-ODBC e quando si passa a questa versione è necessario sostituire la tipologia di connessione: noi useremo la libreria UCanAccess.

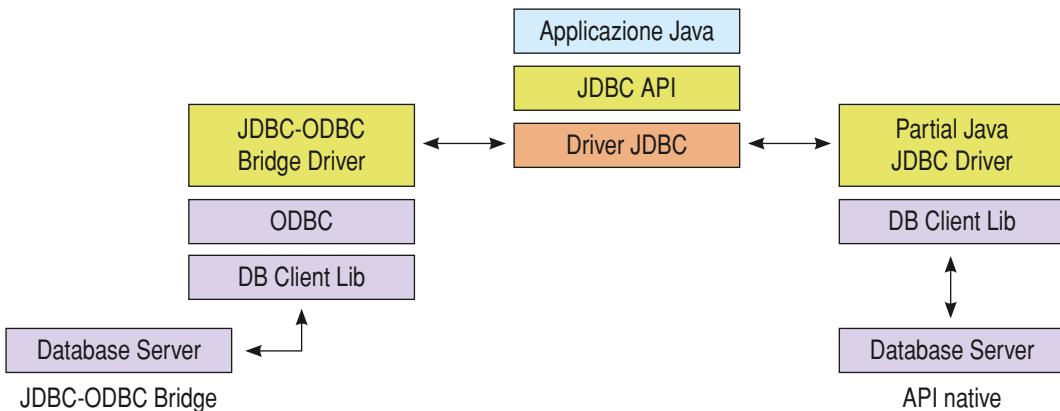


◀ Wora è uno slogan creato da Sun Microsystems per illustrare i benefici cross-platform del linguaggio Java. Fin dall'inizio l'intenzione di Sun è stata quella di realizzare Java come linguaggio indipendente dalla piattaforma grazie alla macchina virtuale: "Lo scrivi una volta, lo esegui dappertutto" significa svincolare il programmatore dall'hardware e quindi dalla configurazione della macchina e questo obiettivo viene a mancare se per "far funzionare" un programma Java su una certa macchina è necessario installare componenti aggiuntivi. ►

## Tipo 2. API Native

Quando si ha disposizione **API native**, come per esempio nel caso di **Oracle**, le chiamate **JDBC** non vengono tradotte in **ODBC** ma vengono utilizzate queste **API** appositamente per il database.

Le **API native** sono generalmente scritte in **C** o **C++** e sono molto performanti ma, come per il caso precedente, devono essere presenti nel **client**, quindi anche questo tipo di driver compromette la filosofia **wora**.

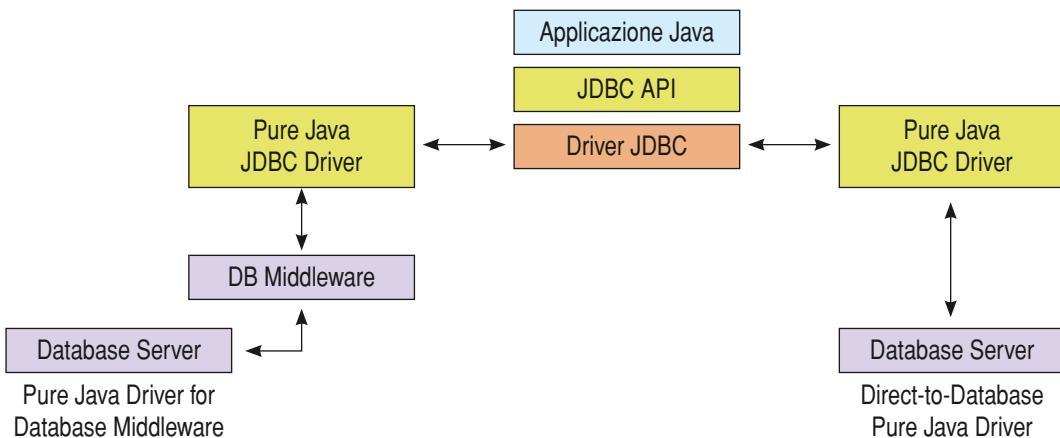


## Tipo 3. Pure Java Driver for Database Middleware

Rispetto ai precedenti, questi tipi di driver, scritti interamente in **Java**, sono particolarmente indicati da **utilizzarsi** con le **applet/servlet**, convertendo le chiamate **JDBC** nel protocollo di una **applicazione middleware** che, nel caso di una **LAN** non congestionata, potrebbe essere un vantaggio: è l'applicazione middleware che traduce la richiesta del **client** nel protocollo proprietario del **DBMS** sottostante garantendo l'accesso a diversi **DBMS**.

## Tipo 4. Direct-to-Database Pure Java Driver

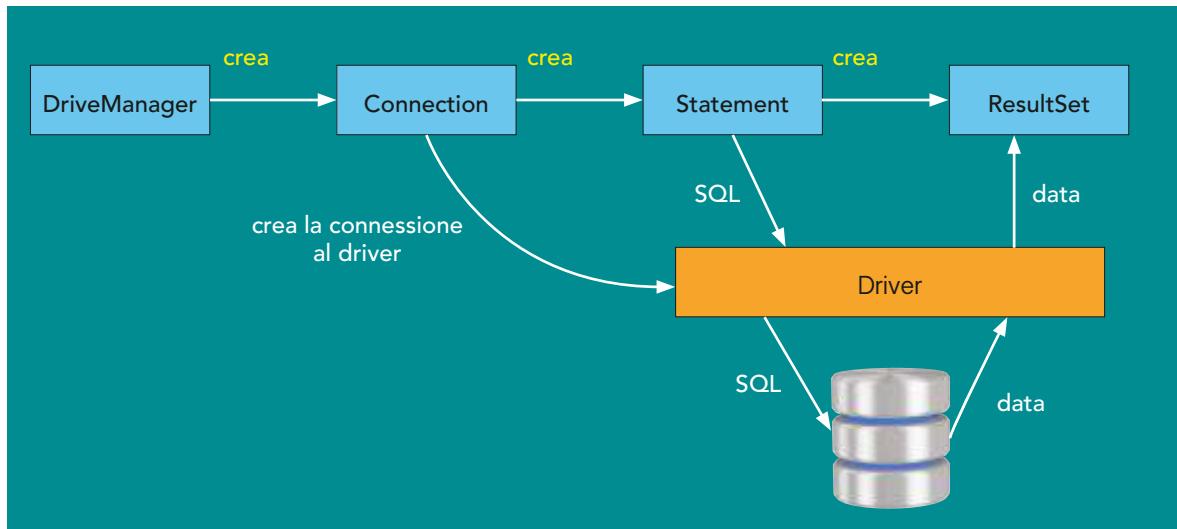
Questa tipologia è quella maggiormente indicata alle applicazioni **Java** in quanto il driver è realizzato totalmente in **Java** e trasforma le chiamate **JDBC** direttamente nel protocollo del database, quindi permette un collegamento diretto senza bisogno di **middleware** o **client** installati.



**UcanAccess** è un **Pure Java Driver**, è quindi noi "sposeremo" questa modalità di connessione.

## ■ Utilizzare JDBC standalone

L'ultima versione disponibile di JDBC è **5.1.39** ed è quella che noi utilizzeremo in questo volume: il suo utilizzo è molto semplice e può essere schematizzato in quattro operazioni, rappresentate nella figura seguente.



### 1) Driver

Per prima cosa è necessario caricare il driver idoneo per l'utilizzo del particolare database che si intende sfruttare: il nostro primo esempio di connessione sarà effettuato servendosi di **MySQL**, che dispone sia di un driver **JDBC** che di un driver **ODBC**.

Utilizziamo il driver JDBC chiamato **Connector/J** presente nella cartella **materiali** del **DVD** allegato al volume oppure scaricabile dalla pagina <http://dev.mysql.com/downloads/connector/j/>.

Affinché tale **driver** possa essere visto dalle applicazioni **Java**, è necessario notificare alla macchina virtuale la presenza dell'archivio **JAR** contenuto nel pacchetto scaricato. È possibile operare in diversi modi e il più semplice è depositare una copia dell'archivio **JAR** di **Connector/J** all'interno della **directory jre/lib/ext**, nella cartella che contiene l'installazione di **Java**. La tecnica più efficace (e più corretta) è quella di aggiungere il percorso dell'archivio **JAR** nella variabile di sistema **CLASSPATH**.

### ESEMPIO

Riportiamo un primo segmento di codice che permette di realizzare la connessione: dopo aver importato il contenuto del pacchetto **java.sql**, che contiene le **API JDBC**, per prima cosa definiamo una stringa dove scriviamo il nome del driver che utilizzeremo per la connessione, che nel nostro caso è **com.mysql.jdbc.Driver**.

```

1 import java.sql.*;
2 public class EsempioDB1
3 {
4     public static void main(String[] args)
5     {
6         // carico il driver per la connessione al DB mysql
7         String DRIVER = "com.mysql.jdbc.Driver";

```

Proviamo a effettuare il suo caricamento in un costrutto **try... catch** per intercettare l'eventuale errore di mancato caricamento: una **java.lang.ClassNotFoundException**, infatti, può essere propagata nel caso in cui non sia possibile trovare il driver nel **CLASSPATH** della macchina virtuale.

```

8     try                      // carico il driver
9     {
10        Class.forName(DRIVER);
11    }catch (ClassNotFoundException e1)
12    {
13        // il driver non può essere caricato
14        System.out.println("Driver non trovato..."); 
15        System.exit(1);
16    }

```

## 2) Connessione

Come secondo passo si **apre una connessione** verso un database necessario all'applicazione, sfruttando il driver caricato al passo precedente. Definiamo due variabili.

```

19 // nome ed indirizzo del database:
20 String URL_micDB = "jdbc:mysql://localhost:3306/provejava";
21 // definizione delle query
22 String query = "SELECT Cognome, Nome, Indirizzo, Citta FROM amici";

```

- La prima per memorizzare l'indirizzo del database; la sintassi desiderata da **Connector/J** è la seguente:

```
jdbc:mysql://[hostname][:port]/[dbname][?param1=val1][&param2=val2]...
```

nel nostro caso stiamo eseguendo **MySQL** nella stessa macchina che eseguirà la classe (**local host**) e la porta ascoltata dal **DBMS** è la numero **3306**, valore di default predefinito di **MySQL**.

- La seconda per contenere la stringa di **query** in linguaggio **SQL** che estrae i nominativi della tabella **amici** riportata di seguito.

Cognome	Nome	Indirizzo	Citta	Sesso
Rossi	Mario	Via Milano,12	Roma	M
Gialli	Rina	Via Napoli,12	Bari	F
Verdi	Gino	Via Roma,14	Torino	M
Bianchi	Filippo	Via Torino,34	Milano	M
Pini	Pino	via del lago,4	Milano	M
Bianchi	Elena	Via Pascoli,23	Roma	F

Dopo aver creato un oggetto **Connection** necessario per stabilire una connessione con il **DBMS** cerchiamo di stabilire la connessione in un blocco **try... catch**, poiché è a rischio di eccezione.

```

21 // stabilisco la connessione
22 System.out.println("Connessione con: " + URL_micDB);
23 Connection connessione = null;
24 try                      // apro la connessione verso il database
25 {
26    connessione = DriverManager.getConnection(URL_micDB, "root", "");
27 }
28 catch (Exception e)
29 {
30     // gestione dell'errore
31     System.out.println("Errore nella connessione: " + e);
32     System.exit(1);
33 }

```

I parametri che passiamo al metodo `getConnection()` sono riportati nella tabella seguente.

nome e indirizzo del database:	"jdbc:mysql://localhost:3306/provejava"
nome utente:	"root"
password (nessuna):	""

Siccome le connessioni occupano risorse, in un ambiente multiutente e multitasking è opportuno adottare la seguente politica:

- aprire una connessione solo quando necessario;
- assicurarsi di chiuderla al termine del suo utilizzo;
- e soprattutto non aprire/chiedere connessioni inutilmente.

### 3) Statement

Se la connessione va a buon fine, viene creato un oggetto `java.sql.Statement`, necessario per interagire con il DBMS attraverso delle query espresse in linguaggio **SQL** che memorizzano il risultato in un oggetto `java.sql.ResultSet`:

```

15 try {
16 {
17     // creo uno Statement per interagire con il database
18     Statement statement = connessione.createStatement();
19     // interroga il DBMS mediante una query SQL
20     ResultSet resultSet = statement.executeQuery(query);
21 }
```

### 4) ResultSet

I risultati ottenuti possono essere manipolati sfruttando i metodi dell'oggetto `ResultSet`, per esempio con un ciclo scorriamo il contenuto del `ResultSet` o lo visualizziamo sullo schermo.

The screenshot shows a Java code editor and a terminal window. The code in the editor is:

```

43 while (resultSet.next()) {
44     String cognome = resultSet.getString(1);
45     String nome = resultSet.getString(2);
46     String indirizzo = resultSet.getString(3);
47     String citta = resultSet.getString(4);
48
49     System.out.println("Lette informazioni...");
50     System.out.println("Cognome: " + cognome);
51     System.out.println("Nome: " + nome);
52     System.out.println("Indirizzo: " + indirizzo);
53     System.out.println("Città: " + citta);
54     System.out.println();
55 }
```

The terminal window titled "BlueJ Terminal Window - classes" displays the output of the program, which reads from a database and prints the results to the console. The output shows four records with columns Cognome, Nome, Indirizzo, and Città.

Il metodo `next()` di un oggetto `resultSet`, ha duplice funzionalità:

- scorre in avanti l'elenco dei record ottenuti;
- restituisce `true` fin quando non si giunge al termine dell'esplorazione.

Il risultato a video mostrato a fianco: ►

The terminal window titled "BlueJ Terminal Window - classes" shows the execution of the Java code. It prints four sets of data, each consisting of Cognome, Nome, Indirizzo, and Città. The data is identical to what was shown in the previous screenshot.

Al termine di ogni operazione la connessione viene chiusa nel ramo della clausola finally, dopo aver controllato se essa è ancora attiva (!=null).

```

62     finally{
63         if (connessione != null){
64             try {
65                 connessione.close(); // chiusura connessione
66             }
67             catch (Exception e)
68             {
69                 System.out.println("Errore nella chiusura della connessione");
70             }
71         }
72     }

```

Il codice completo è nel file [EsempioDB1.java](#) che puoi scaricare dalla cartella **materiali** nella sezione del sito [www.hoepliscuola.it](http://www.hoepliscuola.it) dedicata a questo volume.

## ■ Servlet con connessione a MySQL

Per connettere una **servlet** al database **MySQL** non ci sono particolari differenze rispetto alla applicazione stand-alone prima descritta: naturalmente è necessario aggiungere le direttive **HTML** per visualizzare i risultati nel browser del **client**.

L'unica rettifica/integrazione risulta essere il completamento del comando di connessione dove per le applicazioni lato **server** devo essere indicati l'**utente** e la **password**:

```

33     // apro la connessione verso il database
34     connessione = DriverManager.getConnection(URL_mioDB, "root", "");

```

Riportiamo di seguito solo la parte del codice inerente alla visualizzazione dove è stato aggiunto il codice **HTML**, necessario per visualizzare sullo schermo il risultato della interrogazione, in modo formattato utilizzando il tag **<pre>**:

```

40     // scrollo e mostro i risultati.
41     out.println("<pre>");
42     out.println("<b>cognome"+";"+nome"+";"+indirizzo"+";"+citta</b>");
43     while (resultSet.next()) {
44         String cognome = resultSet.getString(1);
45         String nome = resultSet.getString(2);
46         String indirizzo = resultSet.getString(3);
47         String citta = resultSet.getString(4);
48         out.println(cognome+";"+nome+";"+indirizzo+";"+citta);
49     }

```

Mandando in esecuzione la **servlet** otteniamo il seguente output:



## ■ Applicazione e servlet con connessione ad Access

Come abbiamo già accennato, dalla versione 8 di Java viene deprecato il bridge JDBC-ODBC che, in ogni caso, ha sempre dato qualche problema per le differenze tra le versioni a **32bit** e **64bit** e per le diverse versioni di rilascio da parte della **Microsoft** del database **Access**.

In questa lezione noi proponiamo di utilizzare quello che di fatto è diventato lo “standard post ODBC”, che sono le librerie **UCanAccess**.

Sono scaricabili gratuitamente all’indirizzo <http://ucanaccess.sourceforge.net/site.html> e alla data di scrittura di questa lezione è disponibile la versione riportata in figura:



### Codice per classi standalone

Le rettifiche da portare al programma precedentemente descritto sono minime: per prima cosa è necessario indicare la nuova “libreria di connessione” con l’istruzione di riga 20:

```

17 // parametri per la connessione al database access
18 String DNS = "";
19 String URL_micDB = "";
20 DNS = "jdbc:ucanaccess://"           // connessione alla libreria
21 URL_micDB = "F:/xampp/tomcat/webapps/hoepli/esempioDB/proveJava.mdb"; // path del db
22 URL_micDB = DNS + URL_micDB;
23 System.out.println("Connessione con: " + URL_micDB);

```

Quindi si procede con la creazione della connessione, ricordando che in locale non è presente nessun utente e nessuna password per connettersi al file.mdb.

```

25 // stabilisco la connessione
26 String utente = "";
27 String psw   = "";
28 Connection connessione = null;
29 try                      // apro la connessione verso il database
30 {
31     connessione = DriverManager.getConnection(URL_micDB, utente, psw);
32 }
33 catch (SQLException e)      // gestione dell'errore
34 {
35     System.out.println("Errore nella connessione");
36     System.exit(2);
37 }

```

Una volta definita la query, si procede creando un oggetto di tipo **Statement** necessario per inviare l’interrogazione **SQL** e il suo risultato verrà memorizzato in un oggetto **ResultSet** che, di fatto, è una tabella composta da righe e colonne.

```

39 // definizione della query
40 String query = "SELECT Cognome, Nome, Indirizzo, Citta FROM Amici";
41
42 try                                // esecuzione della query
43 {
44     Statement statement = connessione.createStatement();
45     // interrogo il DBMS mediante una query SQL
46     ResultSet resultSet = statement.executeQuery(query);
47
48     // scorro il resultSet e mostro i risultati
49     while (resultSet.next()) {
50         String cognome = resultSet.getString(1);
51         String nome = resultSet.getString(2);
52         String indirizzo = resultSet.getString(3);
53         String citta = resultSet.getString(4);
54
55         System.out.println("Lette informazioni...");
56         System.out.println("Cognome: " + cognome);
57     }
58 }
```

Come già descritto nell'esempio precedente utilizziamo un ciclo per scorrere il contenuto del **ResultSet** e visualizzarlo sullo schermo.

### Codice per servlet

Per poter utilizzare la libreria **UCanAccess** nel codice delle servlet sono necessarie tre modifiche rispetto a quello scritto per l'accesso a **MySQL**:

- 1 indicare il nome della libreria, come indicato alla riga 9;
- 2 aggiungere al nome del database il parametro che indica le modalità di gestione della memoria: nel nostro caso è necessario porre il parametro **memory=false**;
- 3 indicare il nome completo del driver, come riportato nella riga 17.

```

1 public class ServletDB1 extends HttpServlet
2 {
3
4     private final String protocollo = "jdbc:ucanaccess://";      // connessione alla libreria
5     private final String mdbpath = "tomcat/webapps/hoepli/esempioDB/"; // percorso relativo
6     private final String mdbName = "proveJava.mdb;memory=false";    // nome database
7     private final String user   = "";                                // nome utente
8     private final String psw    = "";                                // password
9
10    // riferimento al database
11    private final String URL_micDB = protocollo + mdbpath + mdbName;
12    // definizione del driver per la connessione al DB Access
13    private final String DRIVER = "net.ucanaccess.jdbc.UcanaccessDriver";
14 }
```

Il resto della **Servlet** non richiede interventi degni di particolari osservazioni: nelle lezioni di laboratorio saranno presentati esempi completi e proposte situazioni da risolvere come esercitazione.

### Output dei programmi

Il risultato delle due esecuzioni è riportato nelle seguenti figure in modo da poter effettuare il confronto.

cognome	nome	indirizzo	citta
Bianco	Maria	via Como,4	Bari
Bianchi	Filippo	Via Torino,32	Milano
Gialli	Pina	Via Napoli,23	Roma
Rossi	Mario	Via Milano,12	Roma
Verdi	Gino	Via Roma,14	Torino



### wora

"**Write once, run anywhere**", or sometimes write once, run everywhere (**WORE**), is a slogan created by Sun Microsystems to illustrate the cross-platform benefits of the Java language. From the beginning, Sun intended Java to be a platform-neutral language. That is, Java programs would be written to run on a Virtual Machine instead of a physical computer. The virtual machine, then, would run on a real computer.

Because the Java Virtual Machine has to strictly adhere to interface and behavior definitions, a programmer can develop a program for the Java Virtual Machine on the sort of computer that he likes, and expect it to run on the sort of computers that his customers like, inside of their Java Virtual Machines.

### Driver

A **device driver** is a software program that enables a specific hardware device to work with the operating system of a computer. A **database driver** is a software routine that accesses a database. It allows an application or compiler to access a particular database format.

### JDBC

Java Database Connectivity (JDBC) is an application program interface (API) specification for connecting programs written in Java to the data in popular databases. The application program interface lets you encode access request statements in Structured Query Language (SQL) that are then passed to the program that manages the database. It returns the results through a similar interface. JDBC is very similar to the SQL Access Group's Open Database Connectivity (ODBC) and, with a small "bridge" program, you can use the JDBC interface to access databases through the ODBC interface. For example, you could write a program designed to access many popular database products on a number of operating system platforms. When accessing a database on a PC running Microsoft's Windows 2000 and, for example, a Microsoft Access database, your program with JDBC statements would be able to access the Microsoft Access database.

### UCanAccess

UCanAccess is an open-source Java JDBC driver implementation that allows Java developers and JDBC client programs (e.g., DBeaver, NetBeans, SQLEO, OpenOffice Base, LibreOffice Base, Squirrel SQL) to read/write Microsoft Access databases.

UCanAccess requires Java 6 or later to run.

When dealing with large databases and using the default "memory" setting (i.e., with driver property mem-

ory=true), it is recommended that users allocate sufficient memory to the JVM using the -Xms and -Xmx options. Otherwise it will be necessary to set the driver's "memory" property to "false".

### UCanAccess - Features

- Supported Access formats: 2000, 2002/2003, 2007, 2010/2013/2016 databases. (Access 97 format supported for **read-only**.)
- SELECT, INSERT, UPDATE, DELETE statements. Transactions and savepoints.
- Access data types: YESNO, BYTE, INTEGER, LONG, SINGLE, DOUBLE, NUMERIC, CURRENCY, COUNTER, TEXT, OLE, MEMO, GUID, DATETIME.
- Concurrent access from multiple application server threads.
- Connection pooling.
- ANSI 92 SQL, core SQL-2008.
- Access SQL.
- Core built-in functions for Access SQL are supported (e.g., Date(), Now(), IIf(), ...).
- You can execute Select queries defined and saved in Access. Since version 1.0.1, UCanAccess can execute Crosstab queries (TRANSFORM ... PIVOT ...)
- LIKE operator accepts both standard % and Access-specific \* as multi-character wildcards, both standard \_ and Access-specific ? as single-character wildcards, # for single digit and [xxx][!xxx] for character intervals (Access).
- Both & and + (Access SQL) as well as || (ANSI SQL) for string concatenation.
- Square brackets for field names, table names and view names (saved Access queries) that include spaces or other special characters, e.g., SELECT [my column] FROM [my table].
- Access date format (e.g., #11/22/2003 10:42:58 PM#).
- Both double quote " and single quote ' as SQL string delimiters.
- Read and write support to complex types (i.e., array of Version, Attachment, SingleValue).
- Since version 2.0.1, UCanAccess supports both constants and functions as default column values in CREATE TABLE statements  
e.g., CREATE TABLE tbl (fld1 TEXT PRIMARY KEY, fld2 LONG DEFAULT 3 NOT NULL, fld3 TEXT(255) NOT NULL, fld4 DATETIME DEFAULT Now(), fld5 TEXT DEFAULT 'bla')
- Command-line console ("console.bat" and "console.sh"). You can run SQL commands and display their results. CSV export command included.

## Verifichiamo le conoscenze



### 1. Risposta multipla

**1** ODBC è l'acronimo di:

- a. Open DataBase Connectivity
- b. Object DataBase Connectivity
- c. Open DataBase Communication
- d. Object DataBase Comunication

**2** ODBC offre all'applicazione un'interfaccia che consente l'accesso a un DB indipendente da (3 risposte):

- a. il particolare dialetto di SQL
- b. il tipo di sistema operativo
- c. il protocollo di comunicazione da usare con il DBMS
- d. la posizione del DBMS (locale o remoto)

**3** I driver JDBC possono essere di quattro diverse tipologie (indicare quello non corretto):

- a. Tipo Bridge JDBC-ODBC
- b. Tipo API Native
- c. Tipo Direct Bridge for Database Middleware
- d. Tipo Pure Java Driver for Database Middleware
- e. Tipo Direct-to-Database Pure Java Driver

**4** L'utilizzo di JDBC è molto semplice e può essere schematizzato in quattro operazioni nell'ordine:

- a. Connection-driver-statement-resulSet
- b. Driver-statement -connection -resulSet
- c. Driver-connection-statement-resulSet
- d. Driver-connection-statement-resulSet
- e. Connection-statement- driver-resulSet

**5** Le connessioni JDBC occupano risorse, in un ambiente multiutente e multitasking è opportuno:

- a. aprire una connessione solo quando necessario
- b. assicurarsi di chiuderla al termine del suo utilizzo
- c. aprire una connessione alla volta
- d. non aprire/chiudere connessioni inutilmente

**6** Quale tra i seguenti è un percorso corretto per un driver JDBC-ODBC?

- a. sun.jdbc.Jdbc.JdbcOdbcDriver
- b. sun.jdbc.jdbc.OdbcJdbcDriver
- c. sun.jdbc.jdbc.JdbcOdbcDriver
- d. sun.jdbc.jdbc.OdbcJdbcDriver



### 2. Vero o falso

**1** JDBC racchiude una serie di classi per poter accedere alle basi di dati.

V F

**2** La Sun ha incluso JDBC nel JDK a partire dalla versione 2.1.

V F

**3** Il codice ODBC binario e il codice del DBMS client devono essere caricati su ogni macchina client che usa JDBC-ODBC bridge.

V F

**4** Il Bridge JDBC-ODBC rispetta la filosofia WORA.

V F

**5** Le Bridge API Native rispettano la filosofia WORA.

V F

**6** La tecnica più corretta per l'utilizzo di Connector/J è quella di copiarla all'interno della directory jre/lib/ext, nella cartella che contiene l'installazione di Java.

V F

**7** Se la connessione va a buon fine, viene creato un oggetto java.sql.Statement.

V F

**8** Le query SQL memorizzano il risultato in un oggetto java.sql.ResultSet.

V F

**9** Con Access non è possibile evitare la definizione di un DSN per puntare a file MDB.

V F

# ESERCITAZIONI DI LABORATORIO 1

## XAMPP E IL SERVER ENGINE TOMCAT

Per poter eseguire una **servlet** il **Web server** deve avere il **servlet container** (o **servlet engine**), che fornisce l'ambiente di esecuzione a una **servlet**: nei nostri esempi noi utilizzeremo **Tomcat**, che è il **servlet container** open source più utilizzato interamente scritto in **Java**.

Mentre scriviamo queste esercitazioni è disponibile la versione **9.0** presente nel **DVD** allegato oppure scaricabile direttamente all'indirizzo <http://tomcat.apache.org/> oppure dalla cartella materiali nella sezione del sito [www.hoepiscuola.it](http://www.hoepiscuola.it) dedicata a questo volume.

### AREA digitale

 Installazione di Tomcat su Apache

### ■ Pacchetti con Web server, motore PHP e DBMS server

Oltre ai **Web server** che abbiamo descritto nell'unità 1 di questo volume, sono disponibili ambienti "integrati" scaricabili gratuitamente da internet che contengono tutti gli strumenti per la realizzazione e la gestione di applicazioni web: ad esempio contengono il **Web server**, un motore per il linguaggio di programmazione dinamico **PHP**, un **Database** e un **Database Administrator** (in alcune versioni persino **Tomcat** e **Perl**).

Vengono anche chiamati generalmente con il termine **LAMP**, acronimo di **Linux**, **Apache**, **MySQL**, **PHP**.

### Il pacchetto XAMPP



◀ **XAMPP** è un acronimo dal seguente significato: la **X** sta per **cross-platform**, la **A** sta per **Apache HTTP Server**, la **M** sta per **MySQL**, la **P** sta per **PhP** e l'ultima **P** sta per **Perl**. ▶

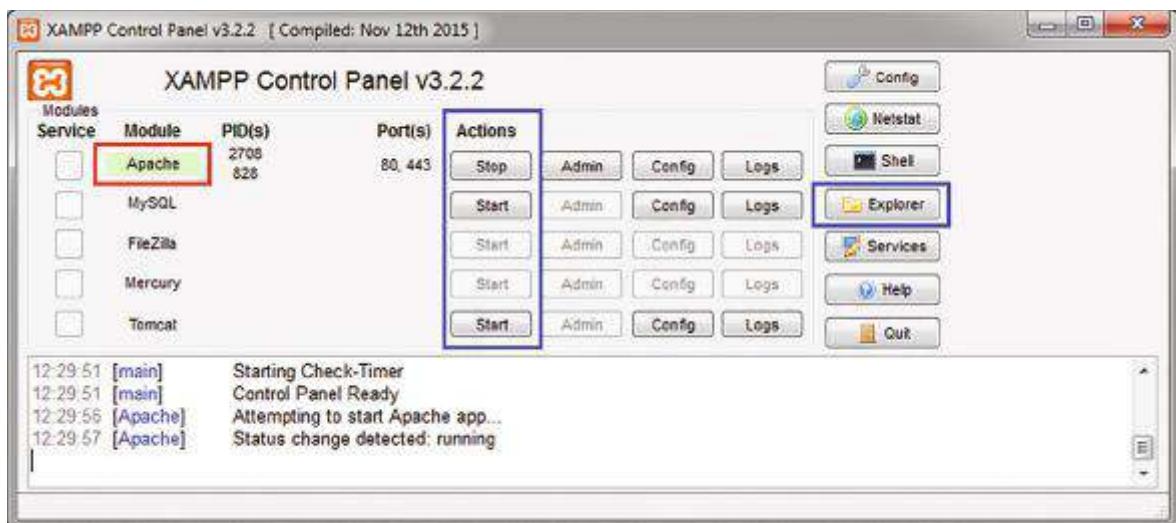
Tra i pacchetti più diffusi c'è il pacchetto ◀ **XAMPP** ▶ che è una piattaforma software gratuita che consente di installare in una sola volta il **Web server Apache**, il motore **PHP**, il **server DBMS MySQL Admin** e altri componenti, tra cui un **server FTP (FileZilla)** e nella versione **Full** anche **Tomcat**.

L'installazione è molto semplice e si può effettuare direttamente dal sito <http://xampp.softonic.it> confermando tutto quanto viene proposto di default: contemporaneamente vengono installati e configurati tutti i suoi componenti.

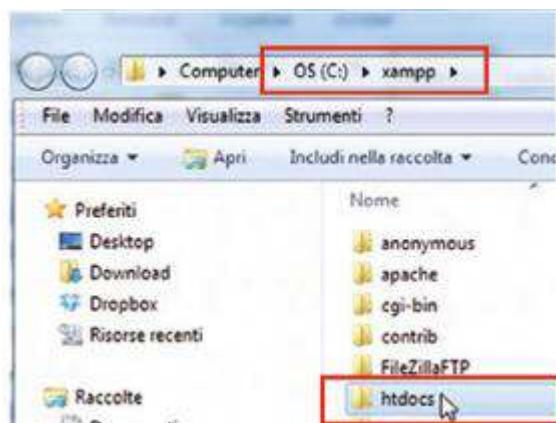
Una volta installato il programma appare la tipica icona nella **system tray** di **Windows**:



Facendo clic su di essa appare la finestra di monitor che consente di avviare o fermare i vari componenti. In questo caso abbiamo avviato **Apache**, e lo verifichiamo dallo sfondo verde e dalla label **[Stop]** sul pulsante delle **Azioni** che ci permette di fermarne l'esecuzione.



Il pulsante **Explore** mostra la directory in cui è stato installato XAMPP.



Vedremo in seguito che la directory **htdocs** sarà la **document root**, dove saranno memorizzati i file per le applicazioni scritte in linguaggio **PHP**.

È anche disponibile una comoda versione portabile di **XAMPP** scaricabile all'indirizzo:  
<http://portableapps.com/apps/development/xampp>

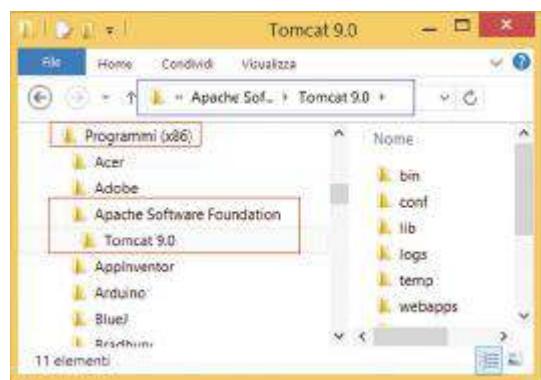
È necessario scaricare la versione completa in quanto in essa è presente anche **Tomcat**.



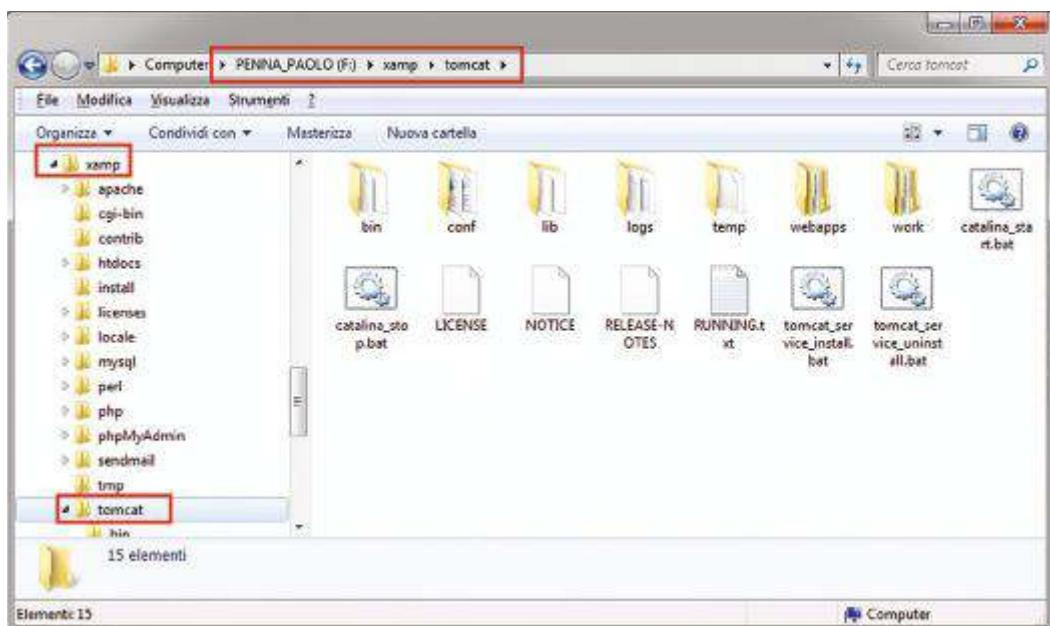
Nella cartella **materiali** del **DVD** allegato al presente volume è presente una copia del file:  
**xampp-portable-win32-5.6.23-0-VC11-installer.exe**  
 che è quella che è stata utilizzata in tutti gli esempi riportati nel presente volume.

## ■ Struttura di Tomcat

Se Tomcat è stato installato nella directory di default, cioè in una sottocartella della directory **programmi**, siamo nella situazione mostrata in figura: ▶

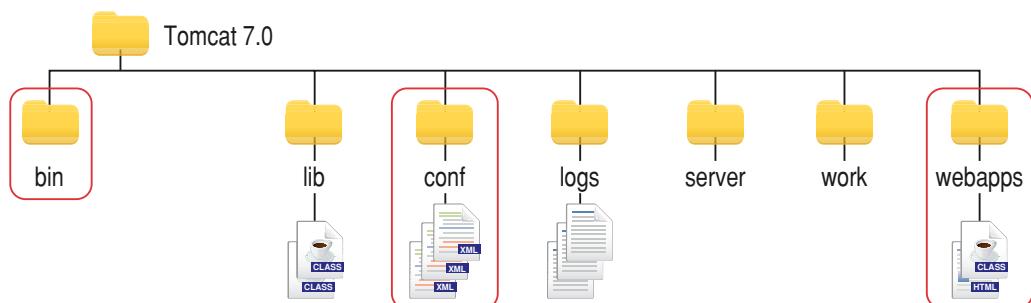


In alternativa, se abbiamo XAMPP installato sulla **pen-drive** e abbiamo scelto come directory di installazione proprio quella che lo contiene, siamo nella seguente situazione: ▼



Nel seguito non faremo distinzioni tra le due possibili situazioni, cioè ambiente **Tomcat** installato su **hard-disk** oppure su **pen-drive**, in quanto hanno le medesime sottocartelle a partire dalla cartella **Tomcat**.

L'interno della directory **Tomcat** ha una struttura più complessa rispetto ad **Apache**: la riportiamo di seguito indicando per ciascuna sottocartella i file contenuti.



Le sottocartelle più significative sono:

- ▷ **tomcat/bin** è la cartella che contiene gli eseguibili per il funzionamento del **web server**;
- ▷ **tomcat/lib** contiene le classi per la **JVM** di **Jakarta/Tomcat**;
- ▷ **tomcat/conf** contiene i file di configurazione di **Tomcat**;
- ▷ **tomcat/webapps** è la cartella dove saranno posizionate le applicazioni **web**, ciascuna a partire da una specifica directory avente come nome proprio l'applicazione.

## Configurazione di Tomcat

La configurazione di **Tomcat** viene effettuata mediante file formato **XML** presenti nella cartella **conf**, tra i quali ricordiamo:

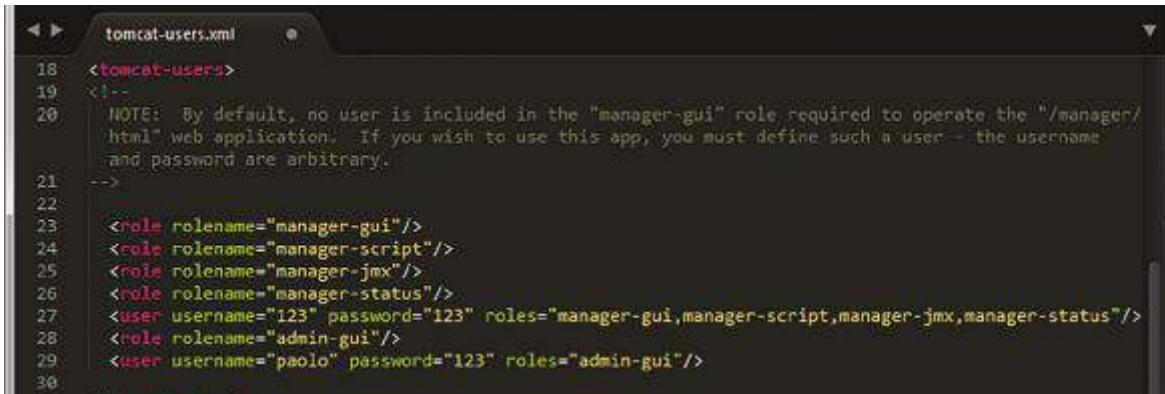
- ▷ **server.xml** contiene la configurazione globale dei server (specifica e configura i connector);
- ▷ **context.xml** permette di definire i parametri per avere dati persistenti per le **Servlet**;
- ▷ **web.xml** contiene i dati per la configurazione delle **Servlet**;
- ▷ **tomcat-users.xml** contiene le informazioni per il controllo d'accesso.

Non entriamo nel dettaglio di ciascuno di essi ma, a titolo di esempio, visualizziamo il contenuto del file **tomcat-users.xml** che può essere modificato per aggiungere gli utenti e assegnare i ruoli per l'accesso e la manutenzione del Web server.

I possibili ruoli sono sostanzialmente due, con diverse opzioni:

- 1 **admin**
- 2 **manager**
  - ▷ **gui**
  - ▷ **script**
  - ▷ **status**
  - ▷ **jmx**

Modifichiamo la configurazione di default e aggiungiamo un nuovo utente (noi abbiamo aggiunto un utente con password **123** e con username **palo**).



```

<?xml version="1.0" encoding="UTF-8"?>
<tomcat-users>
<!--
  NOTE! By default, no user is included in the "manager-gui" role required to operate the "/manager/html" web application. If you wish to use this app, you must define such a user -- the username and password are arbitrary.
-->
<role rolename="manager-gui"/>
<role rolename="manager-script"/>
<role rolename="manager-jmx"/>
<role rolename="manager-status"/>
<user username="123" password="123" roles="manager-gui,manager-script,manager-jmx,manager-status"/>
<user username="palo" password="123" roles="admin-gui"/>
</tomcat-users>

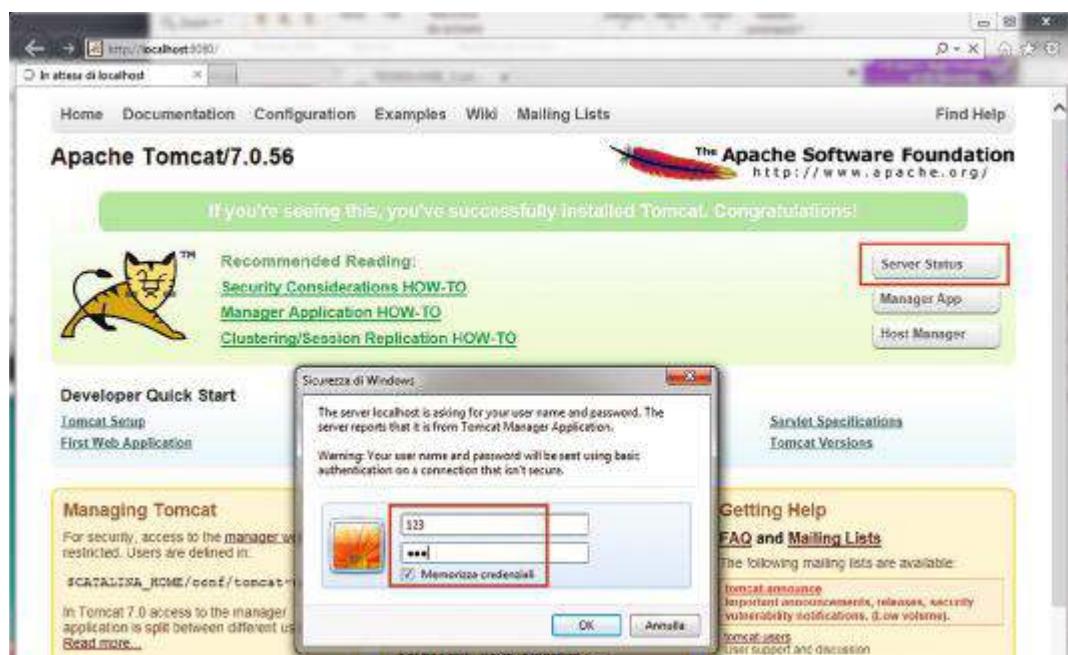
```

In questo modo “ci siamo dati le autorizzazioni” per effettuare tutte le operazioni con **Tomcat**.

Possiamo verificare la nuove possibilità operative accedendo alla sezione riservata cliccando sul pulsante **[Admin]** dopo aver avviato **Tomcat**.



Nella schermata principale di **Tomcat** clicchiamo su **[Server Status]**:



e digitiamo i dati appena inseriti nel file di configurazione alla richiesta di autenticazione:

The screenshot shows the Apache Tomcat Manager application at the URL <http://localhost:8080/manager/status>. The page has a header with the Apache Software Foundation logo and a cartoon cat. Below the header, there's a 'Server Status' section with tabs for 'Manager', 'List Applications', 'HTML Manager Help', 'Manager Help', and 'Complete Server Status'. Under 'Server Information', there's a table with columns: Tomcat Version, JVM Version, JVM Vendor, OS Name, OS Version, OS Architecture, Hostname, and IP Address. The data shows 'Apache Tomcat/7.0.56', '1.8.0\_65-b17', 'Oracle Corporation', 'Windows 7', '6.1', 'amd64', 'PC-Win7', and '192.168.1.100'. The 'JVM' section shows memory usage: Free memory: 125.08 MB, Total memory: 171.50 MB, Max memory: 869.50 MB. It also lists memory pool statistics for 'PS Eden Space', 'PS Old Gen', 'PS Survivor Space', 'Code Cache', 'Compressed Class Space', and 'Metaspace'. The 'ajp-bio-8009' section shows connection statistics: Max threads: 200, Current thread count: 0, Current thread busy: 0, Max processing time: 0 ms, Processing time: 0 ms, Request count: 0, Error count: 0, Bytes received: 0.00 MB, Bytes sent: 0.00 MB. At the bottom, there's a table with columns: Stage, Time, B Sent, B Recv, Client (Forwarded), Client (Actual), VHost, and Request.

Cliccando invece su **Manager App** la videata presentata sarà quella riportata alla pagina seguente; tale videata è molto utile in fase di messa a punto dell'applicazione **web** in quanto permette di fermare e ricaricare la singola applicazione senza bisogno di fermare il **web server**.

The screenshot shows a web browser window with the URL `http://localhost:8080/manager/html`. The page title is "Tomcat Web Application Manager". At the top, there is a logo for "The Apache Software Foundation" and a cartoon cat icon. Below the title, there is a message box containing "Message: OK". A navigation bar with tabs for "Manager", "List Applications", "HTML Manager Help", "Manager Help", and "Server Status". The main content area is titled "Applications" and contains a table with the following data:

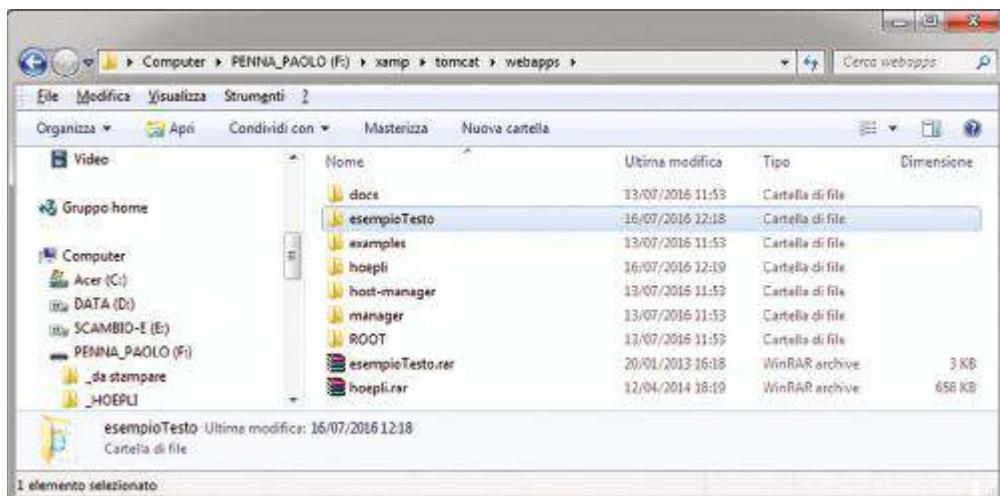
Path	Version	Display Name	Running	Sessions	Commands
/	None specified	Welcome to Tomcat	true	0	<a href="#">Start</a> <a href="#">Stop</a> <a href="#">Reload</a> <a href="#">Undeploy</a> <a href="#">Expire sessions with idle &gt; 30 minutes</a>
/docs	None specified	Tomcat Documentation	true	0	<a href="#">Start</a> <a href="#">Stop</a> <a href="#">Reload</a> <a href="#">Undeploy</a> <a href="#">Expire sessions with idle &gt; 30 minutes</a>
/esempioTesto	None specified	Primo esempio di utilizzo delle servlet	true	0	<a href="#">Start</a> <a href="#">Stop</a> <a href="#">Reload</a> <a href="#">Undeploy</a> <a href="#">Expire sessions with idle &gt; 30 minutes</a>
/examples	None specified	Servlet and JSP Examples	true	0	<a href="#">Start</a> <a href="#">Stop</a> <a href="#">Reload</a> <a href="#">Undeploy</a> <a href="#">Expire sessions with idle &gt; 30 minutes</a>
/hoepli	None specified	Esempi di Servlet e JSP	true	0	<a href="#">Start</a> <a href="#">Stop</a> <a href="#">Reload</a> <a href="#">Undeploy</a> <a href="#">Expire sessions with idle &gt; 2 minutes</a>

## ■ Deployment di una applicazione

Il **deploy** di un'applicazione è già stato descritto nella lezione 1 di teoria dove si sono analizzate tutte le sue componenti; rivediamo sinteticamente dove posizionare i file.

- Nella **root directory** dell'applicazione vengono collocati i file **\*.html**, **\*.jsp**.

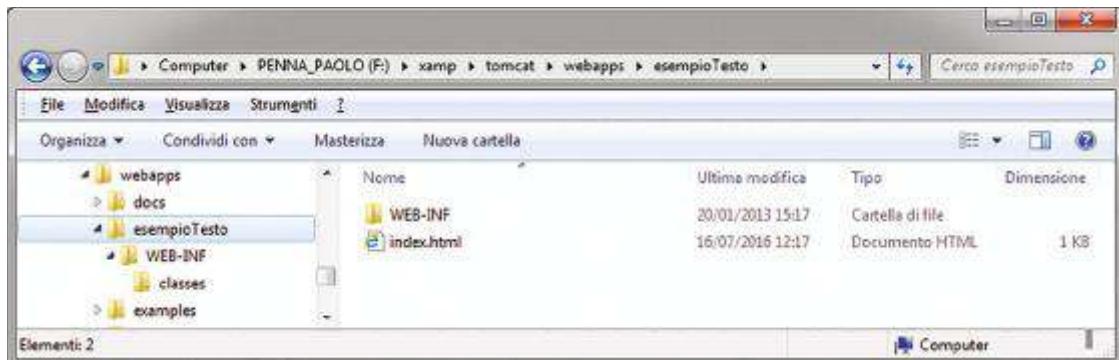
Tutte le applicazioni devono essere collocate nella **root directory**, la cartella **webapps**: nel nostro caso nella cartella `F:\xampp\tomcat\webapps` sono presenti tutte le nostre applicazioni che verranno organizzate ciascuna in una sottodirectory, tra le quali la cartella **esempioTesto** che contiene i file dell'esempio descritto nella lezione 1.



## 2 All'interno creiamo una cartella chiamata WEB-INF e WEB-INF\classes.

Inseriamo i file del nostro progetto:

- nella cartella inseriamo il file “di avvio” della applicazione, cioè **index.html**;
- nella sottodirectory **WEB-INF** il file **web.xml**, cioè il **Deployment Descriptor** analizzato nella lezione 2;
- in **WEB-INF/classes** i file **\*.class** delle **servlet** (bytecode), nel nostro esempio **EsempioTesto.class**;
- in **WEB-INF/lib** gli eventuali file di libreria **\*.jar**.



Dopo aver riavviato il server possiamo mandare in esecuzione l'applicazione dal browser digitando il percorso della root directory e, se tutto è andato a buon fine, verrà caricata la pagina **index.html** della nostra web application che manda in esecuzione produce il risultato della figura di destra.



Attenzione ai caratteri maiuscoli/minuscoli perché qui è tutto case-sensitive (in realtà sarebbe meglio abituarsi a gestire le maiuscole e le minuscole anche nei sistemi dove non incide).

Nella sezione **materiali** del **DVD** allegato al presente volume (oppure sul sito [www.hoepliscuola.it](http://www.hoepliscuola.it) riservato al presente volume) sono presenti tre file compressi:

- **esempioTesto.rar**
- **primaServlet.rar**
- **hoepli.rar**

Il primo contiene i file di questo esempio, il secondo un esempio di una servlet singola, mentre il terzo è il progetto completo con tutti gli esempi e alcune delle soluzioni degli esercizi proposti per le applicazioni **servlet** e **JSP**.

È sufficiente decomprimere il file nella cartella **webapps** per avere a disposizione la seguente **home page** all'indirizzo **localhost/hoepli** (oppure **localhost:8080/hoepli**): ►



La home page che abbiamo a disposizione contiene i link alle seguenti pagine:

Buongiorno (link diretto <i>Buongiorno.java</i> )	Esegui	Sorgenti
Buongiorno con pagina generata in un buffer ( <i>Esempio2.java</i> )	Esegui	Sorgenti
Configura ( <i>Configura.java</i> )	Esegui	Sorgenti
Informazioni sul sistema client-server ( <i>esercitazione</i> )	Esegui	Sorgenti

La seconda pagina raccoglie gli esempi per le **JSP** presentate nella lezione 2: i codici sorgente **Java** sono nel file **servlet\_JSP.rar** (una copia è anche presente nella directory **classes**).

Lezione 4		
La data di oggi ( <i>espressioni - oggi.jsp</i> )	Esegui	Sorgenti
Acquisti ( <i>dichiarazioni ed espressioni acquisti.jsp</i> )	Esegui	Sorgenti
Pagamento/ <i>scriptlet pagamento.jsp</i>	Esegui	Sorgenti



## Prova adesso!

- Realizzare una servlet
- Deployment di una applicazione

- 1 Crea una cartella dentro **TOMCAT/webapps** che conterrà la tua applicazione (per esempio: [primaServlet](#)).
  - 2 Crea una semplice pagina **index.html** dentro **webapps/PrimaServlet** con un link alla **servlet** come il seguente:
- ```
<p><a href="servlet/PrimaServlet">Per eseguire la servlet clicca qui!</a> </p>
```
- 3 Crea la cartella **WEB-INF** dentro la home-directory **PrimaServlet**.
  - 4 Scrivi il descrittore di deployment **web.xml** e mettilo dentro **WEB-INF**.
  - 5 Crea la cartella **classes** dentro **WEB-INF**.
  - 6 Copia in questa cartella il file **PrimaServlet.class**.
  - 7 Collauda la tua prima applicazione da un browser digitando:  
<http://localhost:8080/PrimaServlet>.

Ora realizza il codice **Java** della classe in modo che visualizzi i seguenti parametri:

```

19    out.println("<h3>Informazioni sul sistema client-server</h3>");
20    out.println("Porta del server : " +request.getServerPort()+"<BR>");
21    out.println("Carattere encoding : " +request.getCharacterEncoding()+"<BR>");
22    out.println("Metodo : " + request.getMethod()+"<BR>");
23    out.println("Indirizzo della richiesta : " + request.getRequestURI()+"<BR>");
24    out.println("Informazioni sul percorso : " + request.getPathInfo()+"<BR>");
25    out.println("Protocollo : " + request.getProtocol()+"<BR>");
26    out.println("Nome host remoto : " + request.getRemoteHost()+"<BR>");
27    out.println("Indirizzo host remoto: " + request.getRemoteAddr()+"<BR>");
```

Confronta la tua soluzione con quella riportata [PrimaServlet.java](#).

# ESERCITAZIONI DI LABORATORIO 2

## L'INIZIALIZZAZIONE DELLA SERVLET

Nella **servlet** che abbiamo descritto come primo esempio nella lezione precedente non abbiamo inserito il metodo che ne permette la sua inizializzazione, cioè **void init (ServletConfig config)**: tale metodo, ereditato direttamente dalla classe **GenericServlet** presente nel package **javax.servlet.http**, viene chiamato automaticamente all'atto del caricamento della **servlet** in memoria, cioè alla prima esecuzione della sua prima istanza, come avviene per il costruttore delle classi **Java**.

Se la **servlet** non riesce a completare l'inizializzazione viene generata una eccezione.

### Metodo **init()**

Il metodo **init()** riceve come parametro un oggetto di tipo **ServletConfig** che contiene la configurazione iniziale di una **servlet**.

```

5 public class ContaAccessi extends HttpServlet
6 {
7     int conta;
8     public void init(ServletConfig config)
9     {
10         conta = 0;
11     }
12     public void doGet(HttpServletRequest req, HttpServletResponse res)
13         throws IOException, ServletException
14     {
15         res.setContentType("text/html");
16         PrintWriter out = res.getWriter(); // definizione dell'output
17         out.println("<html><body>Volte " + conta ++ + "</html></body>");
18         out.close();
19     }
20 }
```

Nel sovrascrivere **init()** è importante ricordare di richiamare **init()** della classe padre con il metodo **super.init()**, come riportato in seguito, dove abbiamo rettificato il codice precedente: senza questo invito al super, il codice all'interno della classe genitore **GenericServlet** non funziona e si possono avere dei problemi.

```

8     public void init(ServletConfig config)
9         throws ServletException
10    {
11        super.init(config);
12        conta = 0;
13    }
14 }
```



## Prova adesso!

- Inizializzazione di una servlet
- Configurazione web.xml

Completa la **servlet** posizionandola in una cartella a piacere di **webapps** e aggiornando di conseguenza il corrispondente file **web.xml**; successivamente modifica il codice per fare in modo che il valore del contatore non venga azzerato a ogni nuova esecuzione del motore delle **servlet**.

## ■ Configurazione di una servlet

Molte applicazioni hanno come esigenza comune quella di avere la possibilità di “parametrizzare” il funzionamento dell'applicazione stessa senza dover riscrivere il codice.

Molteplici sono i possibili vantaggi, come quello di:

- ▶ adattare il funzionamento dell'applicazione con piccole modifiche di alcuni fattori;
- ▶ personalizzare il funzionamento per i vari clienti/utenti.

Supponiamo di scrivere una **servlet** per gestire una scuola: naturalmente la stessa **servlet** potrà essere riusata per scuole diverse, dato che queste hanno molte cose in comune e il “riutilizzo del codice” è la prima “legge dello sviluppatore”.

Dobbiamo però gestire alcune piccole differenze tra le varie installazioni, come ad esempio personalizzare il *nome della scuola*, senza dover modificare il codice della **servlet**.

Se mettiamo nella classe della **servlet** una variabile d'istanza con il nome della scuola:

```
5 public class Scuola extends HttpServlet
6     String intestazione="ITIS Fibonacci";
```

Sicuramente risolviamo il problema in modo “elementare” ma per una scuola differente saremo costretti a modificare il codice **Java** e ricompilarlo e dovremo tenere una versione per ogni installazione che faremo ed effettuare eventuali rettifiche a tutte le copie, una alla volta!

Una soluzione migliore è quella di mettere tutti i parametri in un file che andrà letto prima di iniziare a gestire le richieste dei **client**.

Le **API** delle **servlet** forniscono un formato di file apposito e delle funzionalità specifiche che permettono di accedere al contenuto dei parametri senza bisogno di conoscere la struttura del file e/o effettuare complesse operazioni di parsing ma semplicemente accedendo a delle semplici proprietà di oggetti **Java**.

Sostanzialmente ci sono due possibilità per passare dei parametri di inizializzazione a una **servlet**:

- a. lettura di **file ResourceBundle**;
- b. lettura di parametri costanti di configurazione **ServletConfig** dal file **web.xml**.

### Utilizzo di file ResourceBundle

Un file **ResourceBundle** è un file di testo di tipo **.properties** che viene associato a una applicazione **Java** che mette a disposizione la classe astratta **java.util.ResourceBundle** che consente di definire delle risorse contenenti le informazioni utili a localizzare un'applicazione.

Un file **ResourceBundle** contiene le informazioni nella forma **chiave/valore** dove la chiave identifica uno specifico elemento.

L'utilizzo tipico dei **ResourceBundle** è quello che permette di gestire le diverse lingue in una applicazione: per esempio un file **Resources.properties** conterrà i valori di default per tutte le stringhe dell'applicazione e un file **Resources\_fr\_FR.properties** per la versione francese ecc.

La tecnica del file **ResourceBundle** può essere utilizzata anche per inizializzare la **servlet** (che però noi non utilizzeremo).

## Lettura di parametri tipo `ServletConfig`

Le **servlet** possono essere configurate aggiungendo particolari righe nella pagina **web.xml**; è possibile definire coppie **chiave/valore** che vengono lette dall'oggetto **ServletConfig** semplicemente con i metodi:

- ▶ `getInitParameterNames()`
  - ▶ `getInitParameter(String)`

Per aggiungere uno o più parametri “costanti” di **inizializzazione** nella **servlet** è sufficiente aggiungere per ciascuno di essi la seguente coppia di **meta-tag**:

```
<init-param>  
<param-name>nomeParametro</param-name>  
<param-value>valoreParametro</param-value>  
</init-param>
```

## ESEMPIO

Rendiamo parametrico il `nomeScuola`, la `citta` e il `colore` desiderato per visualizzarla aggiungendo tre coppie di parametri nel file `XML` e in ciascuna ne inizializziamo un valore:

```
93 < servlet >
94     < servlet-name > Scuola </ servlet-name >
95     < servlet-class > Scuola </ servlet-class >
96     < init-param >
97         < param-name > citta </ param-name >
98         < param-value > Como </ param-value >
99     </ init-param >
100    < init-param >
101        < param-name > nomeScuola </ param-name >
102        < param-value > ITIS Alan Turing </ param-value >
103    </ init-param >
104    < init-param >
105        < param-name > colore </ param-name >
106        < param-value > #FF0000 </ param-value >
107    </ init-param >
108 </ servlet >
109
110 < servlet-mapping >
111     < servlet-name > Scuola </ servlet-name >
112     < url-pattern > /servlets/servlet/Scuola </ url-pattern >
113 </ servlet-mapping >
114
115
```

Per accedere ai propri parametri di configurazione la **servlet** utilizza l'interfaccia **ServletConfig** che viene passata come parametro al metodo **init()**: mediante il suo metodo **getInitParameter()** è possibile ottenere il valore di un parametro in base al nome col quale è stato definito nel file **XML**:

```
String getInitParameter(String nomeParametro)
```

### ESEMPIO

Completiamo la definizione del metodo **init()** del nostro esempio dopo aver definito due variabili globali che ci serviranno per memorizzare i parametri letti con **getInitParameter()**:

```
11 public void init(ServletConfig config)
12     throws ServletException
13 {
14     super.init(config);
15     citta = config.getInitParameter("citta");
16     nomeScuola = config.getInitParameter("nomeScuola");
17     colore = config.getInitParameter("colore");
18 }
```

Ora i parametri sono nelle due variabili e possono essere utilizzati all'interno della **servlet** per definire per esempio il titolo della pagina **HTML** di risposta:

```
37 // genera codice html;
38 out.println(PAGE_TOP);
39 out.println("<h3><font color=" + [colore] + ">" + nomeScuola + "</font></h3>");
40 out.println("<h3> di " + [citta] + "</h3>");
41 out.println(PAGE_BOTTOM);
```

Si può accedere ai parametri di configurazione anche tramite il metodo **getServletConfig()** che inoltre può essere invocato in qualunque momento, non necessariamente solo all'interno di **init()**.

Completiamo la **servlet** e la mandiamo in esecuzione ottenendo:



**AREA digitale**  
Netbeans, Eclipse e il file web.xml



### Prova adesso!

- Configurare una servlet
- Definizione parametri

- 1 Modifica il file **XML** cambiando il valore dei parametri e quindi rimanda in esecuzione al **servlet**.
- 2 Successivamente cancella i tag di definizione, una riga alla volta, osservando volta per volta cosa accade all'out rieseguendo la servlet.  
Osservazione: ricordati di riavviare sempre **Tomcat**!
- 3 Modifica l'esempio **Configura1.java** aggiungendo la chiamata di **getServletConfig()** per leggere il contenuto delle variabili di inizializzazione all'interno della **servlet**.

# ESERCITAZIONI DI LABORATORIO 3

## L'INTERAZIONE TRA CLIENT E SERVLET GET/POST CON LE SERVLET

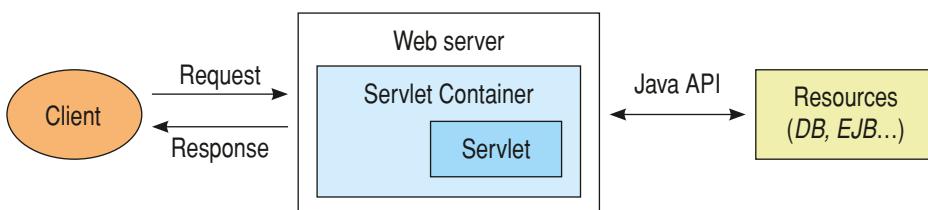
### ■ Servlet con passaggio dei parametri

In questa lezione di laboratorio ci occuperemo del **passaggio di parametri** alla **servlet**.

Oltre ai parametri di configurazione letti dal file **web.xml** mediante l'oggetto **ServletConfig** le **servlet** devono acquisire i parametri dal client e comportarsi di conseguenza.

Lo schema di interazione tra **client** e **server** è il seguente:

- 1 il **client** fa una richiesta **HTTP** al **Web server** (**request**);
- 2 il **Web server** carica la **servlet** (solo la prima volta) e crea un **thread** per eseguirla;
- 3 il **container** esegue la **servlet** richiesta;
- 4 la **servlet** genera la **risposta** (**response**);
- 5 la **risposta** viene restituita al **client**.



Le **form** del **client** hanno sostanzialmente due modalità di invio dei dati alla **servlet**:

- agganciati alla **URL** come query string (**GET**);
- inseriti nel body del pacchetto **HTTP** (**POST**).

Ogni **servlet** estende la classe **HttpServlet** per gestire il protocollo **HTTP** implementando i metodi definiti per gestire l'interazione **HTTP** col **client** che si ritengono necessari, che sono:

- **doGet**, per gestire richiesta di tipo **GET** o **HEAD**;
- **doPost**, per gestire richiesta di tipo **POST**.

Tali metodi ricevono tutti 2 argomenti:

- l'oggetto **HttpServletRequest** per la gestione dei dati ricevuti dal **client**;
- l'oggetto **HttpServletResponse** per la gestione dei dati da inviare al **client**.

A seconda del metodo **HTTP** utilizzato per la richiesta la **servlet** accede ai dati nei seguenti modi:

- con tutti i metodi **HTTP** viene utilizzato il metodo **getParameter(<parametro>)** che ritorna una stringa col contenuto del parametro indicato. Sono anche disponibili i metodi **getParameterNames()** e **getParameterValues()** che rispettivamente forniscono i nomi dei parametri e un array che ne contiene i valori.

La loro sintassi è la seguente:

```
String getParameter(String)      // ritorna il valore del parametro dal suo nome
Enumeration getParameterNames() // ritorna una enumerazione dei parametri
String[] getParameterValues()   // ritorna una array di valori del parametro
```

- col metodo **GET** si può utilizzare il metodo **getQueryString()** che restituisce una stringa che però deve successivamente essere analizzata per individuare i nomi dei parametri e i rispettivi valori.

```
String getQueryString()           // ritorna la query string completa
```

### ESEMPIO

Come primo esempio scriviamo una pagina **HTML** come quella della figura seguente, dove mediante due **textbox** inviamo alla **servlet** due campi, **nome** e **cognome**, utilizzando il metodo **GET**.



Nella **servlet** implementiamo il metodo **doGet()** e leggiamo i parametri con il metodo **getParameter()** e le seguenti istruzioni:

```
11 public void doGet(HttpServletRequest request, HttpServletResponse response)
12 throws IOException, ServletException
13 {
14     response.setContentType("text/html");
15     PrintWriter out = response.getWriter();    // oggetto di comunicazione
16     // i parametri nome e cognome vengono passati dal browser
17     String nome = request.getParameter("nome");
18     String cognome = request.getParameter("cognome");
```

Con queste variabili realizziamo la pagina **HTML** da inviare come risposta al **client**:

```
41 // contenuto della pagina HTML
42 out.println("Ciao "+nome+" "+cognome+"!</p>");
43 out.println("<b>La data di oggi e': " + new java.util.Date()+"</b></p>");
```

A schermo il risultato è il seguente:



Analogo risultato lo si ottiene utilizzando il metodo **POST**.



## Prova adesso!

- Passaggio parametri con doPost()
- Utilizzo metodo doGet()

Modifica la pagina **Servlet6.html** e la classe **Form1.java** per effettuare le medesime operazioni utilizzando il metodo **POST** e scrivendo il metodo **doPost()**.

A video devi ottenere il seguente risultato:



dove osserviamo che non è presente la **query string** a differenza del caso precedente.

Senza dover riscrivere due metodi, cioè **doGet()** e **doPost()**, per accettare entrambe le possibilità offerte dal client **HTML** è possibile scrivere nella **servlet** solo uno di essi e richiamarlo dentro l'altro: per esempio, dopo aver scritto il metodo **doPost()** lo richiamiamo nel metodo **doGet()** in questo modo:

```
49 public void doGet(HttpServletRequest request, HttpServletResponse response)
50     throws IOException, ServletException
51 {
52     doPost(request, response);
53 }
```



## Prova adesso!

- Passaggio parametri con doPost()
- Passaggio parametri con doGet()

Aggiungi alla classe che hai appena realizzato il metodo sopra scritto e richiama la classe sia mediante **POST** che mediante **GET** in modo da verificarne il funzionamento. Successivamente aggiungi dei **radiobutton** come in figura:



e visualizza sullo schermo il piatto prescelto:



Infine memorizza i risultati in un file in modo da poter effettuare un sondaggio via web sottoponendo la domanda a tutti i tuoi compagni di classe e visualizzando i risultati con una pagina simile alla seguente:



## ■ Parametri passati dal client con query string diretta

È anche possibile passare dei parametri al momento della chiamata della **servlet**, cioè accodandoli alla linea di comando, mediante la sintassi:

```
http://.../NomeServlet?nomeParametro=Paolo
```

Dove:

- **NomeServlet** è il nome della **servlet**;
- **nomeParametro** è il nome del parametro utilizzato nella classe;
- **Paolo** è il valore attuale del parametro.

### ESEMPIO

Nel nostro esempio la **servlet** si chiama **Direttal**, il nome del parametro **para1** e il valore attuale **Paolo**; la attivazione viene effettuata con la seguente riga di comando:

```
http://localhost:8080/hoepli/servlets/servlet/Direttal?para1=Paolo
```

In alternativa alla digitazione dell'istruzione nel browser, può essere comodo avviare le **servlet** **Configura1** mediante un richiamo all'interno di una pagina **HTML** col seguente comando:

```
11  <p>Richiama la classe <font color="#FF0000">Direttal.class</font></p>
12  <p><a href="servlet/Direttal?para1=Paolo">Per eseguire la servlet clicca qui!</a></p>
13
```

Nella **servlet** viene definito come nel caso precedente il metodo `doGet()` che legge il parametro `para1`:

```

41 public void doGet(HttpServletRequest request, HttpServletResponse response)
42     throws IOException, ServletException
43 {
44     // il parametro para1 viene passato dal browser accodato con ?para1=Paolo
45     String nomeUtente = request.getParameter("para1");

```

che viene visualizzato semplicemente con:

```

52 // contenuto della pagina HTML
53 out.println(saluti+" "+nomeUtente+"!");

```

Il risultato è quello riportato di seguito.



I parametri sono stati passati dalla pagina **HTML** con il **GET**, e quindi “ben visibili” nella stringa di **query**.

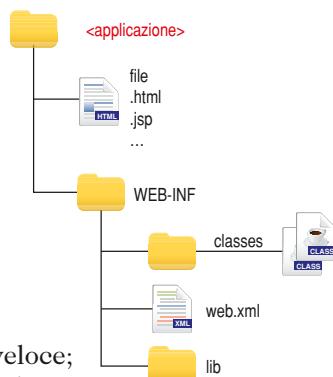
## ■ Il file.war (web archives)

Una applicazione web può essere memorizzata in un unico file compreso di estensione **.war** (**Web Archives**) che permette di trasferire semplicemente le applicazioni web dal team di sviluppo al **web server** per essere pubblicata in Internet, memorizzandola nella cartella **webapps** di **Tomeat**.

All’interno del file **.war** saranno presenti tutti i file del progetto memorizzati secondo l’albero descritto dalla “Servlet 2.4 specification” che riportiamo a lato: ►

Per crearli vi sono due possibilità:

- creare un archivio **.zip** e quindi rinominarlo, soluzione semplice e veloce;
- utilizzare il programma **jar.exe** (presente nella directory **bin** dell’ambiente **Java**) con il seguente comando.



```

C:\Java8\bin>jar
Usa: jar {xtxui}[fmnOPMe] [jar-file] [manifest-file] [entry-point] [-C dir] file ...
Ozioni:
  -c crea un nuovo archivio
  -t visualizza l'indice dell'archivio
  -x estrae i file con nome (o tutti i file) dall'archivio
  -u aggiorna l'archivio esistente
  -v genera output commentato dall'output standard
  -f specifica il nome file dell'archivio
  -m include informazioni manifest dal file manifest specificato
  -n esegue normalizzazione Pack200 dopo la creazione di un nuovo archivio
  -e specifica il punto di ingresso per l'applicazione stand-alone
    inclusa nel file jar eseguibile
  -B solo memorizzazione; senza compressione ZIP
  -P conserva i componenti iniziali '/' (percorso assoluto) e '\..\\' (directo
ry padre) dai nomi file
  -M consente di non creare un file manifest per le voci
  -i genera informazioni sull'indice per i file jar specificati
  -C imposta la directory specificata e include il file seguente
Se un file è una directory, verrà elaborato in modo ricorsivo.
Il nome del file manifest, del file di archivio e del punto di ingresso devono
essere specificati nello stesso ordine dei flag 'm', 'f' ed 'e'.

```

Un esempio di esecuzione è il seguente:

```
F:\xampp\tomcat\webapps>jar -cvf secondoEsempio.war miaapp2/
aggiunto manifest
aggiunta in corso di: miaapp2</in = 0> <out = 0><memorizzato 0%>
aggiunta in corso di: miaapp2\index.html<in = 332> <out = 211><compresso 36%>
aggiunta in corso di: miaapp2\WEB-INF</in = 0> <out = 0><memorizzato 0%>
aggiunta in corso di: miaapp2\WEB-INF\classes</in = 0> <out = 0><memorizzato 0%>

aggiunta in corso di: miaapp2\WEB-INF\classes\EsempioTesto.class<in = 1177> <out =
= 635><compresso 46%>
aggiunta in corso di: miaapp2\WEB-INF\classes\EsempioTesto.java<in = 948> <out =
= 419><compresso 55%>
aggiunta in corso di: miaapp2\WEB-INF\web.xml<in = 730> <out = 333><compresso 54%
>
```



◀ Il **manifest file** è un file di testo non formattato che contiene alcune informazioni utili a definire le proprietà di un archivio **JAR**: per esempio, in esso viene indicato quale file contiene il metodo main e il class path da usare per l'esecuzione del **JAR**. ▶

Ora basta collocare il file nella **root** di **Tomeat** (NON nella directory **webapps**!) e automaticamente, grazie ai dati contenuti nel ◀ **manifest file** ▶, la nostra applicazione verrà configurata.



## Prova adesso!

- Creazione file .war
- Utilizzo manifest file

- 1 Realizza una semplice applicazione web dove una pagina **HTML** richiama una **servlet** (per esempio **Login1.class**) mediante un pulsante e una routine **Javascript**, come nell'esempio riportato in figura, per inviare i dati di accesso a un'area riservata:

Esempio di utilizzo delle servlet con queryString

Inserisci il tuo nome e la password sul pulsante per inviare i dati alla servlet login

amministratore  
  
Submit Reimposta

Nota: viene restituita una pagina HTML generata dalla servlet login1.java

- 2 Realizza un file **.war** che contiene tutti i file e lascia sul **server** solo tale file, rimuovendo le directory che hai utilizzato per la realizzazione della applicazione.
- 3 Successivamente effettua un controllo per consentire o meno l'accesso confrontando i dati presenti in un file **autorizzazioni.txt** e modificando la pagina di risposta in base al risultato.

# ESERCITAZIONI DI LABORATORIO 4

## LA PERMANENZA DEI DATI CON LE SERVLET: I COOKIE

### ■ I cookie con le servlet

Quasi tutte le applicazioni web sono composte da più di una pagina e, nel soddisfare le richieste dello stesso **client**, è utile “ricordarsi” delle precedenti pagine che sono state visitate o dei parametri che abbiamo fatto scegliere all’utente per personalizzare la sua visita.

Tra i dati che dobbiamo ricordare è l’avvenuto accesso a una sezione riservata, in modo da non richiedere le credenziali “a ogni successiva pagina” che il nostro visitatore richiede.

Sappiamo che il protocollo **HTTP** è **stateless**, cioè dopo una richiesta effettuata dal **client** e una risposta soddisfatta dal **server** si “riparte da capo”, cioè l’interazione **client-server** viene “dimenticata” e si passa a considerare la richiesta successiva.

Abbiano alcune alternative per risolvere il problema, tra le quali ricordiamo:

- ▶ inserire informazioni inerenti allo stato direttamente negli **URL**, accodandoli come stringa di **query**: questa soluzione, oltre che a essere complessa è intrinsecamente insicura;
- ▶ aggiungere campi nascosti (**hidden**) nella pagina **HTML**, ma anche questa soluzione è insicura;
- ▶ utilizzare dei **cookie**, che vedremo in questa esercitazione;
- ▶ utilizzare il meccanismo delle **sessioni**, che vedremo nella prossima esercitazione.

### I cookie

I cookie sono dei piccoli file di testo (che possono anche essere limitati a una lunghezza di 4096 byte) che vengono gestiti e memorizzati dal **browser**, quindi dal **client**.

Un **browser** è tenuto ad accettare al massimo 20 **cookie** per sito e 300 in totale per ogni utente: dato che i **cookie** vengono inviati nell’**header HTTP**, questi devono essere creati soltanto prima che qualsiasi altro contenuto sia inviato al **client**.

**Server** e **client** si scambiano l’informazione presente nei **cookie** inglobandola negli header del protocollo **HTTP**: il **browser** allega i **cookie** scaricati da un dato **host** a **ogni** richiesta effettuata da questo **host**.

I **cookie**, oltre che ad essere usati dal motore delle servlet per gestire le sessioni e quindi risolvere tutti quei problemi legati alla sessione descritte nella prossima esercitazione, possono servire per:

- ▶ customizzazione di un sito in base alle pagine precedentemente visitate;
- ▶ pubblicità mirata alle pagine precedentemente visitate;
- ▶ eliminazione di username e password per l’accesso al sito.

## I cookie in Java

In Java l'API delle **servlet** fornisce la classe `javax.servlet.http.Cookie` che ci consente di utilizzare e manipolare facilmente i **cookie**.

Un **cookie** è un'istanza della **classe Cookie** che ha come costruttore:

```
Cookie (String nome, String valore)
```

Un **cookie** viene trasmesso dal **server** al **client** come oggetto di output, ovvero viene associato all'oggetto risposta (OUTPUT), quindi per creare un **cookie** si utilizza il metodo `addCookies()` dell'interfaccia `HttpServletResponse`, che prende in ingresso un oggetto `Cookie`.

A ogni **cookie** viene associato un **nome** e mediante il costruttore è possibile assegnargli un **valore**:  
 ▶ il **nome** deve rispettare le regole dettate in [RFC 2109](#), cioè può contenere solo caratteri ASCII alfanumerici esclusi virgole, punti e virgola e spazi, inoltre non può iniziare col simbolo "\$"; nelle [API Java](#), il nome del cookie non può essere cambiato dopo la sua creazione;  
 ▶ il **valore** del cookie non ha vincoli sul contenuto.

### ESEMPIO

Vediamo un semplice esempio di creazione di un **cookie** che memorizza il colore preferito dell'utente:

```
9  Cookie mioCookie = null;           // definisco un cookie
10 myCookie = new Cookie("colore", "blue"); // creo l'oggetto e lo inizializzo
11 response.addCookie(mioCookie);
```

Un **cookie** viene trasmesso dal **client** al **server** come oggetto di input, ovvero viene associato all'oggetto richiesta (INPUT), quindi per leggere i **cookie** si utilizza il metodo `getCookies()` dell'oggetto `HttpServletRequest`, che restituisce un array di oggetti `Cookie`.

Non è possibile accedere direttamente al singolo **cookie**, ma il metodo `getCookies()` ritorna tutti i **cookie**. Li mettiamo in un vettore e lo scorriamo fino a individuare il **cookie** desiderato con questo segmento di codice:

```
27  Cookie[] cookies = request.getCookies();
28  for(int i = 0; i < cookies.length; i++)
29      out.println(cookies[i].getName() + " " + cookies[i].getValue()+"<br>");
```



**Zoom su...**

### PROPRIETÀ DEI COOKIE

È possibile aggiungere degli attributi ai **cookie** per stabilirne la validità, sia in termini di tempo che in termini di dominio o di percorso, o semplicemente per associarli un commento; i metodi sono i seguenti:

```
void setComment (String s) e String getComment()
void setDomain (String s) e String getDomain()
void setMaxAge (int secondi) e int getMaxAge()
```

Per esempio:

```
13  myCookie.setComment("colore di sfondo");    // commento
14  myCookie.setDomain ("hoepli.com");           // dominio
15  myCookie.setPath ("/");                     // percorso
16  myCookie.setMaxAge(900);                   // durezza in secondi (15 minuti)
```

Vediamo un secondo esempio dove effettuiamo l'inserimento dei dati dal **client** e la registrazione dei **cookie** da parte del **server**.

## AREA digitale



Metodi principali classe Cookie

### ESEMPIO

Scriviamo una  **servlet** che visualizza una semplice pagina **HTML** che richiede all'utente due valori, il primo che verrà utilizzato come nome del nuovo **cookie**, e il secondo che ne costituirà il valore.

```

78  out.print("<form action=\"\"");
79  out.println("Cookie!\" method=POST>");
80  out.print("Nome  Cookie: ");
81  out.println("<input type=text length=20 name=nameCookie><br>");
82  out.print("Valore Cookie: ");
83  out.println("<input type=text length=20 name=valoreCookie><br>");
84  out.println("<input type=submit></form>");
```

La pagina generata è la seguente:



La  **servlet** legge i due parametri nel metodo **doPost()** che richiama **getParameter()** con le seguenti istruzioni e definisce un oggetto del tipo **Cookie**:

```

42  String nomeCookie = request.getParameter("nomeCookie");
43  String valoreCookie = request.getParameter("valoreCookie");
44
45  Cookie aCookie = null;
```

In questo oggetto andiamo a memorizzare i valori ricevuti come parametro e lo aggiungiamo all'oggetto **response** in modo da settare il nuovo **cookie**:

```

46  // controlla i parametri passati dal metodo doPost e doGet
47  if(nomeCookie != null && valoreCookie != null){
48      aCookie = new Cookie(nomeCookie, valoreCookie);
49      response.addCookie(aCookie);
50  }
```

Dall'oggetto **request**, con il metodo **getCookies()** andiamo a “caricare” i **cookie** che sono stati definiti nel **client** e, nel caso ce ne fosse qualcuno settato, li visualizziamo con un ciclo:

```

59  Cookie[] cookies = request.getCookies();
60  if((cookies != null) && (cookies.length > 0)){
61      out.println("Il tuo browser ha settato i seguenti cookies <br><br>");
62      for (int i = 0; i < cookies.length; i++){
63          Cookie cookie = cookies[i];
64          out.print ("Nome  Cookie : " + (cookie.getName())+ "<br>");
65          out.println("Valore Cookie : "+ (cookie.getValue())+ "<br><br>");
66      }
67  }else{
68      out.println("non ci sono cookies settati nel tuo browser");
69  }
```

Ai quali aggiungiamo l'ultimo **cookie** settato, quello che abbiamo appena ricevuto dalla form:

```

71 if(aCookie != null){
72     out.println("<P>");
73     out.println("Questo è l'ultimo cookie che hai settato:<br><br>");
74     out.print("NomeCookie : " + (nomeCookie) + "<br>");
75     out.print("Valore Cookie : " + (valoreCookie));
76 }

```

Mandiamo in esecuzione la **servlet**, inseriamo ad esempio come nome **cookie** “**colore**” e come valore “**rosso**” e inviamo la query; a schermo avremo il seguente risultato:



Aggiungiamo ora un secondo **cookie** (a piacere) e confermiamo con [**Invia query**] ottenendo la seguente schermata, dove sono visibili entrambi i cookie che abbiamo settato:



## Prova adesso!

- Creazione dei cookies
- Utilizzo dei cookies

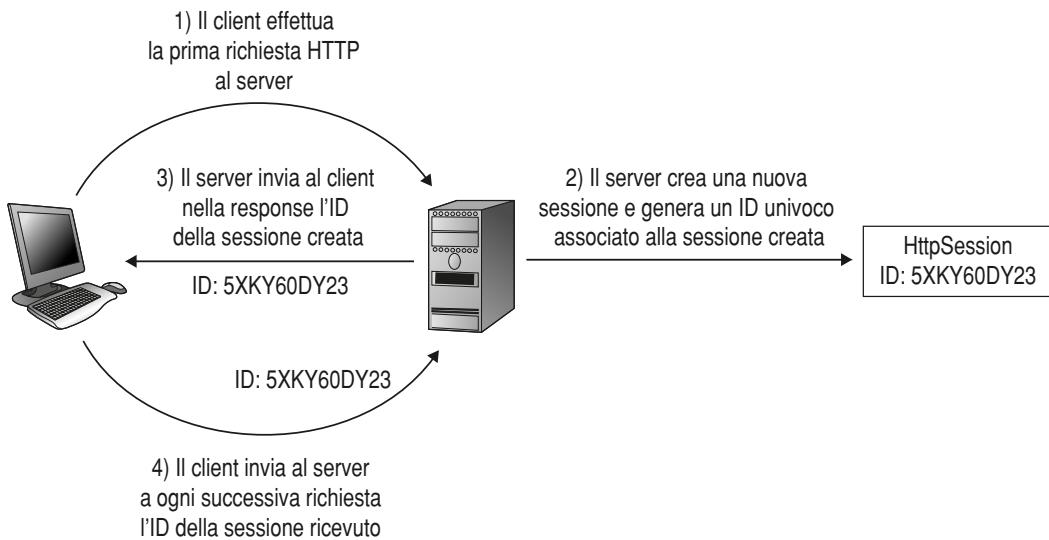
Aggiungi gli opportuni controlli sull'input in modo da evitare gli errori dovuti a valori non corretti inseriti nei campi da parte dell'utente. Quindi aggiungi una **list box** contenente il nome dei possibili colori che l'utente può selezionare come sfondo: memorizza il colore come **cookie** e utilizzalo come sfondo per le successive pagine che verranno trasmesse al **client**.

# ESERCITAZIONI DI LABORATORIO 5

## LA PERMANENZA DEI DATI CON LE SERVLET: LE SESSIONI

Oltre ai **cookies** un secondo metodo per effettuare la “permanenza dei dati” tra **client** e **server** è quello delle **sessions**. La prima differenza sostanziale tra **cookies** e **sessions** è che con le **sessions** i dati sono memorizzati sul **server** e non sul **client** mentre sul **client** viene memorizzato solo un ID di **sessione** (sempre attraverso un **cookie**).

Il funzionamento è il seguente: al primo collegamento da parte di un utente a una **servlet** che supporta le **sessions** viene creato un oggetto in cui è possibile memorizzare le informazioni di interesse e gli viene assegnato un identificatore unico che viene inviato al **client** e memorizzato in un **cookie**.



A ogni successiva richiesta che viene effettuata dal **client** questo trasmette al **server** anche l'**ID della sessione** in modo che il **server** possa riconoscere l'utente e recuperarne i dati che ha precedentemente salvato.

**Il client** invia solo l'**ID di sessione**, quindi si riducono le informazioni scambiate fra **client** e **server**, si riduce il traffico in rete con benefici sia per le prestazioni che per la sicurezza.

Se il **browser** non supporta il **cookie**, il numero di **sessione** deve essere registrato nelle **query string** al momento della creazione del codice della pagina, come nell'esempio seguente:

```
http://localhost/servlet/miaServlet?sessionId=69163392288624
```

## Sessioni Java

Le API di Java mettono a disposizione la classe `javax.servlet.http.HttpSession` per definire le sessioni: in particolare il metodo `getSession()` permette di creare e restituire il contenuto della sessione.

Quando viene invocato questo metodo, per esempio con

```
HttpSession sessione = request.getSession();
```

abbiamo due possibili situazioni:

- non esiste una sessione corrente per questo **client** questa viene creata e il **server** genera un **Session ID** univoco per quel particolare **client**;
- è già presente l'**ID** di sessione, vengono caricati i dati relativi a quel **client**.

Una volta ottenuto l'oggetto **session**, questo può essere utilizzato tramite i suoi metodi per memorizzare qualsiasi tipo di informazione: in pratica la creazione di una **sessione** comporta la predisposizione sul **server** di un'area di memoria per la gestione di tutte le informazioni.

Per gestire le **sessioni** sono disponibili i seguenti due gruppi di metodi:

- per avere informazioni sulla sessione (ID della sessione, tempo trascorso, tempo di inattività)

```
String getID()          // ritorna l'ID della sessione
long getCreationTime() // ritorna il time stamp della creazione della sessione
long getLastAccessedTime() // ritorna il time stamp dell'ultimo accesso alla sessione
void invalidate()      // interrompe la validità della sessione
```

- per memorizzare dati di sessione nella forma di coppia **chiave = valore**, dove per evitare conflitti si segue la convenzione di dare un nome agli oggetti secondo lo schema:

<nome-applicazione>.<nome-oggetto>

```
void setAttribute(String chiave, Object valore) // aggiunge un oggetto
Object getAttribute(String chiave) // recupera un oggetto dalla sessione
void removeValue(String chiave)    // elimina un oggetto dalla sessione
```

Dato che le informazioni sono memorizzate sul **server** e che il numero dei **client** è numeroso, è necessario introdurre un meccanismo che permetta al **server** di liberare spazio quando la sessione non è più utilizzata; è possibile utilizzare un timer alla cui scadenza il container la elimina e per impostare il timeout si utilizza il seguente metodo:

```
public void setMaxInactiveInterval (int timeout)
```

oppure definendolo direttamente nel descrittore di deployment `web.xml`:

```
10 <session-config>
11   <session-timeout>15</session-timeout> <!-- in minuti -->
12 </session-config>
```

In questo modo la sessione scade dopo 15 minuti.

Si può anche invalidare esplicitamente la sessione richiamando dalla **servlet** il metodo `invalidate()`.

### ESEMPIO

Come primo esempio inseriamo in una **servlet** un contatore di richieste, cioè una variabile che conta quante volte la **servlet** è stata mandata in esecuzione. Creiamo la **sessione** alla prima esecuzione della **servlet** (o la ricarichiamo alle successive esecuzioni) e “leggiamo dalla sessione” l’oggetto desiderato, per esempio `Sessioni1.mioContatore`.

```

39 // crea una sessione se questa non esiste
40 HttpSession sessione = request.getSession();
41 // recupera e setta il valore del contatore
42 Integer tanti = (Integer) sessione.getAttribute("Sessioni1.mioContatore");

```

Rispettiamo la convenzione per il naming della variabile di attribuire un nome composto dal nome della classe (**Sessioni1**) seguito dal **"."** e dal nome della variabile (**mioContatore**).

Analizziamo il valore dell'oggetto **tanti** per incrementarne il valore; lo confrontiamo con **null**, che è il valore che contiene nel caso in cui la **servlet** viene eseguita per la prima volta e l'oggetto non è stato ancora definito, altrimenti ne incrementiamo il valore:

```

58 // alla fine il nuovo valore viene salvato
59 if (tanti ==null)                                // se non esiste
60 {
61     tanti = new Integer(1);                      // ne crea uno nuovo
62     output.println("E' la prima volta che viene eseguita la servlet</P> ");
63 }
64 else  // altrimenti
65 {
66     tanti = new Integer(tanti.intValue()+1);    // lo incrementa
67     output.print("La servlet e' stata eseguita nr. volte : " );
68     output.println(tanti);
69 }
70 sessione.setAttribute("Sessioni1.mioContatore", tanti);

```

Con l'istruzione 61 salviamo il valore dell'oggetto **tanti** nella **sessione** prima creata.

Mandando in esecuzione la **servlet** e richiamandola alcune volte otteniamo la seguente situazione:



Per vedere il nome del **cookie** che viene memorizzato sul **client** e l'identificatore della **sessione** appena creata possiamo utilizzare la **servlet** realizzata nella esercitazione precedente:

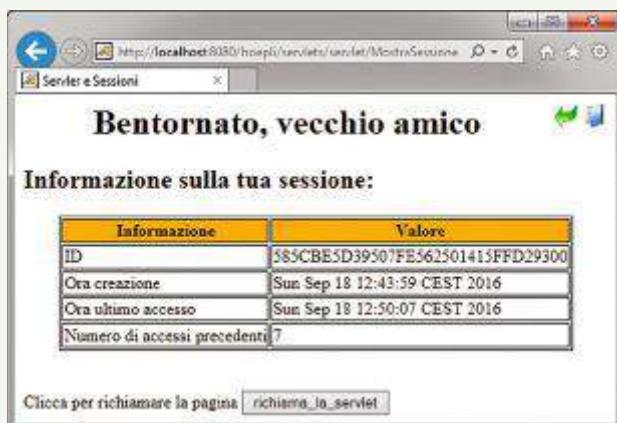




## Prova adesso!

- Creazione sessioni
- Utilizzo dati di sessione

- 1 Completa l'esempio modificando la **servlet Sessioni1.java** aggiungendo la data del primo accesso (e quindi di creazione della **sessione**) e modificando a 3 minuti il tempo di "expired".
- 2 Verificane il funzionamento richiamando la **servlet** dopo 5 minuti.
- 3 Quindi realizza una **servlet** che visualizza quante volte un utente durante la sua sessione di lavoro ha avuto accesso alla **servlet**, modificando in base ad essa la scritta di saluto, come ad esempio.



Una possibile codifica la puoi trovare nel file **MostraSessione.java**

## ■ Un esempio completo: memorizziamo il linguaggio di programmazione

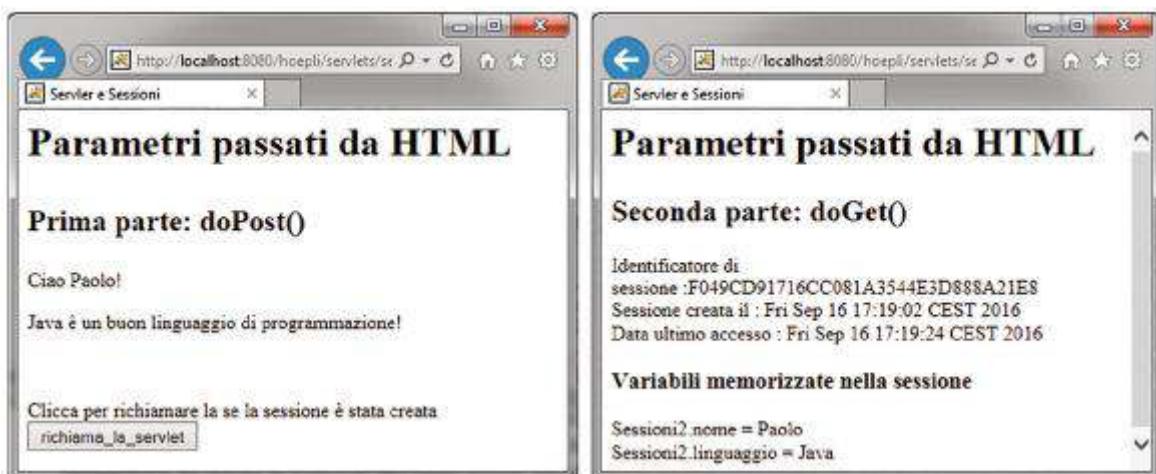
Realizziamo ora una **servlet** più complessa, dove l'utente inserisce il proprio nome e seleziona da una list box un linguaggio di programmazione, come dalla seguente pagina **HTML**:



Con il metodo **POST** inviamo la form alla **servlet** che memorizza le due variabili in una sessione:

```
ss // aggiungiamo i parametri alla sessione
56 session.setAttribute("Sessioni2.linguaggio", linguaggio);
57 session.setAttribute("Sessioni2.nome", nome);
```

Viene visualizzata una prima schermata dove vengono visualizzate in una form le scelte dell'utente che vengono memorizzate in una sessione: mediante il postback la form richiama se stessa col metodo **GET** e visualizza le informazioni sulla sessione e sulle variabili.



## Prova adesso!

- Creazione sessioni
- Utilizzo dati di sessione

- Completa l'esempio aggiungendo altre due caselle di testo e inserendo un ciclo che visualizza tutte le variabili di sessione mediante un enumeratore; quindi predisponi due stringhe così fatte:

```
private final static String names[] = {"Pascal", "C", "Java", "Visual Basic"};
private final static String isbn[] = {"978-88-203-48229", "978-88-203-48243", "978-88-203-48236", "978-88-203-34727"};
```

- A ogni linguaggio che viene selezionato visualizza il corrispondente codice ISBN del libro suggerito, come nella seguente schermata:



**AREA** *digitale*



Codifica del Session ID nei link

# ESERCITAZIONI DI LABORATORIO 6

## JDBC E MySQL

### ■ Il driver JDBC

Per poter connettere una **servlet** a un **database** è necessario per prima cosa caricare il **driver** specifico per il **database** che si desidera utilizzare: in questa esercitazione effettueremo la connessione a **MySQL**, che dispone sia di un driver **JDBC** sia di un driver **ODBC**.

**MySQL** è sicuramente il più famoso e diffuso **DBMS Open Source** ed è disponibile per tutte le piattaforme. Per poter effettuare le nostre prove è necessario che **MySQL** sia presente sul nostro **PC**: è già compreso nel pacchetto **XAMP/LAMP**, così come è anche presente nel pacchetto **EasyPHP** un secondo prodotto **Open Source** che contiene l'intero ambiente di sviluppo costituito da **Apache**, **MySQL**, **PHP**.



In alternativa il download può essere eseguito gratuitamente partendo dall'indirizzo <http://www.mysql.com/>, ma in tal caso per connettere **Java** al database **MySQL** è necessario scaricare il driver **JDBC** chiamato **Connector/J** che può essere prelevato dalla pagina <http://dev.mysql.com/downloads/connector/j/> oppure dalla cartella **materiali** nel **DVD** allegato al presente volume.

### AREA digitale

[Configurazione manuale di JDBC](#)

### ■ Predisposizione dell'archivio MySQL

La connessione a un database richiede necessariamente la presenza del database **MySQL** per poter effettuare il test del codice **Java**. Per creare un archivio di prova ci si può connettere con il **DBMS** dalla linea di comando e digitare manualmente le istruzioni per la creazione del database, oppure utilizzare la comoda interfaccia offerta da **XAMPP**, che utilizza **phpMyAdmin**, o installare un pacchetto specifico per la gestione e l'amministrazione dei database: **HeidisSQL**, che è un prodotto **Open Source** che può essere scaricato direttamente da <http://www.heidisql.com/> oppure dalla cartella **materiali** nel **DVD** allegato al presente volume.

Creiamo un database **amici** e all'interno di esso la tabella **provejava** come in figura:

Cognome	Nome	Indirizzo	Citta	Sesso
Rossi	Mario	Via Milano,12	Roma	M
Gialli	Rina	Via Napoli,12	Bari	F
Verdi	Gino	Via Roma,14	Torino	M
Bianchi	Filippo	Via Torino,34	Milano	M

La tabella può essere creata manualmente oppure mediante la seguente **query**:

```
1 CREATE TABLE `amici` (
2   `Cognome` CHAR(50) NULL DEFAULT NULL,
3   `Nome` CHAR(50) NULL DEFAULT NULL,
4   `Indirizzo` CHAR(50) NULL DEFAULT NULL,
5   `Citta` CHAR(50) NULL DEFAULT NULL,
6   `Sesso` CHAR(1) NULL DEFAULT NULL
7 )
```

Analogamente la tabella può essere popolata manualmente o mediante la **query**:

```
INSERT INTO `amici` (`Cognome`, `Nome`, `Indirizzo`, `Citta`, `Sesso`)
VALUES ('Gialli', 'Rina', 'Via Napoli, 23', 'Bari', 'F');
```

## ■ Connessione al database

Scriviamo ora una prima **servlet** che visualizza il contenuto della tabella **amici** che abbiamo appena creato nel database **provajava**. Estraiamo i dati con la semplice query seguente:

```
6 // definizione della query
7 String miaQuery = "SELECT Cognome, Nome, Indirizzo, Citta FROM amici";
```

Si vuole ottenere sullo schermo qualcosa di simile alla seguente schermata:

cognome	nome	indirizzo	citta
Rossi	Mario	Via Milano,12	Roma
Gialli	Rina	Via Napoli,12	Bari
Verdi	Gino	Via Roma,14	Torino
Bianchi	Filippo	Via Torino,34	Milano
Bartolazzi	Pino	via del lago,4	Milano
Bianchi	Elena	Via Pascoli,23	Roma

Inseriamo nella **servlet** un insieme di costanti necessarie per definire i parametri di connessione alla tabella, a partire dall'URL fino alla password che ci permette di accedere ai dati:

```

9  private final String url      = "jdbc:mysql://";
10 private final String serverName = "localhost";
11 private final String portNumber = ":3306/";
12 private final String databaseName= "provejava";
13 private final String userName   = "root";
14 private final String password   = "";
15 // riferimento al database: connessione MySql
16 private final String URL_mioDB = url + serverName + portNumber + databaseName;
17 // definizione del driver per la connessione al DB MySQL
18 private final String DRIVER = "com.mysql.jdbc.Driver";

```

Dopo aver creato l'oggetto che ci permette di effettuare la connessione, eseguiamo di seguito:

```

71 // apro la connessione verso il database.
72 connessione = DriverManager.getConnection(URL_mioDB, userName, password);
73 // ottengo lo Statement per interagire con il database
74 Statement statement = connessione.createStatement();
75 // interrogo il DBMS mediante una query SQL
76 ResultSet resultSet = statement.executeQuery(miaQuery);

```

Se tutto è andato a buon fine i dati sono ora presenti nel **resultSet** e con un semplice ciclo li visualizziamo sullo schermo ottenendo quanti ci eravamo proposti: il codice completo è nel [file ServletBD2.java](#).

```

90 out.println("<b>cognome"+c#9;+"nome"+c#9;+"indirizzo"+c#9;+"citta</b>");
91 while (resultSet.next())
92 {
93     String cognome = resultSet.getString(1);
94     String nome = resultSet.getString(2);
95     String indirizzo = resultSet.getString(3);
96     String citta = resultSet.getString(4);
97     out.println(cognome+c#9;+"nome"+c#9;+"indirizzo"+c#9;+"citta");
98 }

```

## ■ Interrogazione del database

Vediamo ora una **servlet** per interrogare il database estraendo i nominativi in base al sesso; il parametro da passare alla **servlet** viene selezionato da una pagina **HTML** simile alla seguente:



In base al valore del parametro scelto viene definita la stringa di query:

```

68 // predisponiamo la query in base al parametro scelto
69 String scelta = request.getParameter( "scelta" );
70 String query = "SELECT * FROM Amici where sesso='"+scelta+"' ";

```

Selezionando *maschi* si ottiene:

The screenshot shows a web browser window with the URL `http://localhost:8080/hoepcli/servlets/servlet/ServletDB5?scelta=M`. The title bar says "Estrazione in base al sesso della tabella amici del database MySQL provajava". The content area displays the query "Sesso selezionato = M" and the result of the SQL query "SELECT \* FROM Amici where sesso='M'". The result is a table with columns: cognome, nome, indirizzo, citta. The data rows are:

cognome	nome	indirizzo	citta
Rossi	Mario	Via Milano,12	Roma
Verdi	Gino	Via Roma,14	Torino
Bianchi	Filippo	Via Torino,34	Milano
Pini	Pino	via del lago,4	Milano

Selezionando *femmine* si ottiene:

The screenshot shows a web browser window with the URL `http://localhost:8080/hoepcli/servlets/servlet/ServletDB5?scelta=F`. The title bar says "Estrazione in base al sesso della tabella amici del database MySQL provajava". The content area displays the query "Sesso selezionato = F" and the result of the SQL query "SELECT \* FROM Amici where sesso='F'". The result is a table with columns: cognome, nome, indirizzo, citta. The data row is:

cognome	nome	indirizzo	citta
Gialli	Rina	Via Napoli,12	Bari



## Prova adesso!

- Connessione a MySQL
- Interrogazione mediante query

- 1 Scrivi una pagina **HTML** che selezioni da una list box una città ed effettui la richiesta a una **servlet** che interroghi il database prima creato selezionando gli amici che abitano in quella città.

The screenshot shows a web browser window with the URL `http://localhost:8080/hoepcli/servlets/servletDB6.html`. The title bar says "Esempio di Servlet con query". The content area displays the title "Servlet con sessioni e query parametrica (database MySQL)". Below it is the instruction "Selezione la città desiderata per l'elenco dei tuoi amici." followed by a dropdown menu set to "Torino". At the bottom are two buttons: "Submit" and "Reimposta". A note at the bottom states: "Nota: viene restituita una pagina HTML generata dalla servlet ServletDB6.java".

Una possibile esecuzione è la seguente:

cognome	nome	indirizzo	città
Verdi	Sina	Via Roma,14	Torino

- 2 Successivamente aggiungi alla scelta della città di residenza anche la possibilità di selezionare solo maschi o femmine.
- 3 Come ultima applicazione, aggiungi una form che permetta di inserire i dati di nuovi amici e scrivi una servlet che effettui l'inserimento dei dati nella tabella **provejava** popolando la tabella con almeno 15 record.
- 4 Quindi completa la servlet di ricerca scritta in precedenza facendo in modo che, nel caso in cui non fossero presenti nell'archivio record che soddisfino la ricerca, venga richiamata la pagina che ne permette l'inserimento.

Un backup degli archivi popolati con gli esempi descritti nel libro è presente nella cartella:



Basta effettuare l'importazione all'interno di **MySQL** con l'operazione [**Importa**] e selezionando il file opportuno tramite [**Sfoglia**]:



# ESERCITAZIONI DI LABORATORIO 7

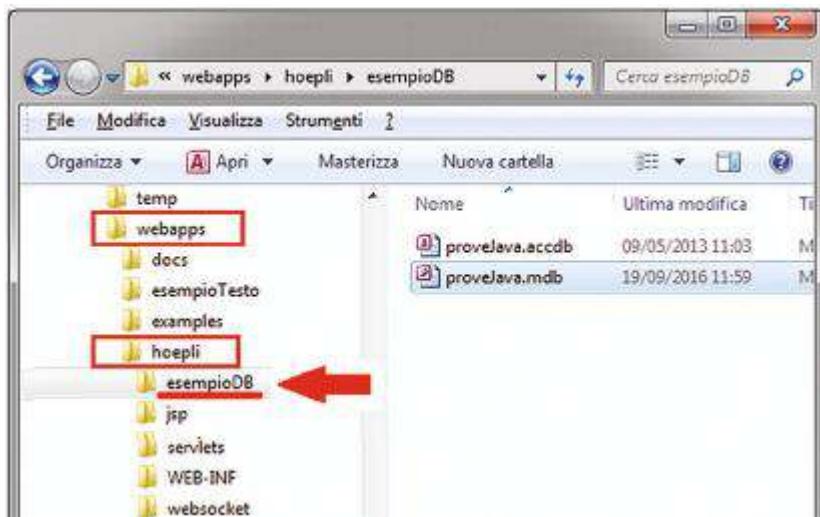
## SERVLET E DATABASE MDB CON PARAMETRI

### ■ Premessa

In questa esercitazione replicheremo quanto fatto per MySQL nella lezione precedente realizzando con Access un file **proveJava.mdb** che contenga una tabella **Amici**, che andiamo a realizzare e popolare con dati simili a quelli riportati in figura:

Cognome	Nome	Indirizzo	Città	Sesso	Eta	Sposato
Bianca	Maria	via Como, 4	Bari	F	15	<input type="checkbox"/>
Bianchi	Filippo	Via Torino, 32	Milano	M	20	<input type="checkbox"/>
Gialli	Pina	Via Napoli, 23	Roma	F	25	<input type="checkbox"/>
Rossi	Mario	Via Milano, 12	Roma	M	30	<input checked="" type="checkbox"/>
Verdi	Gino	Via Roma, 14	Torino	M	35	<input checked="" type="checkbox"/>
Bruni	Alice	Via Bari, 3	Napoli	F	27	<input checked="" type="checkbox"/>
Neri	Anna	Via Rodi, 43	Como	F	50	<input type="checkbox"/>
Otto	Ciro	Via Priva, 32	Ancona	M	28	<input checked="" type="checkbox"/>

Memorizziamo il file in un'apposita cartella del **web server**, per esempio come in figura:



Se stiamo utilizzando una versione di Java inferiore alla 8.0 possiamo utilizzare ODBC come meccanismo di connessione: a tal fine è necessario registrare il database con la apposita utility disponibile in **Windows**, solitamente nominata **Origine Dati (ODBC)**.

**AREA digitale**



Registrazione database in Windows mediante ODBC

Nella nostra trattazione per realizzare la connessione ad **Access** utilizzeremo le librerie **UCanAccess** (che sono una evoluzione/completamento delle librerie **jackcess**) che, di fatto, hanno sostituito universalmente il meccanismo basato su **ODBC**.

Inoltre la libreria **UCanAccess** consente di "svincolarsi" dal meccanismo **JDBC-ODBC** che se da una parte agli inizi sembrava molto comodo, con macchine a 64 bit, con **Java 8** installato oppure con la connessione a **DBMS** diversi ha portato molteplici problemi soprattutto con **recordset** di grosse dimensioni.

La loro installazione è immediata: basta copiare i file nella directory **lib** delle librerie Java.

## AREA digitale

 Installazione librerie UCanAccess

## ■ Passaggio dei parametri da HTML

Scriviamo una **servlet** che interroga il **database** selezionando, ad esempio, i nominativi presenti in base al sesso: il parametro da passare alla **servlet** viene selezionato da una pagina **HTML**, già scritta a pagina 257. Come accennato nella lezione 2 teorica definiamo nelle prime righe della **servlet** le stringhe necessarie alla connessione al database:

```

1 public class ServletDB3 extends HttpServlet
2 {
3
4     private final String protocollo = "jdbc:ucanaccess://"; // connessione alla libreria
5     private final String mdbpath = "tomcat/webapps/hoepli/esempioDB/"; // percorso relativo
6     private final String mdbName = "proveJava.mdb;memory=false"; // nome database
7     private final String user = ""; // nome utente
8     private final String psw = ""; // password
9
10    // riferimento al database: connessione ODBC
11    private final String URL_mioDB = protocollo + mdbpath + mdbName;
12
13    // definizione del driver per la connessione al DB Access
14    private final String DRIVER = "net.ucanaccess.jdbc.UcanaccessDriver";
15
16
17

```

Cerchiamo di effettuare la connessione che renderà attivo all'interno del nostro programma l'apposito driver in un costrutto **try ... catch**:

```

22    // carico il driver per la connessione al DB Access
23    try{
24        Class.forName(DRIVER);
25    }catch (ClassNotFoundException e1) // il driver è caricato
26    {
27        System.out.println("Driver non trovato...");
28        System.exit(1);
29    }

```

Analogamente proviamo ad effettuare la connessione al **database Access**:

```

47    // definisco un oggetto per la connessione
48    Connection connessione = null;
49    try{ // apro la connessione verso il database.
50        connessione = DriverManager.getConnection(URL_mioDB);
51    }catch (Exception e){ // gestione dell'errore
52        System.out.println("Connessione al database non riuscita!");
53        System.exit(1);
54    }

```

La **servlet** legge ora il parametro "scelta" col metodo **doGet()** e definisce la stringa di interrogazione:

```

56    String scelta = request.getParameter( "scelta" );
57    out.println("<h4>Sesso selezionato = "+scelta+"</h4>");
58
59    // definizione della query
60    String query = "SELECT * FROM Amici where Sesso='"+scelta+"'";
61
62    if (scelta.equals("M"))
63        query = "SELECT Cognome, Nome, Indirizzo, Citta FROM Amici";
64

```

La query viene eseguita con la seguente istruzione, in modo da averne i risultati nell'oggetto `resultSet`:

```

63 try{
64     // ottengo lo Statement per interagire con il database
65     Statement statement = connessione.createStatement();
66     // interrogo il DBMS mediante una query SQL
67     ResultSet resultSet = statement.executeQuery(query);

```

Il risultato ritornato dalla `executeQuery` è tipo una `Enumeration`, i cui vari elementi sono le righe della tabella come ritornate dalla `query`, accessibili via via tramite `next` e quindi con un semplice ciclo li visualizziamo sullo schermo:

```

71     out.println("<b>cognome #9; nome #9; indirizzo #9; citta</b>");
72     while (resultSet.next()) {
73         String cognome = resultSet.getString(1);
74         String nome = resultSet.getString(2);
75         String indirizzo = resultSet.getString(3);
76         String citta = resultSet.getString(4);
77
78         out.println(cognome+"#9;"+nome+"#9;"+indirizzo+"#9;"+citta);
79     }

```

Se ad esempio nella pagina [HTML](#) selezioniamo *femmine*:



si ottiene:





## Prova adesso!

- Connessione ad Access
- Interrogazione mediante query

- 1** Scrivi una pagina **HTML** che selezioni da una list box una città ed effettui la richiesta a una servlet che interroga il database prima creato selezionando e visualizzando gli amici che abitano in quella città.

Esempio di Servlet con query

**Servlet con sessioni e query parametrica (database MDB)**

Selezione la città desiderata per l'elenco dei tuoi amici.

Como  Reimposta

Nota: viene restituita una pagina HTML generata dalla servlet **ServletDB4.java**

Una possibile esecuzione è la seguente:

Esempio di Servlet con sessioni e query parametrica (database MDB)

**Contenuto tabella amici del database provajava.mdb**

Città scelta = Torino

Query effettuata:

```
SELECT Cognome, Nome, Indirizzo, Città FROM Amici WHERE Città='Torino'
```

cognome	nome	indirizzo	città
Verdi	Gino	Via Roma, 14	Torino

- 2** Successivamente aggiungi alla scelta della città di residenza anche la possibilità di selezionare solo maschi o femmine.  
**3** Scrivi una nuova servlet che seleziona il campo che si vuole visualizzare combinato con l'età, come nell'esempio riportato di seguito:

Esempio query JDBC

**Servlet che accede ad un database Access**

Selezionare il campo su cui fare la query tra i seguenti:

Campo da visualizzare:

Eta inferiore a:

Una possibile esecuzione è la seguente:



- 4 Come ulteriore applicazione, aggiungi una form che permetta di inserire i dati di nuovi amici e scrivi una **servlet** che effettui l'inserimento dei dati nella tabella **provejava** popolando la tabella con almeno 15 record.
- 5 Quindi completa la **servlet** di ricerca scritta in precedenza facendo in modo che nel caso in cui non fossero presenti nell'archivio record che soddisfino la ricerca venga richiamata la pagina che ne permette l'inserimento.

Un copia del file **Access** è presente nella cartella **source** del **DVD** allegato al presente volume (file **esempioDB.rar**).

# 5

# Applicazioni lato server in Java: JSP

L1 JSP: Java Server Pages

L2 Java Server Pages e Java Bean

## Esercitazioni di laboratorio

- ① JSP: primi esempi di Java Server Pages; ② JSP: Java Server Pages con parametri da HTML; ③ JSP e database MySQL; ④ JSP e MDB; ⑤ JSP Bean e database

### Conoscenze

- Conoscere le caratteristiche delle pagine JSP
- Conoscere le caratteristiche delle Bean
- Acquisire le caratteristiche dell'interfaccia JDBC
- Conoscere i tipi di driver per la connessione ai database

### Competenze

- Realizzare una applicazione web
- Riconoscere i componenti di una pagina JSP
- Saper realizzare un'applicazione con JSP e Bean
- Generare un file.WAR

### Abilità

- Realizzare un'applicazione WEB dinamica con pagine JSP
- Richiamare una Bean in una pagina JSP
- Connettere le applicazioni web JSP con MySQL e Access
- Connettere i JSP Bean ai database

## AREA *digitale*



Esercizi



Esempio di codice Java generato da Tomcat: la classe oggi.jsp.java



Esempi proposti

Consulta il DVD in allegato al volume



Soluzioni

Puoi scaricare il file anche da hoeplieduca.it

# JSP: Java Server Pages

In questa lezione impareremo...

- le caratteristiche di una pagina JSP
- i componenti di una pagina JSP
- a realizzare una applicazione Web dinamica con pagine JSP

## ■ Le Java Server Pages (JSP)

La tecnologia **Java Server Pages (JSP)** è basata su **Java** e permette lo sviluppo di **applicazioni web** con contenuti dinamici senza dover essere legati a un ambiente in particolare.

Come con le **servlet**, anche con le **JSP** è possibile avere una netta separazione dei ruoli tra la presentazione grafica della pagina e la logica di gestione scritta appunto in **Java**.

Una pagina **JSP** è rappresentata da codice **HTML** con incapsulate al suo interno delle direttive **JSP**, racchiuse in un particolare **tag <%...%>**, rappresentanti codice **Java**: a differenza delle **servlet**, dove è necessario compilare il codice **Java** e collocare il file **.class** nella directory **classes** prima di iniziare il caricamento delle pagine della **wap**, con le pagine **JSP** la compilazione *avviene in automatico*, al momento del caricamento della pagina stessa.

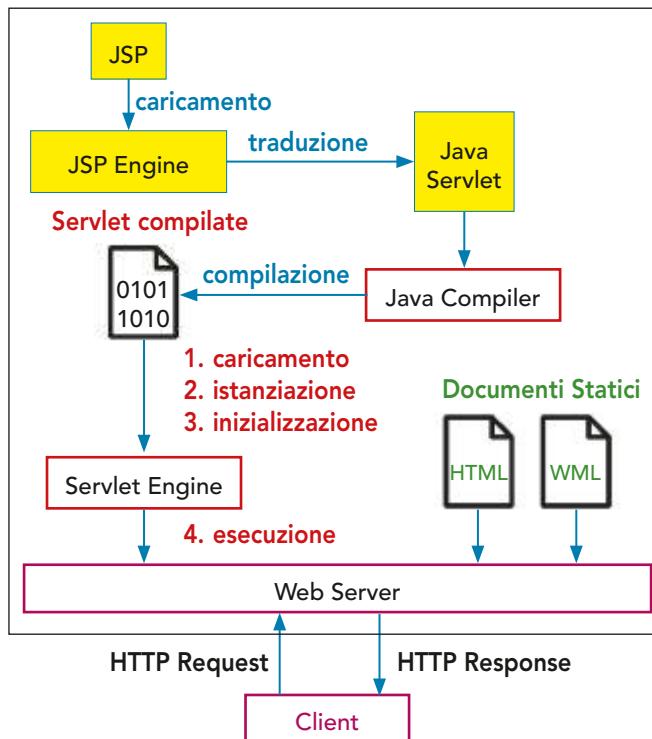
Infatti, alla prima richiesta di una pagina **JSP** il **server** effettuerà i seguenti passi:

- compila il codice **JSP** per ottenere il codice **Java** rappresentante una **servlet**; in pratica estrae i comandi **Java** e ne ricava il codice della **classe**;
- invia alla nuova classe la richiesta **HTTP** per ottenerne la risposta desiderata servendosi di un programma preprocessore che elabora la risorsa rimpiazzando i tag con codice **HTML**;
- spedisce infine la pagina risultante al **browser** del **client**.

Questo procedimento avviene solo alla prima richiesta di caricamento di una **JSP** dato che la classe prodotta viene mantenuta in cache dal **servlet engine** così da permettere una rapida risposta alle richieste successive.

La differenza sostanziale con le **servlet** è che queste sono una classe **Java** che utilizza il protocollo **HTML** per ricevere e inviare delle richieste tra **client** e **server**, mentre una pagina **JSP** è una pagina che racchiude sia del codice statico (**HTML**) che del codice dinamico (**Java**) che verrà eseguito sul **server**, secondo lo schema operativo delle pagine **PHP** o **ASP**.

Lo schema completo che riporta il meccanismo di funzionamento di una JSP è il seguente:



Naturalmente il **server** deve avere a disposizione il compilatore **Java** (**javac.exe**) per poter generare le classi: non basta il **JRE**, sufficiente per eseguire le **servlet**.

## Caratteristiche delle pagine JSP

La programmazione di pagine JSP è una “fusion” tra la programmazione dinamica in linguaggio PHP e quella effettuata mediante la realizzazione delle **servlet**: il programmatore JSP “pensa” come il programmatore PHP di inserire “frammenti” di codice di linguaggio di programmazione “immerosi” in tag di linguaggio nell’HTML ma è consapevole che alla sua prima esecuzione la pagina verrà trasformata in una **servlet** equivalente, e tale **servlet** può anche essere testata a parte.

È agevole lo sviluppo e la **prototipazione rapida** in quanto, essendo la compilazione automatica, basta apporre le modifiche e richiamare la pagina direttamente dal browser per vederne il risultato.

Inizialmente le **Web Application** si basavano solo sulle **servlet**, ma erano una tecnologia alla portata dei programmatori più esperti dato che in una classe **Java** bisognava inserire sia il codice gestionale che il codice che curava il layout della pagina Web; l’introduzione delle **JSP** è stata fatta per semplificare il lavoro dei **web designer** che potevano così dedicarsi al layout della pagina senza entrare in merito della logica di gestione dell’applicazione. Riducendo al minimo

le **scriptlet** `<% ... %>` e inserendo in essi anche richiami a **JavaBean** si sposta tutta la logica all’interno di essi in modo da separarla dalla grafica: in caso di modifiche estetiche viene aggiornato il template **HTML** dell’applicazione mentre in caso di modifiche logiche si modificano le righe **JSP**.



◀ Un **JavaBean** è un componente **Java** al 100% che opera all’interno di una qualsiasi macchina virtuale; è una classe **Java** che implementa l’interfaccia **java.io.Serializable** e utilizza metodi pubblici **get/set** per esporre gli attributi. Li descriveremo nella prossima lezione. ▶

Con questa premessa, una volta creata una pagina **JSP**, questa può essere mantenuta o modificata da chiunque conosca anche solo l'**HTML**.

## ■ Tag in una pagina JSP

In un pagina **JSP** le parti variabili sono contenute all'interno di tag speciali che possono avere due tipi diversi di sintassi:

- **Scripting-oriented tag;**
- **XML-oriented tag.**

### Scripting-oriented tag

Le **scripting-oriented tag** sono definite da delimitatori entro cui è presente lo scripting (*self-contained*) e possono essere così classificati:

- **commenti JSP:** hanno sintassi: `<%-- commento --%>` e vengono eliminati dal **container** in fase di compilazione/traduzione della pagina; i commenti **HTML** sono considerati come normale elemento di testo;
- **elementi di scripting:** codice **Java** (o altro *scripting language*) da inserire nella  **servlet** risultante:
  - **scriptlet:** con sintassi `<% code %>`; possono contenere variabili locali valide solo per una singola esecuzione della  **servlet** (prodotta in seguito alla compilazione della **JSP**);
  - **dichiarazioni:** con sintassi `<%! dichiarazioni [dichiarazioni]...%>`; sono metodi o variabili la cui durata è quella della  **servlet** (conservano il loro valore per tutte le esecuzioni della  **servlet**);
  - **espressioni:** con sintassi `<%= espressioni %>`; l'espressione viene valutata e scritta nella pagina di risposta della **JSP**;
- **direttive JSP:** con sintassi `<%@ direttiva %>`; controllano la struttura della  **servlet** risultante.

Inoltre sono presenti altri due tipi di istruzioni:

- **template text:** tutte le parti di testo che non sono definite come elementi **JSP** sono copiate nella pagina di risposta;
- **azioni JSP:** istruzioni specifiche che controllano il comportamento del motore **JSP**.

### XML-oriented tag

Le **XML-oriented tag** utilizzano come sintassi la notazione tipica delle pagine **XML** e permettono di definire i medesimi tipi di comandi dei tag **scripting-oriented**, e sono anche chiamate **Actions** o **Standard Actions**. Anch'esse contemplano i quattro tipi precedentemente elencati.

## ■ Tag scripting-oriented

### Espressioni

Vediamo un esempio di pagina **JSP** con **espressioni**: per valutare le espressioni **Java** si usano i delimitatori `<%=` e `<%>`: il risultato viene convertito in stringa e inserito nella pagina al posto del tag.

Scriviamo una semplice **JSP** che visualizzi data e ora correnti:

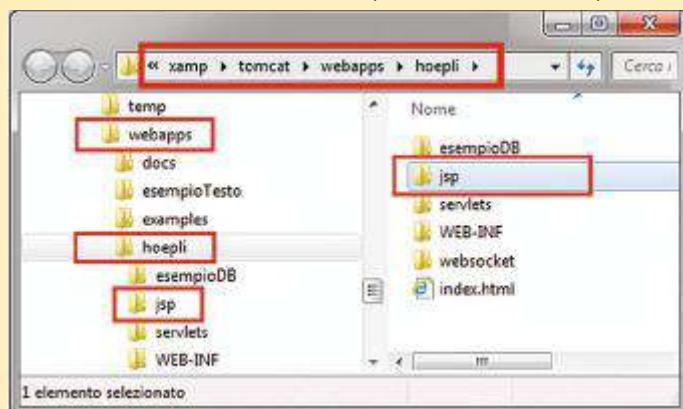
```

oggi.jsp
1  <HTML>
2  <BODY>
3  <TITLE> Primo codice JSP </TITLE>
4  Ciao, la data di oggi ègrave;
5  <%= new java.util.Date().toString() %> <!--espressione-->
6  </BODY>
7  </HTML>

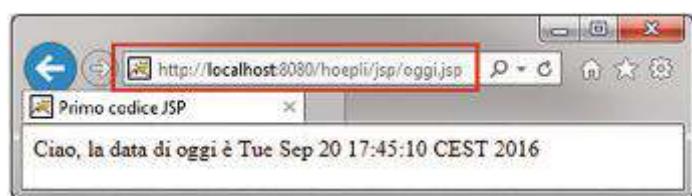
```

Tutti i nostri codici sorgenti verranno collocati nella cartella: **hoepli/jsp**.

Nella figura è possibile vedere la struttura completa dell'albero a partire del **web server**:



Lo mandiamo in esecuzione digitando nel browser semplicemente il nome della pagina (col suo percorso completo): nel browser avremo un output come quello della figura a lato. ►



## AREA digitale

 Esempio di codice Java generato da Tomcat: la classe `oggi_jsp.java`

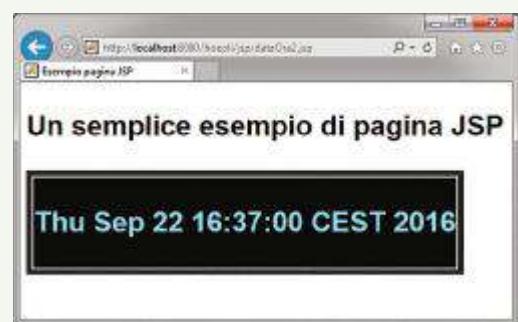
Come detto in precedenza, il **container Tomcat** ha convertito la pagina **JSP** in una **servlet** (classe `oggi_jsp.java`), ha compilato la classe generando il **bytecode** e lo ha mandato in esecuzione; il codice sorgente e la classe generata vengono memorizzate al seguente indirizzo:



## Prova adesso!

- Scrittura codice JSP
- Posizionamento file nel web server

- 1 Realizza la tua prima **JSP** modificando lo script di pagina precedente e inserendo nello stesso segmento di codice una parte di codice che migliora graficamente l'output, come ad esempio quello riportato in figura, magari definendo un foglio di stile.
- 2 Confronta la tua soluzione con quella riportata nel file `dataOra2.jsp` e visualizza la classe `dataOra2_jsp.java` generata da **Tomcat**.



## Dichiarazioni

Per dichiarare variabili e metodi si utilizzano i delimitatori `<%!` e `%>` che vengono poi referenziati in qualsiasi punto del codice **JSP**, richiamandoli all'interno di una espressione con `<%=` e `%>`.

La pagina **JSP** è trasformata in una classe **servlet** e i metodi così dichiarati diventano metodi della **servlet**: il **container** ne crea un'istanza e a ogni richiesta genera in corrispondenza un **thread**.

### ESEMPIO

Integriamo l'esempio precedente, su data e ora, con due dichiarazioni, rispettivamente di una stringa (*utente*) e di un metodo (`getTotale()`) e salviamolo nella pagina `acquisti.jsp`:



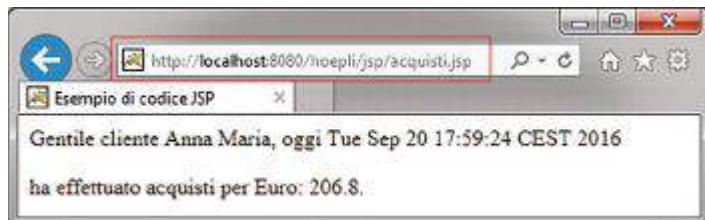
```

<HTML>
<BODY>
<TITLE> Esempio di codice JSP </TITLE>
<!-- dichiarazioni -->
<%
String utente = "Anna Maria";
double[] prezzi = {11.5, 73.8, 121.5};
double getTotale(){
    double totale = 0.0;
    for (int x = 0; x < prezzi.length; x++)
        totale += prezzi[x];
    return totale;
}
%>
<!-- espressioni -->
<P>Gentile cliente <%=utente%>, oggi <=% new java.util.Date().toString()%></P>
<P>ha effettuato acquisti per Euro: <%=getTotale()%>. </P>
</HTML>
</BODY>

```

Per meglio comprendere e distinguere i tag **HTML** dalle istruzioni e dichiarazioni **Java** si utilizza la convenzione di scrivere in MAIUSCOLO i primi e in minuscolo tutto il resto delle istruzioni.

Richiamando la pagina avremo il seguente output:



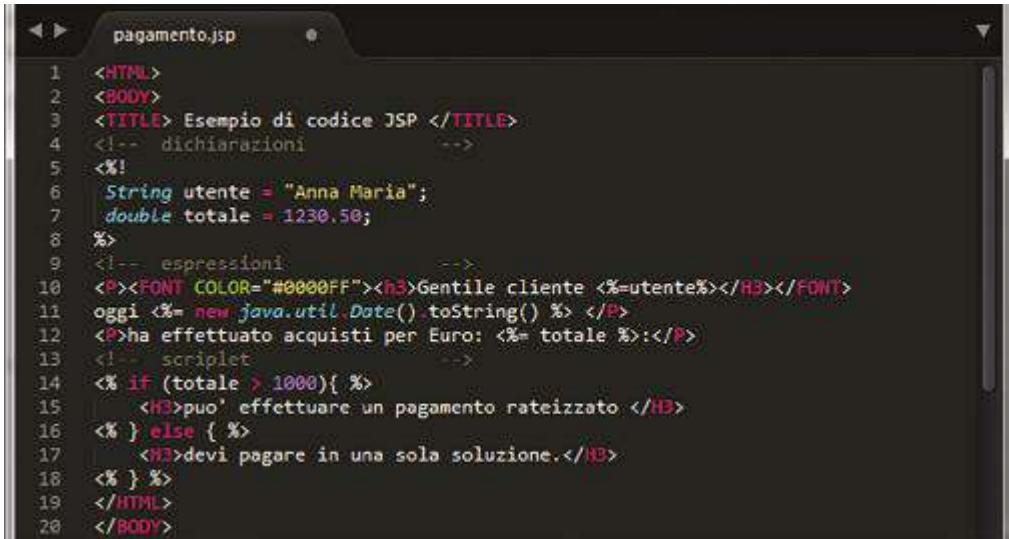
## Scriptlet

Nella pagina **JSP** è possibile inserire frammenti di codice **Java**, delimitati dal tab `<% e %>`, che prendono il nome di **scriptlet** e possono comprendere le istruzioni di controllo per la gestione del flusso di esecuzione del codice, quali la selezione e l'iterazione, tipiche dei linguaggi di programmazione: quando la pagina viene tradotta all'atto della sua esecuzione tutti gli **scriptlet** divengono un unico blocco di codice **Java**.

Vedremo che è anche possibile indicare con la direttiva `page` il linguaggio di programmazione per le `scriptlet`, che di default è `Java` ma potrebbe anche essere un qualunque linguaggio di scripting.

### ESEMPIO

Sempre nell'esempio precedente aggiungiamo una condizione sull'ammontare totale dell'importo che deve essere corrisposto per gli acquisti, permettendo un pagamento rateale se questo è superiore a 1000 Euro (file `pagamento.jsp`).

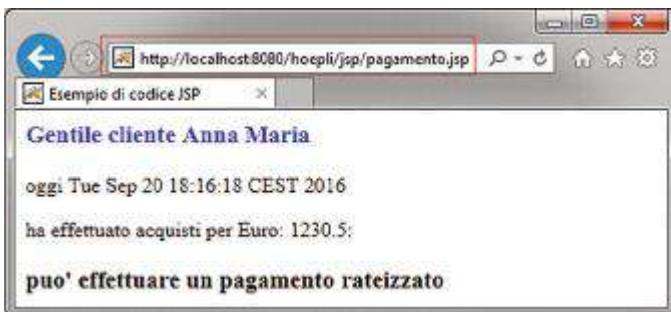


```

1 <HTML>
2 <BODY>
3 <TITLE> Esempio di codice JSP </TITLE>
4 <!-- dichiarazioni -->
5 <%!
6 String utente = "Anna Maria";
7 double totale = 1230.50;
8 %>
9 <!-- espressioni -->
10 <P><FONT COLOR="#0000FF"><H3>Gentile cliente <%=utente%></H3></FONT>
11 oggi <%= new java.util.Date().toString() %> </P>
12 <P>ha effettuato acquisti per Euro: <%= totale %>;</P>
13 <!-- scriptlet -->
14 <% if (totale > 1000){ %>
15   <H3>pou' effettuare un pagamento rateizzato </H3>
16 <% } else { %>
17   <H3>devi pagare in una sola soluzione.</H3>
18 <% } %>
19 </HTML>
20 </BODY>

```

Mandandolo in esecuzione avremo il seguente output:



The browser window displays the following content:

```

Gentile cliente Anna Maria
oggi Tue Sep 20 18:16:18 CEST 2016
ha effettuato acquisti per Euro: 1230.5;
pou' effettuare un pagamento rateizzato

```



**Zoom su...**

### OGGETTI IMPLICITI

Il blocco di codice specificato dalla `scriptlet` viene inserito all'interno del metodo `service()` della `servlet` corrispondentemente generata alla richiesta di esecuzione della pagina `JSP` (più precisamente all'interno del metodo `_jspService()`, richiamato da `service()`).

Avremo quindi implicitamente a disposizione i seguenti nove oggetti:

Oggetto	Classe/Interfaccia
page	javax.servlet.jsp.HttpJspPage
config	javax.servlet.ServletConfig
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
out	javax.servlet.jsp.JspWriter
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
pageContext	javax.servlet.jsp.PageContext
exception	Java.lang.Throwable

Descriviamoli sinteticamente uno per uno:

#### page

L'oggetto **page** rappresenta l'istanza corrente della **servlet**: ha come tipo l'interfaccia **HTTPJspPage** che discende da **JSPpage**, la quale a sua volta estende **servlet** e quindi può essere utilizzata per accedere a tutti i metodi definiti nelle **servlet**.

#### config

Contiene la configurazione della **servlet** (parametri di inizializzazione) e generalmente è lasciato inutilizzato, visto che le informazioni messe a disposizione da questo oggetto隐式 sono generalmente settate e rilevate in automatico. Mette a disposizione i seguenti due metodi:

- ▷ **getInitParameterName()**: restituisce tutti i nomi dei parametri di inizializzazione;
- ▷ **getInitParameter(name)**: restituisce il valore del parametro passato per nome.

#### request

È l'oggetto di **classe HttpServletRequest** che rappresenta la richiesta **HTTP** che ha portato all'attivazione della pagina **JSP/servlet**: fornisce i metodi di accesso alla request **HTTP** corrente.

#### response

È l'oggetto di **classe HttpServletResponse** che rappresenta la risposta **HTTP** da inviare al **client**.

#### out

È un oggetto di **classe JSPWriter.out** e rappresenta il flusso di output su cui viene prodotta la pagina **Web**: spesso il suo uso è implicito, ma all'occorrenza si può fare riferimento a questa variabile invocandone i metodi nella tabella allegata nella sezione **CLIL**.

#### session

È l'oggetto di **classe HttpSession** che rappresenta la sessione **HTTP** all'interno della quale è stata invocata la pagina **JSP**: fornisce le informazioni sul contesto di esecuzione della **JSP**, cioè della sessione corrente di esecuzione per l'utente.

#### application

È un oggetto che fornisce informazioni sul contesto di esecuzione della **JSP** e permette di accedere e di memorizzare gli oggetti per renderli accessibili da qualsiasi utente e modificabili da ogni pagina.

#### pageContext

È un oggetto di classe **PageContext**, che rappresenta l'insieme degli oggetti impliciti associati all'intera pagina: l'oggetto può essere trasferito da una pagina **JSP** a un'altra ma viene poco utilizzato per lo scripting.

#### exception

È un oggetto connesso alla gestione degli errori che viene utilizzato nelle Error Page, quelle che sono appositamente dichiarate con l'attributo **errorPage** impostato a **true**, e rappresenta l'eccezione che non viene gestita da nessun blocco **catch**.

**ESEMPIO**

Realizziamo un semplice contatore di accessi gestito con una **sessione**:

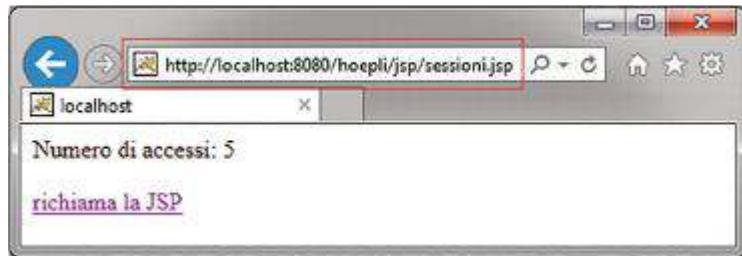


```

1 <%! Integer x = new Integer(0); %>
2 <%
3 if(session.isNew())                                // se non c'e' una sessione la crea
4 {
5   session.putValue("contatore", x);                  // mette x nella sessione
6 }
7 else
8 {
9   x = (Integer)session.getValue("contatore");        // legge dalla sessione
10  if(x == null)                                     // se x non e' presente
11    session.putValue("contatore", new Integer(1));    // lo inserisce con valore 1
12  else
13    session.putValue("contatore", new Integer(x.intValue() + 1));
14 }
15 >
16 <HTML>
17 <BODY>
18 <P>Numero di accessi: <%= x.intValue() %> </P>
19 <P><a href="sessioni.jsp">richiama la JSP</a></P>
20 </BODY>
21 </HTML>

```

Una sua esecuzione, dopo che viene richiamata la **JSP** alcune volte, ha il seguente output:

**Direttive**

Sono comandi **JSP** valutati a tempo di compilazione e le più utilizzate sono le seguenti:

- ▶ **include**: permettono di includere un altro documento;
- ▶ **page**: permette di importare **package**, dichiarare pagine d'errore, definire il modello di esecuzione della **JSP** relativamente alla concorrenza ecc;
- ▶ **taglib**: carica una libreria di custom tag implementate dallo sviluppatore.

**Direttiva include**

La sintassi della direttiva **include** è la seguente:

```
<%@ include file="nomeFile.xxx"%>
```

Permette di includere sia file statici (**HTML**) che altre **JSP**.

**ESEMPIO**

Di seguito è riportato un semplice esempio che utilizza la direttiva **include** per inserire una intestazione nella pagina **includi.jsp** richiamando una pagina **HTML** dove è presente una immagine:



```

1 <HTML>
2 <BODY>
3 <TITLE>JSP con direttiva include </TITLE>

```

```

4  <!-- dichiarazioni -->
5  <%!
6  String utente = "Anna Maria";
7  %>
8  <!-- direttiva include -->
9  <%@ include file = "intestazione.html"%>
10 <!-- espressioni -->
11 <P>Gentile cliente <%= utente %></P>
12 <P>La ringraziamo per aver visitato il nostro catalogo.</P>
13 </HTML>
14 </BODY>

```

Mandando in esecuzione la pagina JSP avremo il seguente output:



La direttiva `include` diventa indispensabile quando vogliamo creare dei metodi da utilizzare in più pagine, cioè per crearci una libreria di metodi/funzioni da utilizzare in ogni nostra pagina JSP.



## Prova adesso!

- Utilizzo direttiva `include`

- 1 In un file ([funzioni.jsp](#)) scrivi due metodi: il primo che riceve in input un testo e restituisce una stringa compatibile nel linguaggio [SQL](#) e il secondo metodo che dato un oggetto ci restituirà la sua trasformazione in stringa o nel caso di oggetto nullo una stringa vuota.
- 2 Scrivi poi la pagina che includerà e utilizzerà tale file ([postback.jsp](#)): questa deve permettere l'inserimento del nome e del cognome di un utente e quindi di inviare a se stessa tali dati dopo averli trasformati richiamando entrambe le funzioni prima definite.



Riassumendo possiamo dire che con la direttiva `include`, è come copiare il testo (sia statico che codice JSP) all'interno della pagina stessa. Il metodo è molto comodo evitando così la ridondanza di codice senza avere alcun effetto sulle prestazioni.

## Direttiva page

La sintassi della direttiva **page** è la seguente:

```
<%@page attributo="valore"%>
```

Permette di specificare alcune proprietà della pagina JSP che possono essere usate sia al momento della compilazione per costruire il corpo di alcuni metodi della **servlet** oppure in fase di **run time**. La seguente tabella riporta per ogni attributo la descrizione del suo effetto:

ATTRIBUTO	DESCRIZIONE
language="java"	Specifica il linguaggio usato negli scriptlet.
extends="classe"	Dichiarazione extends della servlet.
import="package" o "classe"	Dichiarazione di import della servlet.
session="true" o "false"	Supporto alle sessioni (default true).
buffer="none" o "nkb"	Dimensione del buffer di output (default 8kb).
autoFlush="true" o "false"	Flush automatico del buffer (default true).
isThreadSafe="true" o "false"	La pagina è thread-safe (possibili più thread) (default true).
info="testo"	Stringa di informazioni generiche.
errorPage="URL relativa"	Pagina da visualizzare in caso di errori.
contentType="tipo MIME"	Tipo MIME della pagina (default text/html).
isErrorPage="true" o "false"	True se questa è una pagina di errore.
pageEncoding="charset"	Encoding della pagina (default ISO-8859-1).

### ESEMPIO

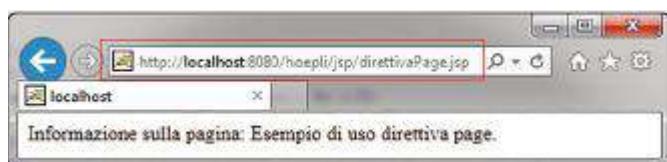
Per esempio, attribuiamo all'attributo **info** un valore:

```

1 <HTML>
2 <BODY>
3 <%@page info="Esempio di uso direttiva page."%>
4 <P>
5 Informazione sulla pagina: <%= getServletInfo()%>
6 </P>
7 </BODY>
8 </HTML>

```

L'esecuzione della **JSP** produce il seguente output:



## Direttiva taglib

Nelle **JSP** è possibile definire dei tag addizionali che possono essere usati nella pagina nell'**HTML** definendone il loro comportamento in apposite librerie, le **tag library**, che vengono importate con la direttiva **taglib**.

Sono disponibili **tag library** standard prodotte da terze parti e messe a disposizione della comunità **Java** oppure potete realizzare voi stessi le vostre librerie e riutilizzarle in tutte le vostre applicazioni.

La sintassi per includere una **tag library** è:

```
<%@taglib uri="libreria" prefix="prefisso" %>
```

Il prefisso indicato è quello che verrà utilizzato all'interno della pagina per specificare i tag presenti nella libreria (sarà il prefisso di ogni tag presente nella libreria).

### ESEMPIO

Se in una pagina JSP utilizziamo la **taglib** definita in **miaTLib.tld** che contiene per esempio la **classe SfondoBlu**, la definisco e la utilizzo nel modo seguente:

```
taglib.jsp
1 <%@taglib uri="/mialib.tld" prefix="mieiTag" %>
2 <HTML>
3 <HEAD><TITLE> Esempi di TAGLIB </TITLE></HEAD>
4 <BODY>
5 <mieiTag:sfondoBlu/>
6 </BODY>
7 </HTML>
```

Avviando l'esecuzione dello script, quando viene individuato questo tag, **Tomecat** richiama gli opportuni metodi dalla classe associata al nome **SfondoBlu** nel file **miaTLib.tld**.

## ■ XML-oriented tag o JSP Standard Actions

**JSP Standard Actions** sono tag XML specifici che influenzano l'esecuzione della pagina e l'output spedito al **client**. Ne riportiamo alcuni esempi:

### <jsp:useBean>

Associa un'istanza di un **JavaBean** a un ambito e ad un ID tramite una variabile di script dichiarata con lo stesso nome; se non trova un'istanza “in vita” cerca di crearla. Gli attributi più importanti legati a questa azione sono:

- **id**: rappresenta l'identità dell'istanza dell'oggetto all'interno dell'ambito specificato;
- **scope**: rappresenta l'ambito dell'oggetto, per esempio: **page, request, session, application**;
- **class**: rappresenta il nome di classe che definisce l'implementazione dell'oggetto.

```
<jsp:useBean id="myBean" scope="application" class="mypackage.MyBean" />
```

Sarà l'argomento descritto nella prossima lezione.

### <jsp:include>

Fornisce un modo per includere risorse aggiuntive, di tipo statico o dinamico, nella pagina JSP corrente. Gli attributi più importanti legati a questa azione sono:

- **page**: rappresenta la **URL** relativa della risorsa da includere;
- **flush**: questo attributo contiene un valore boolean che indica se il buffer debba o meno essere svuotato.

```
<jsp:include page="filename" />
```

### <jsp:forward>

Consente al motore **JSP** l'inoltro, all'atto dell'esecuzione, della risorsa corrente a una risorsa statica, a una **servlet** o a una pagina **JSP**. L'unico attributo possibile per questa azione è:

- **page**: rappresenta la **URL** relativa all'oggetto da inoltrare.

```
<jsp:forward page="url" />
```

## <jsp:param>

Viene impiegata per fornire informazioni a coppie di tag/valore, includendoli come sottoattributi delle azioni <jsp:forward>, <jsp:useBean>, <jsp:include> all'atto dell'esecuzione, della risorsa corrente a una risorsa statica, a una servlet o ad una pagina JSP. L'unico attributo è:

► **name**: rappresenta il nome del parametro referenziato;

► **value**: rappresenta il valore del parametro referenziato.

```
<jsp:param name="paramName" value="paramValue" />
```

### JSP

Java Server Pages or JSP for short is Sun's solution for developing dynamic web sites. JSP provide excellent server side scripting support for creating database driven web applications. JSP enable the developers to directly insert java code into jsp file, this makes the development process very simple and its maintenance also becomes very easy. JSP pages are efficient, it loads into the web servers memory on receiving the request very first time and the subsequent calls are served within a very short period of time.

### How JSP and JSP Container function

A JSP page is executed in a JSP container or a JSP engine, which is installed in a web server or in a application server. When a client asks for a JSP page the engine wraps up the request and delivers it to the JSP page along with a response object. The JSP page processes the request and modifies the response object to incorporate the communication with the client. The container or the engine, on getting the response, wraps up the responses from the JSP page and delivers it to the client. The underlying layer for a JSP is actually a servlet implementation. The abstractions of the request and response are the same as the ServletRequest and ServletResponse respectively. If the protocol used is HTTP, then the corresponding objects are HttpServletRequest and HttpServletResponse.

### Benefits of JSP

One of the main reasons why the JavaServer Pages technology has evolved into what it is today and it is still evolving is the overwhelming technical need to simplify application design by separating dynamic content from static template display data. Another benefit of utilizing JSP is that it allows to more cleanly separate the roles of web application/HTML designer from a software developer.

The JSP technology is blessed with a number of exciting benefits, which are chronicled as follows:

1. The JSP technology is platform independent, in its dynamic web pages, its web servers, and its underlying server components. That is, JSP pages perform perfectly without any hassle on any platform, run on any web server, and web-enabled application server. The JSP pages can be accessed from any web server.
2. The JSP technology emphasizes the use of reusable components. These components can be combined or manipulated towards developing more purposeful components and page design. This definitely reduces development time apart from the At development time, JSPs are very different from Servlets, however, they are precompiled into Servlets at run time and executed by a JSP engine which is installed on a Web-enabled application server such as BEA WebLogic and IBM WebSphere.



Method	Description
isAutoFlush()	Returns true if the output buffer is automatically flushed when it becomes full, false if an exception is thrown.
getBufferSize()	Returns the size (in bytes) of the output buffer.
getRemaining()	Returns the size (in bytes) of the unused portion of the output buffer.
clearBuffer()	Clears the contents of the output buffer, discarding them.
clear()	Clears the contents of the output buffer, signaling an error if the buffer has previously been flushed.
newLine()	Writes a (platform-specific) line separator to the output buffer.
flush()	Flushes the output buffer, then flushes the output stream.
close()	Closes the output stream, flushing any contents.

## Verifichiamo le conoscenze

### 1. Risposta multipla

**1** L'acronimo WAR significa:

- a. Web Application Repository
- b. Web Archives Repository
- c. Web Application Root
- d. Web Archives Root

**2** Quale tra i seguenti oggetti NON è implicito direttamente in service()?

- a. page
- b. config
- c. request
- d. response
- e. in
- f. out
- g. session
- h. application
- i. pageContext
- j. exception

**3** Quale tra le seguenti non è una direttiva?

- a. include
- b. session
- c. taglib
- d. page

**4** Quale tra i seguenti non è un attributo della direttiva page?

- a. language
- b. import
- c. buffer
- d. session
- e. include
- f. info

### 2. Associazione

Associa a ogni TAG il corrispondente significato.

- |                    |                  |
|--------------------|------------------|
| 1 ..... <%= ... %> | a) dichiarazione |
| 2 ..... <%! ... %> | b) direttiva     |
| 3 ..... <% ... %>  | c) scriptlet     |
| 4 ..... <%@ ... %> | d) espressione   |

### 3. Vero o falso

- |                                                                                                            |                                                                                                                                                                             |
|------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>1</b> Una applicazione Web viene impacchettata come file WAR.                                           |   |
| <b>2</b> Una JSP è un'astrazione di alto livello per rappresentare delle servlet.                          |   |
| <b>3</b> Una pagina JSP è formata da codice HTML con encapsulato al suo interno delle direttive JSP.       |   |
| <b>4</b> Una pagina JSP prima viene compilata e quindi viene copiata nella cartella classes.               |   |
| <b>5</b> Come per le servlet il server deve avere a disposizione l'ambiente JRE.                           |   |
| <b>6</b> Le scripting-oriented tag sono definite da delimitatori entro cui è presente lo scripting.        |   |
| <b>7</b> Le XML-oriented tag permettono di definire i medesimi tipi di comandi dei tag scripting-oriented. |   |
| <b>8</b> A ogni chiamata di una pagina JSP il container ne crea un'istanza (genera un thread).             |   |
| <b>9</b> Le scriptlet in una pagina JSP devono necessariamente essere scritte in Java.                     |   |
| <b>10</b> Le direttive non producono nessun output visibile.                                               |   |

# Java Server Pages e Java Bean

In questa lezione impareremo...

- ▷ le caratteristiche dei Bean
- ▷ a richiamare un Bean in una pagina JSP
- ▷ a realizzare una applicazione con JSP e Bean

## ■ Java Bean

All'interno delle pagine JSP è possibile utilizzare particolari classi Java con un'interfaccia molto semplice: sono i Java Bean (letteralmente “chicchi di Java”, cioè “chicchi di caffè”).

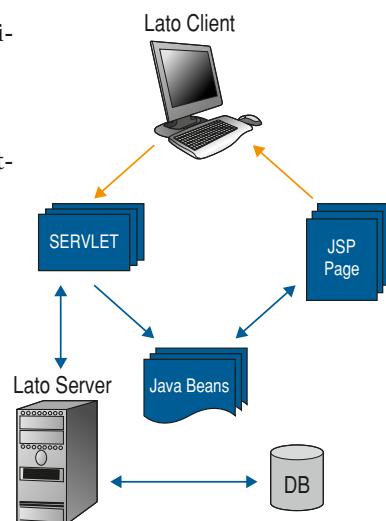
Un Java Bean è un componente nella tecnologia Java, cioè una classe, che può essere utilizzato in modo standard in più applicazioni.

Lo scopo dei componenti è quello di essere *riutilizzati* in contesti diversi e permettere di separare la logica dell'applicazione dalla parte di controllo e di presentazione delle informazioni.

Un Java Bean è quindi una classe particolarmente semplice che risponde sostanzialmente ai seguenti requisiti:

- ▷ è una classe **public**;
- ▷ ha un **costruttore public** di default con zero argomenti;
- ▷ soddisfa una serie di direttive relative ai suoi metodi e ai suoi attributi (detti proprietà).

Grazie alle loro caratteristiche i Bean vengono usati in molte librerie Java, come per esempio nelle swing: quando i Java Bean vengono usati per la gestione del risultato di interrogazioni su una base di dati prendono il nome di Java Data Bean.



## Incapsulamento degli attributi

I **Java Bean** rispettano le direttive che realizzano l'**incapsulamento**, cioè che tutti gli attributi devono essere di tipo privato e a essi si accede solo con i metodi *setter* e *getter*.

In altre parole per ogni attributo sono definiti due metodi:

► **getAttributo()**: metodo che permette la lettura dell'attributo;

► **setAttributo()**: metodo che permette la modifica dell'attributo.

I due metodi hanno il nome della variabile che successivamente verrà utilizzata nella pagina **JSP** denominati con i prefissi **set** e **get**: la prima lettera della variabile deve essere in maiuscolo. Gli attributi nella **JSP** saranno indicati col termine **proprietà** (**property**).

Le proprietà di tipo **boolean** seguono una regola leggermente diversa: il metodo di lettura ha la forma **isAttributo()** anziché **getAttributo()**.

Vediamo un esempio, che ci servirà successivamente per realizzare il primo “collegamento” tra una pagina **JSP** e una classe **Java**, dove la variabile che vogliamo utilizzare è la stringa **testo**:

Definiamo la classe e il suo costruttore:

```
1 package mieiBean;
2 import java.util.*;
3 public class CiaoBean implements java.io.Serializable
4 {
5     private String testo;
6     public CiaoBean() {
7         testo = " ";
8     }
9 }
```

Questo semplice esempio ha un solo attributo, **testo**, che viene inizializzato nell'unico metodo costruttore che deve essere privo di parametri.

```
10 public String getTesto() {
11     GregorianCalendar dataAttuale=new GregorianCalendar();
12     int ore = dataAttuale.get(GregorianCalendar.HOUR_OF_DAY);
13     if ( ore < 12 ) {
14         testo = "Buongiorno: e' mattina!";
15     } else if ( ore < 17 ) {
16         testo = "Buon pomeriggio!";
17     } else {
18         testo = "Buona sera !";
19     }
20     return testo;
21 }
```

Non esiste alcun vincolo di obbligatorietà tra il nome dei metodi e il nome dell'attributo che gestiscono: per esempio avremmo potuto chiamare il metodo che modifica l'attributo **testo** col nome **getParola()** oppure **getSaluto()**.

Vedremo che le variabili di sessione che usano la pagina **JSP** come collegamento alla **Bean** sfruttano il nome dei metodi e non quello dell'attributo: per comodità (e per non creare confusione) si consiglia di denoninarli allo stesso modo.

## Specifica del Bean

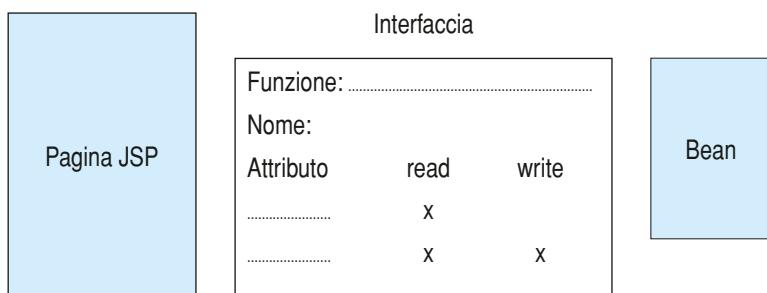
Ai Bean è comodo associare nella documentazione la cosiddetta scheda di “[specifiche del Bean](#)” in modo da semplificare l’utilizzo nelle JSP (e anche nelle servlet!).

La scheda di “**specifica del Bean**” descrive la funzione svolta dal **Bean**, il nome della classe e degli attributi indicandone i possibili utilizzi, cioè se di sola lettura, di scrittura o di lettura/scrittura.

Per l'esempio precedente possiamo avere:

	Specifiche del Bean			
Funzione	In base all'ora del giorno predispone un saluto diverso			
Nome	CiaoBean			
Proprietà	Tipo	Read (get)	Write (set)	Significato
testo	String	X		Messaggio di saluto

Come vedremo la presenza di questa scheda permette di utilizzare il **Bean** senza conoscerne il contenuto: di fatto si realizza quella che è la filosofia della programmazione a oggetti, dove le classi sono delle **black box** che vengono utilizzate esclusivamente mediante la loro **interfaccia pubblica**.



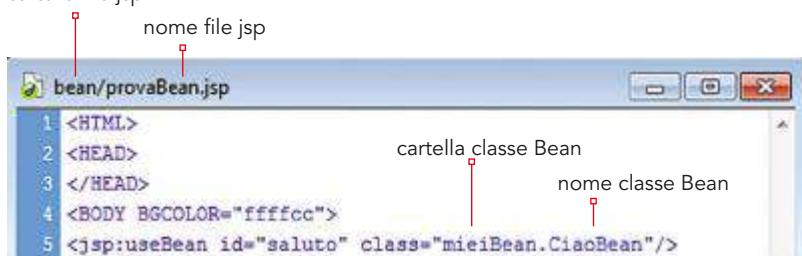
## ■ Uso di Java Bean

**Java** permette di automatizzare la chiamata a un metodo di un **Bean** da parte di una pagina JSP mediante un insieme di elementi di azione standard, denotati del tag <jsp: ... >.

È quindi possibile utilizzare una classe separando la realizzazione del codice Java dalla realizzazione delle pagine JSP.

Per utilizzare un **Bean** in una pagina **JSP** si utilizza la sintassi della riga 5:

## cartella file isp



Con questa istruzione si inizializza una istanza della classe **Bean CiaoBean** in un oggetto di nome (o id) **saluto**: da ora in poi nella nostra pagina JSP possiamo fare riferimento ai metodi della classe **Bean** richiamandoli sull'oggetto di riferimento appena creato, cioè **saluto**.

La sintassi completa dell'istruzione è la seguente:

```
<jsp:useBean id="saluto" class="CiaoBean" scope="page|request|session|application"/>
```

l'attributo **scope** è opzionale e permette di definire l'ambito di **accessibilità** e il **tempo di vita** dell'oggetto (il valore di default è [page](#)). I possibili valori sono riportati nella seguente tabella:

SCOPE	ACCESSIBILITÀ	TEMPO DI VITA
page	Solo la pagina corrente.	Fino a quando la pagina viene completata o fino al forward.
request	La pagina corrente, quelle incluse e quelle a cui si fa il forward.	Fino alla fine dell'elaborazione della richiesta e alla retribuzione della risposta.
session	Richiesta corrente e tutte le altre richieste dello stesso client.	Tempo di vita della sessione.
application	Richiesta corrente e ogni altra richiesta che fa parte della stessa applicazione.	Tempo di vita dell'applicazione.

Per accedere e modificare gli attributi della classe **Bean**, qui chiamati **property**, si utilizzano rispettivamente i seguenti tag: `<jsp:getProperty>` e `<jsp:setProperty>`.

Leggiamo per esempio il contenuto dell'attributo `testo` con la seguente istruzione:

```
11 <jsp:getProperty name="saluto" property="testo"/>
```

che equivale a dire: “dell’oggetto di nome `saluto` prendi il risultato del metodo `get` sulla proprietà `testo`”, cioè esegui il metodo `getTesto()`.

Un tag simile ci permette di scrivere all'interno del **Bean**, modificando il valore dell'attributo desiderato:

```
19 <jsp:setProperty name="saluto" property="testo" value= "ciao"/>
```

Mandiamo in esecuzione il nostro programma e otteniamo:



I codici completi sono disponibili nei file **provaBean.jsp** e **CiaoBean.java** nella cartella materiali.



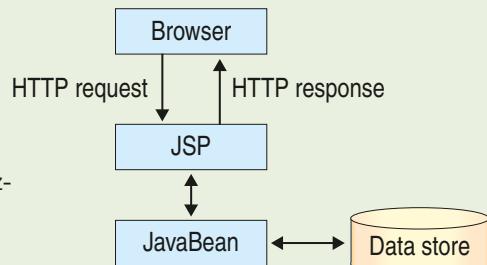
## **Zoom su...**

## MODEL 1

L'architettura J2EE a due livelli costituita da:

- ▶ JSP per il livello di presentazione;
  - ▶ JavaBean per il livello di business logic;

viene denominata **Model 1** e può essere schematizzata nel disegno a lato: ▶



## ■ Configurazione dell'applicazione

Un'applicazione web che utilizza **Bean** deve essere opportunamente configurata e i diversi file vanno posizionati accuratamente nelle diverse cartelle.

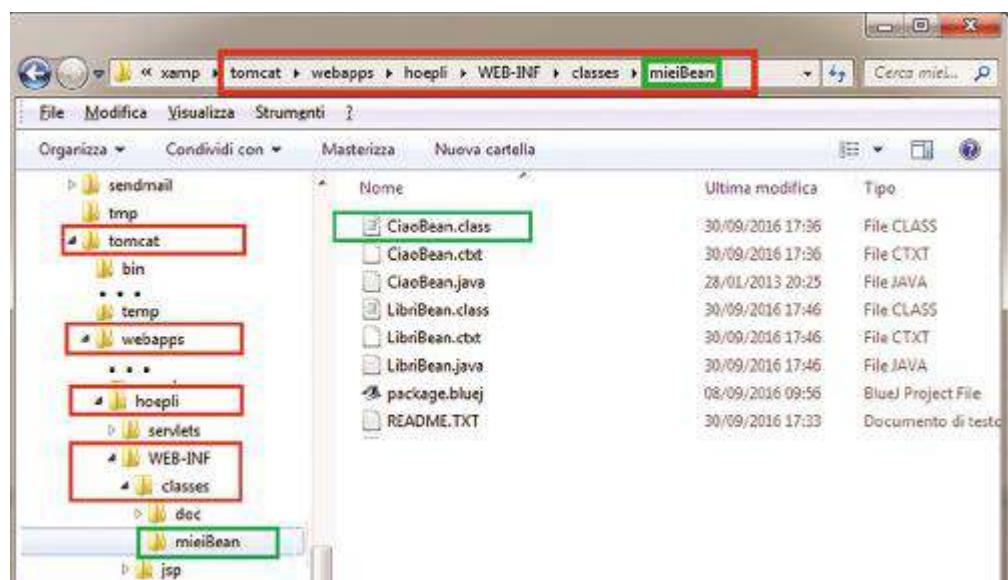
Come primo esempio disponiamo i file che abbiamo realizzato sino a ora:

- file del **Bean**, che contiene classe **CiaoBean.class**;
- file **ProvaBean.jsp** che contiene la nostra pagina **JSP**.

È consigliabile mettere tutti i **Bean** in un package altrimenti il **server** potrebbe non essere in grado di trovare la classe e potrebbe essere necessario inserire la riga di comando che li importa direttamente:

```
<%@ page import = "nomeclasse" %>
```

Metteremo i nostri esempi nel package **mieiBean** che collocheremo nella sottocartella **mieiBean** della cartella **classes** di Tomcat.



Per permettere la visibilità di un **Bean** si deve inoltre inserire nel file **web.xml** il classpath relativo all'applicazione Web che lo realizza.

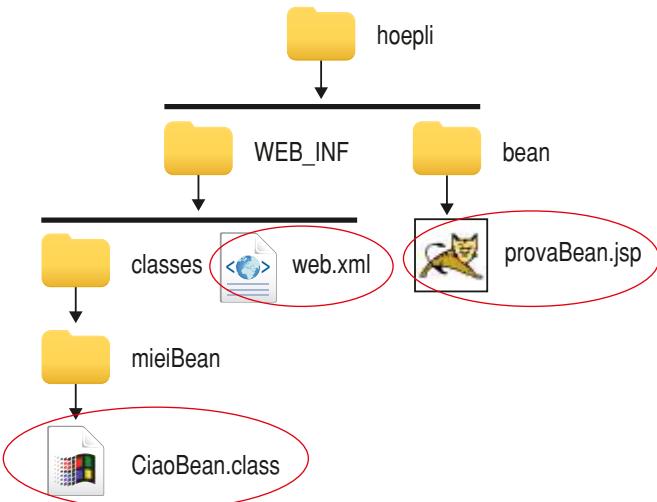
```

276   <servlet>
277     <servlet-name>CiaoBean</servlet-name>
278     <servlet-class>CiaoBean</servlet-class>
279   </servlet>
280   <servlet-mapping>
281     <servlet-name>CiaoBean</servlet-name>
282     <url-pattern>/servlets/servlet/mieiBean/CiaoBean</url-pattern>
283   </servlet-mapping>

```

↓                    ↓  
cartella            classe

Il file **provaBean.jsp** può essere posizionato in una “cartella a piacere”, ad esempio in **bean**: di seguito riportiamo l'albero completo dove inserire/modificare i file della nostra applicazione.



**Prova adesso!**

- Realizzare e utilizzare i Java Bean

Realizza un Bean che analizzando la data corrente visualizza sullo schermo il nome della stagione e quanti giorni mancano per il veglione di S. Silvestro, che si tiene il 31 dicembre.

## ■ Passaggio parametri al Bean

Aggiungiamo al nostro esempio una pagina **HTML** che richiama la pagina **JSP** passandogli un valore come parametro che andrà successivamente a modificare una proprietà del **Bean**.

Realizziamo una applicazione dove un utente seleziona una categoria di libri e, mediante un **Bean** che ha la seguente interfaccia:

Specifica del Bean				
Funzione	Visualizzi i libri disponibili per una categoria richiesta			
Nome	LibriBean			
Proprietà	Tipo	Read	Write	Significato
categoria	String	X	X	Selettore per la categoria
elenco	String	X		Elenco dei libri disponibili

Sullo schermo, come risposta, viene visualizzato l'elenco dei testi disponibili.

Il **Bean** ha nome **LibriBean** e viene aggiunto al file **web.xml**:

```

286 <servlet>
287     <servlet-name>LibriBean</servlet-name>
288     <servlet-class>LibriBean</servlet-class>
289 </servlet>
290 <servlet-mapping>
291     <servlet-name>LibriBean</servlet-name>
292     <url-pattern>/servlets/servlet/mieiBean/LibriBean</url-pattern>
293 </servlet-mapping>

```

Il codice della classe non necessita di spiegazioni: il **Bean** viene inserito nel **package mieiBean** e nel costruttore inizializziamo le stringhe contenenti i titoli dei libri disponibili, non accessibili dall'esterno né in lettura e né in scrittura.

```

1 package mieiBean;
2 import java.util.*;
3 public class LibriBean implements java.io.Serializable {
4 {
5     private String informatica, linguaggi, poesie;
6     private String categoria;
7
8     public LibriBean(){
9         inizializza();
10    }
11
12    private void inizializza(){
13        informatica = "Algoritmi 1 ISBN XXX-YY1 <br> Tecnologie Informatiche ISBN XXX-YY2<br>";
14        linguaggi = "Il linguaggio C ISBN XXX-ZZ1 <br> Java per tutti ISBN XXX-ZZ2 <br> Progra";
15        poesie = "Poesie d'amore e di disperazione ISBN XXX-ZZ1 <br> Vita nuova ISBN XXX-ZZ2 <br>";
16    }
17 }
```

Naturalmente questo esempio è di solo scopo didattico: al posto delle stringhe sarebbe opportuno interrogare il database e "ritornare" all'utente il risultato della query: questa modifica verrà aggiunta al progetto dopo che avremo affrontato lo studio della connessione delle **servlet** e delle **JSP** con i database **MySQL** e **Access**.

Per la proprietà **categoria** inseriamo entrambi i metodi **set** e **get**:

```

18 // Metodi getter e setter
19 public void setCategoria(String categoria){
20     this.categoria = categoria;
21 }
22 public String getCategoria(){
23     return categoria;
24 }
```

Per la proprietà **elenco** l'unico metodo definito è **getElenco()** che ci ritorna in una variabile l'elenco dei libri della categoria prescelta:

```

26 public String getElenco() {
27     String msg = "Nessuna categoria scelta";
28     if ("informatica".equals(categoria)){
29         msg = informatica;
30     }
31     else if ("linguaggi".equals(categoria)){
32         msg = linguaggi;
33     }
34     else if ("poesie".equals(categoria)){
35         msg = poesie;
36     }
37     return msg;
38 }
```

È doveroso osservare che non è presente un attributo con il nome di **elenco**; **elenco** è una proprietà che viene "costruita" in questo metodo: la pagina **JSP** richiama il metodo e riceve un valore, non una variabile!

La pagina **HTML** che richiede all'utente di selezionare una categoria di libri è la seguente:



e ha la codifica riportata di seguito:

```

1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2 <HTML>
3 <HEAD><TITLE> JSP e Bean con query</TITLE></HEAD>
4 <BODY>
5 <FORM ACTION="libriBean1.jsp" METHOD="GET">
6   <P><STRONG><font color="#0000FF" size="4" face="Arial, Helvetica, sans-serif">JSP
7     con Bean e selezione da HTML</font></STRONG></P>
8   <P>Seleziona la categoria di libri che ti interessano:</P>
9   <P>
10    <SELECT NAME="scelta">
11      <OPTION VALUE="informatica">informatica</OPTION>
12      <OPTION VALUE="linguaggi">linguaggi</OPTION>
13      <OPTION VALUE="poesie">poesie</OPTION>
14    </SELECT>
15    <INPUT TYPE="submit" VALUE="Submit">
16    <INPUT TYPE="reset">
17  </P>
18  <P><strong>Nota</strong>: viene restituita una pagina <font color="#FF0000">HTML</font>
19    generata da <font color="#FF0000">libriBean1.jsp </font></P>
20 </FORM>
21 </BODY>
22 </HTML>
```

dove come **ACTION** verrà richiamata la seguente pagina **libriBean.JSP** che nella riga 4 crea un oggetto della classe **LibriBean** del package **mieiLibri** assegnandogli **libri** come identificatore.

```

1 <HTML>
2 <HEAD></HEAD>
3 <BODY BGCOLOR="#ffffcc">
4 <jsp:useBean id="libri" class="mieiBean.LibriBean" />
5 <H1></H1>
6 <H2><font color="#FF0000" face="Arial, Helvetica, sans-serif">Benvenuto nella
7   nostra libreria</font></H2>
8 <HR>
9 <H2><font color="#0000FF">Ti suggerisco i seguenti libri di:
10 <% out.println( request.getParameter("scelta") );%>
11 </font></H2>
```

Con l'istruzione 10 viene letto il parametro **scelta** passato dalla pagina **HTML** che viene nuovamente letto nel tag 14 e assegnato come valore alla proprietà **categoria** della **Bean**.

```
13 <!-- setto nel Bean la variabile categoria con la scelta dell'utente -->
14 <jsp:setProperty name="libri" property="categoria"
15     value="<% request.getParameter(\"scelta\") %>" />
```

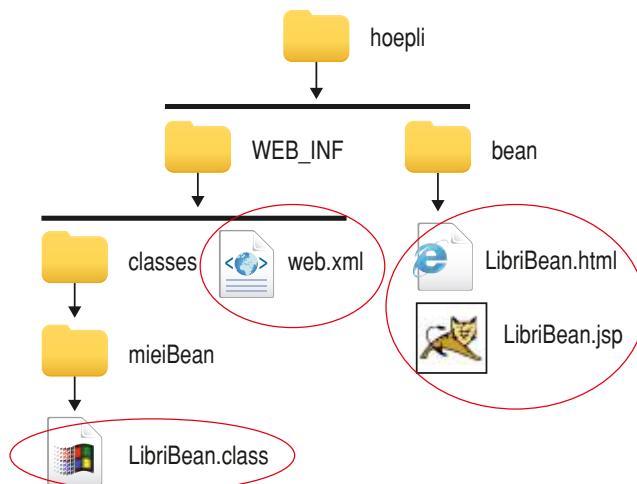
Con l'istruzione 17 viene letta la proprietà **elenco** che direttamente visualizza sullo schermo il risultato della **get**:

```
16 <!-- ricevo dal Bean la stringa di risposta -->
17 <jsp:getProperty name="libri" property="elenco"/>
18 <HR></BODY>
19 </HTML>
```

I tag di formattazione dell'output devono essere messi nella **Bean** e non nella pagina **JSP** che visualizza semplicemente la stringa che riceve dall'oggetto della classe **Bean**. Un'esecuzione ha il seguente output:



I singoli file vengono collocati nelle cartelle di seguito schematizzate:



## Passaggio diretto

Una possibile alternativa alla procedura appena realizzata è quella di effettuare direttamente il passaggio del parametro letto dalla pagina **HTML**, indicandolo con la parola chiave **param**, nel tag di riga 14:

```
13 <!-- setto nel Bean la variabile categoria con la scelta dell'utente -->
14 <jsp:setProperty name="libri" property="categoria" param="scelta" />
```

Imposta il valore della proprietà di un Bean al valore di un parametro della richiesta.

Se nel tag `<jsp:setProperty>` l'attributo `value` non viene specificato, di default viene utilizzato il parametro della richiesta **HTTP** con lo stesso nome: con questa modalità è possibile elaborare molto semplicemente dati in **GET** o in **POST**.

Sempre in `<jsp:setProperty>`, se l'attributo `property` è impostato a `"*"`, verranno impostate tutte le proprietà del **Bean** con nome uguale a un parametro in ingresso.



## Prova adesso!

- Passaggio parametri ai Bean

- 1 Modifica i file dell'esempio precedente in **libriBean1.html** e **libriBean1.jsp** introducendo il passaggio diretto del parametro dalla pagina **HTML** al **Bean** come indicato nella riga 14 di figura.
- 2 Successivamente aggiungi anche la possibilità di selezionare la lingua nella quale sono stati scritti i libri semplicemente passando i parametri con:

```
16 <jsp:setProperty name="libri" property="categoria" param="scelta" />
17 <jsp:setProperty name="libri" property="lingua" />
```

Una soluzione è riportata in **libriBean2.html** e **libriBean2.jsp**: inoltre è stata modificata anche la classe **LibriBean.java**.

## ■ Conclusioni

Oltre al notevole vantaggio di riutilizzare i componenti in contesti diversi, nell'ultimo esempio si è visto come è possibile gestire semplicemente i dati provenienti da una form e passarli velocemente al **Bean** dove implementare anche una logica complessa per effettuare controlli sui campi inseriti dall'utente. Così facendo si mantiene l'isolamento della logica del programma (nel codice **Java**) dalla pagina web stessa (pagina **JSP**).

### **Bean: definition**

A Java Bean is a reusable software component that can be visually manipulated in builder tools.

To understand the precise meaning of this definition of a Bean, clarification is required for the following terms: software component, builder tool, visual manipulation.

### **Reusable Software Components**

Reusable software components are designed to apply the power and benefit of reusable, interchangeable parts from other industries to the field of software construction. Other industries have long profited from reusable components. Reusable electronic components are found on circuit boards. A typical part in your car can be replaced by a component made from one of many different competing manufacturers. Lucrative industries are built around parts construction and supply

in most competitive fields. The idea is that standard interfaces allow for interchangeable, reusable components.



### **Beans, Widgets, Controls, and Components**

Depending on your background, you may use different words to describe GUI components for a given software platform. If you come from a Windows background, you probably think in terms of visual controls, possibly Visual Basic Extensions (VBXs) or OLE Controls (OCXs) and now Active X Controls. If you're more accustomed to environments like Motif or X Windows, you probably think in terms of toolkits or widgets.

In visual application builder environments Beans are sometimes referred to as reusable software components, or custom controls.

## Verifichiamo le conoscenze



### 1. Risposta multipla

**1** Quale affermazione è errata per un Java Bean?

- a. è una classe `public`
- b. ha due metodi definiti di default
- c. ha un costruttore `public` di default
- d. il costruttore ha zero argomenti

**2** Cosa non descrive la scheda di "specifica del Bean"?

- a. la funzione svolta dal Bean
- b. il nome della classe
- c. il nome degli attributi
- d. gli accessi agli attributi
- e. il tipo del costruttore

**3** Quale tra le seguenti opzioni non è prevista per l'attributo scope dell'istruzione `<jsp:useBean..=>`?

- |             |                |
|-------------|----------------|
| a. Page     | d. Session     |
| b. Request  | e. Application |
| c. Response |                |

**4** L'architettura J2EE Model 1 è costituita da:

- |                |                                        |
|----------------|----------------------------------------|
| a. un livello  | c. tre livelli                         |
| b. due livelli | d. numero differente tra JSP e servlet |

**5** Se xxx è la root del mio sito, il package CiaoBean può avere il seguente <url-pattern>:

- a. xxx/servlet/mieiBean/CiaoBean
- b. servlets/xxx/mieiBean/CiaoBean
- c. servlets/servlet/mieiBean/CiaoBean
- d. xxx /CiaoBean/mieiBean



### 2. Vero o falso

**1** Un Java Bean è un componente nella tecnologia Java, cioè una classe.

V F

**2** I Java Data Bean sono particolari Bean che gestiscono la data.

V F

**3** I Java Bean rispettano le direttive che realizzano l'incapsulamento.

V F

**4** Nei Java Bean tutti gli attributi devono essere di tipo privato.

V F

**5** Gli attributi nella JSP saranno indicati col termine proprietà (property).

V F

**6** Le proprietà di tipo boolean sono: `getAttributo ()` e `setAttributo ()`.

V F

**7** Le classi Bean devono implementare la classe `Serializable`.

V F

**8** Il nome dei metodi e il nome dell'attributo che gestiscono devono essere uguali.

V F

**9** La chiamata a un metodo di un Bean da parte di una pagina JSP avviene con il tag `<jsp: ... >`.

V F

**10** Per modificare gli attributi della classe Bean si utilizza `<jsp:getProperty>`.

V F

**11** Per permettere la visibilità di un Bean deve essere modificato il file `web.xml`.

V F

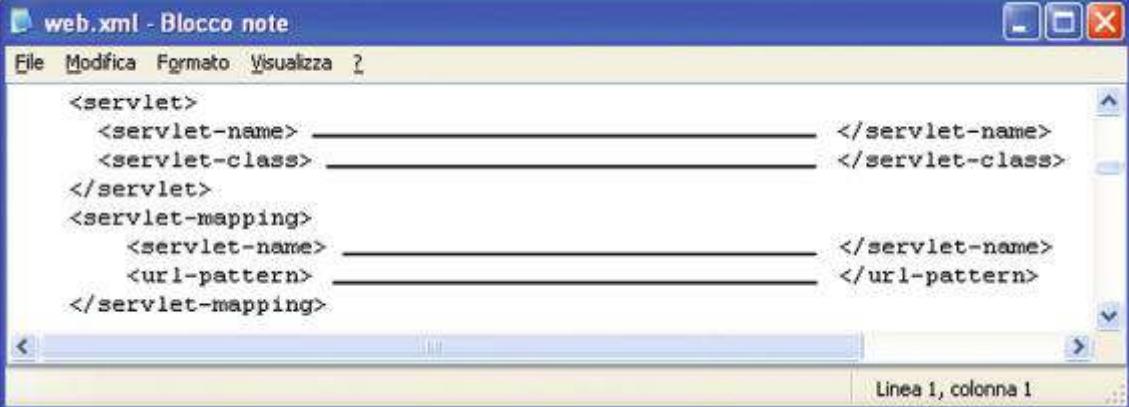
**12** La pagina JSP richiama il metodo e riceve una variabile.

V F

## Verifichiamo le competenze

### Problemi

- 1 Modifica il file web.xml aggiungendo un Bean di nome HelloBean:



```

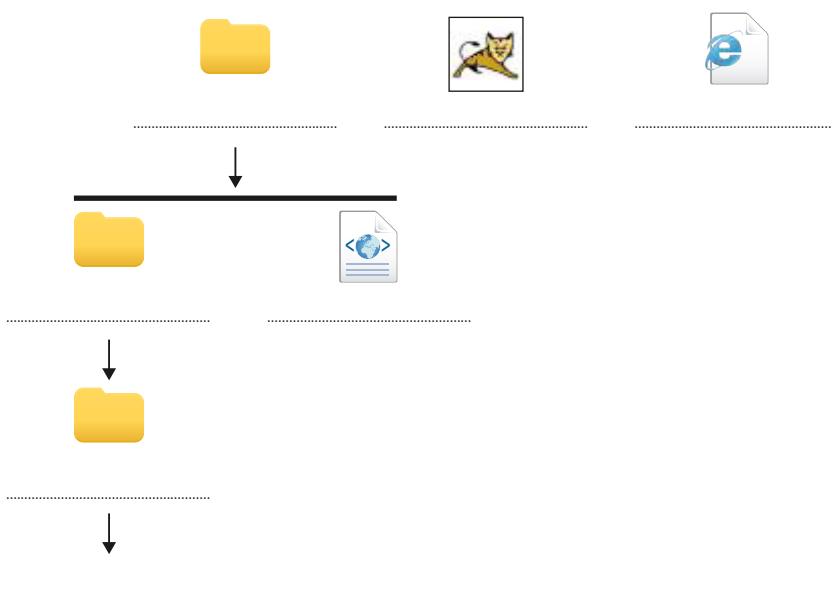
<servlet>
    <servlet-name> _____ </servlet-name>
    <servlet-class> _____ </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name> _____ </servlet-name>
    <url-pattern> _____ </url-pattern>
</servlet-mapping>
  
```

Linea 1, colonna 1

- 2 Disponi correttamente i nomi delle directory e dei file di una generica applicazione "Ciao Utente":

Directory: mieiBean – Web-inf – classes

File: Web.xml – HelloBean.class – HelloBean.html – HelloBean.jsp



- 3 Realizza un'applicazione Hello.xxx che riceva dall'utente il nome e il cognome e controlli se è presente in un elenco (precaricato in un array): in questo caso ne aggiorni la città di nascita che è presente nella pagina HTML.

# ESERCITAZIONI DI LABORATORIO 1

## JSP: PRIMI ESEMPI DI JAVA SERVER PAGES

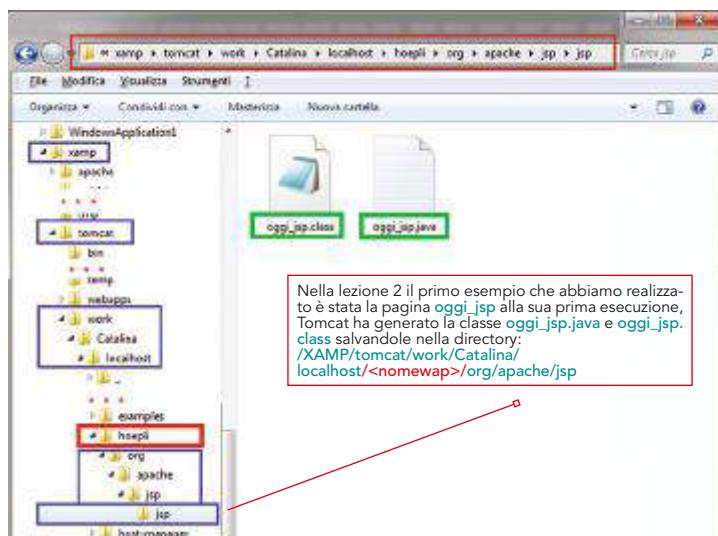
### ■ Codice sorgente Java delle pagine JSP

La prima volta che al **server** viene richiesta una pagina **JSP** questa viene “trasformata” dal **JSP compiler** in una **servlet**, cioè in un file sorgente **.java**: la **servlet** viene a sua volta compilata e produce un file eseguibile **.class** che viene caricato nel **server**, viene eseguito e tenuto pronto per eventuali altre richieste successive.

1. Il **client** richiede la pagina “**oggi**”
2. Il **server** legge la pagina **oggi.jsp**
3. Il **server** genera la **servlet oggi\_jsp.java**
4. Il **server** compila la **oggi\_jsp.java** ottenendo una classe eseguibile **oggi\_jsp.class**
5. Il **server** esegue la **servlet**
6. La risposta viene inviata al **client**



Individuiamo nel nostro **server** la posizione dei due file generati automaticamente:





## Prova adesso!

- Realizzare una pagina JSP

Scrivi la classica pagina “Ciao Mondo” in **JSP**, quindi apri il file **ciaoMondo\_JSP.java** tale che:

<http://localhost:8080/hoepli/jsp/ciaoMondo.jsp> -> Ciao Mondo

<http://localhost:8080/hoepli/jsp/ciaoMondo.jsp?nome=Paolo> -> Ciao Paolo

Analizza il codice generato da **Tomcat**: quali sono le principali osservazioni che puoi fare?

## ■ Java Server Pages: inclusione di file

La direttiva **include** diventa indispensabile quando vogliamo creare dei metodi da utilizzare in più pagine, cioè per crearci una libreria di metodi/funzioni da utilizzare in ogni nostra pagina **JSP**.

Includiamo ad esempio segmenti di codice **HTML** condivisi da tutte le pagine del nostro sito:

```

10  <body>
11  <%@ include file="header.html" %>
12  <%@ include file="menu.html" %>
```

La utilizziamo ad esempio per realizzare una semplice pagina come quella riportata a fianco: ►



## ■ Java Server Pages: iterazioni

Nella pagina **JSP** è possibile inserire delle istruzioni di iterazione, per esempio con il **ciclo for**, molto utilizzato per la costruzione di tabelle che visualizzano i dati generati dalle funzioni **Java**.

Per esempio realizziamo una semplice pagina che visualizzi il fattoriale dei primi 10 numeri utilizzando una funzione ricorsiva **fact()** così codificata:

```

fattoriale.jsp
1 <%! public long fact (long x) {
2   if (x == 0)
3     return 1;
4   else
5     return x * fact(x - 1);
6 }
7 %>
```

La visualizzazione dei risultati viene effettuata con un segmento di codice “misto” tra **HTML** e **Java**, che per ogni iterazione genera una riga a video richiamando la funzione che esegue il calcolo: ►

```

8 <%@page><html><title>Fattoriale ricorsivo</title></head>
9 <body>
10 <H1 ALIGN="CENTER">Fattoriale ricorsivo con ciclo FOR</H1>
11 <TABLE ALIGN="CENTER" BORDER="1">                                <!-- Intestazione tabella -->
12 <TR>
13   <TD><B>x</B></TD>
14   <TD><B>fact(x)</B></TD>
15   <TD><B>fact(x-1)</B></TD>
16 </TR>
17 <!-- istruzione Java si alternato con tag HTML per comporre la riga e scrivere -->
18 <%for (long x = 0; x <= 10; ++x) { %>          <-- per tutte le righe -->
19   <TR><TD>x</TD><TD>fact(x)</TD><TD>fact(x-1)</TD></TR> <-- genera un valore -->
20 <TD>x</TD>
21 </TR>
22 </TABLE>
23 </body>
24 </html>
```

Mandandola in esecuzione otteniamo il seguente risultato: ►

n	f(n)
0	1
1	1
2	2
3	3
4	6
5	24
6	120
7	720
8	5040
9	40320
10	362880
11	3628800



## Prova adesso!

- Iterazione in JSP

Scrivi una **JSP** che generi la sequenza di **Fibonacci** per i primi 10 numeri visualizzandoli in una tabella simile a quella del fattoriale, ma alternando il colore di sfondo per le righe di indice pari con quelle di indice dispari.

## ■ Variabili locali e globali

È necessario tener ben presente che:

- una **variabile dichiarata** nelle righe iniziali con `<%!` è una variabile di istanza condivisa da tutti i **thread** e quindi sono possibili problemi dovuti all'**accesso concorrente**;
- una variabile dichiarata in una **scriptlet** (che vedremo subito di seguito) è locale al metodo di elaborazione della richiesta e quindi ogni **thread** ne ha la propria copia in utilizzo **esclusivo**.

Scriviamo un segmento di codice dove siano presenti due variabili, una **locale** e una **globale**, e verifichiamone il funzionamento richiamando più volte la stessa pagina dal **browser**, cioè mandandone in esecuzione più istanze: ►

```

1 <%-- dichiarazione globale --%>
2 <%!>
3 int contatoreGlobale = 0;
4 %
5 <html>
6 <body style="background-color:#ffffcc">
7 <title> Esempio di variabili JSP </title>
8 <div style="font-color:#FF0000">Variabile globale e locale </div></body>
9 </html>
10 Contatore globale -> si incrementa: <%= ++contatoreGlobale %>
11 <br> servizio di dichiarazione locale
12 <%>
13 int contatoreLocale = 0;
14 %
15 <div style="font-color:#0000ff">Contatore locale -> non viene incrementato: <%= ++contatoreLocale %> </div>
16 <br>
17 <a href="localeGlobale.jsp">Ricarica la pagina</a></div>
18 </body>
19 </html>

```

Esempio di variabili JSP	
Contatore globale -> si incrementa: 5	Variabile globale e locale
Contatore locale -> non viene incrementato: 1	
<a href="#">Ricarica la pagina</a>	

Se ora mandiamo in esecuzione la pagina e la ricarichiamo più volte possiamo osservare come il contatore globale viene incrementato mentre quello locale, a ogni "ricarica", ha sempre il valore iniziale, cioè 1. ►



## Prova adesso!

- Variabili locali e globali

Scrivi una pagina **HTML** che richieda all'utente il nome e una password e, sfruttando le variabili locali e globali, memorizzi 2 utenti e conti quante pagine sono visualizzate per ciascuno di essi.

Nel caso di accesso di un ulteriore utente, l'applicazione segnali l'impossibilità del servizio in quanto, per poter accedere, il terzo utente deve attendere che uno dei due precedenti utilizzatori abbia visualizzato almeno 5 pagine. In tal caso, il nuovo utente sostituisce quello che ha già collezionato il maggior numero di accessi.

## Cookie e sessioni

La direttiva **page** permette di includere l'utilizzo delle librerie di **Java** e quindi "apre le porte" a un insieme pressoché infinito di possibilità.

Vediamo ad esempio come è possibile definire e utilizzare i **cookie**: scriviamo due segmenti di codice rispettivamente il primo per scrivere un **cookie** e il secondo per leggerlo. ►

Come prime istruzioni del codice **JSP** dobbiamo indicare il linguaggio utilizzato in modo da poter definire un oggetto della **classe Cookie** e utilizzarne su di esso i metodi disponibili. ►

Il resto del codice non presenta difficoltà: leggiamo i cookie dall'oggetto implicito **request** e lo analizziamo alla ricerca del cookie dal nome desiderato (nel nostro caso **provacookie**). ►

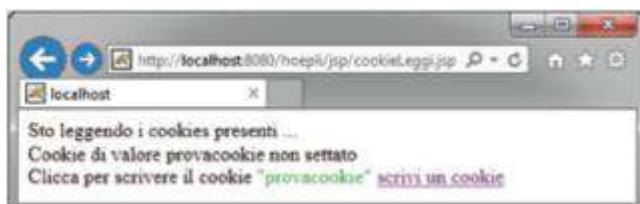
```
<%@ page language="java" %>
<%! Cookie mioCookie = new Cookie("provacookie","pepolo"); %>
<%
mioCookie.setMaxAge(3600); // permanenza del cookie in secondi
mioCookie.setSecure(false); // trasmesso anche con protocoli altri che HTTP
response.addCookie(mioCookie);
%>
Ho scritto un cookie di nome provacookie... <a href="cookieLeggi.jsp">guardalo!</a>
```

```
<%@ page language="java" %>
<%! Cookie mioCookie = null; // oggetto classe Cookie %>
<%! int indice = 0; %>
<%! Cookie[] cookiesUtente; %>
```

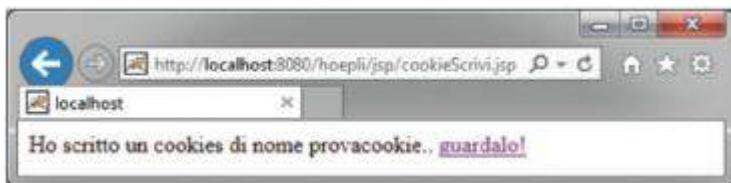
```
11 <> Sto leggendo i cookies presenti ... <br>
12 <% cookiesUtente = request.getCookies(); %>
13 while(indice < cookiesUtente.length){ // scorre tutti i cookie
14   if((cookiesUtente[indice].getName()).equals("provacookie")){
15     break;
16   }else{
17     indice++;
18   }
19 }if(indice < cookiesUtente.length){ // se trovato prima della fine
20   mioCookie = cookiesUtente[indice]; // allora era quello che cercavo
21 }else{
22   mioCookie = null; // altrimenti non è presente
23 if(mioCookie != null){
24   out.println("Cookie di valore provacookie contiene: "+mioCookie.getValue());
25 }else{
26   out.println("Cookie di valore provacookie non settato");
27 }
28 <br>
29 Clicca per scrivere il cookie <a href="style1">provacookie</a> <br>
"cookieScrivi.jsp">scrivi un cookie </i>
```

Richiamiamo tra loro le due pagine in modo da visualizzare prima e dopo il contenuto della memoria:

- chiamo **cookieLeggi.jsp** inizialmente: nessun **cookie** settato.



- 2 chiamo `cookieScrivi.jsp` per settare il mio **cookie provacookie**. ►



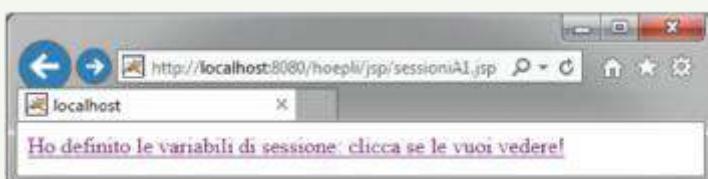
- 3 chiamo nuovamente `cookieLeggi.jsp` e visualizzo il contenuto del **cookie provacookie**. ►



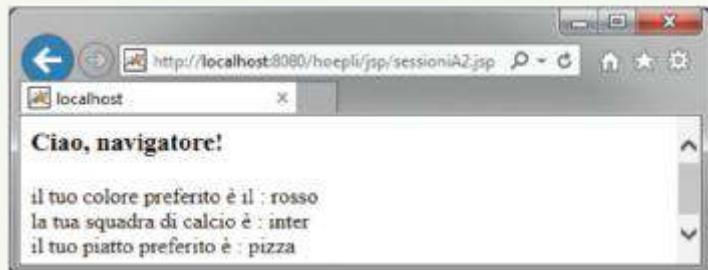
## Prova adesso!

• Variabili di sessione

- 1 Scrivi tre segmenti di codice per definire e visualizzare il contenuto di una sessione. Il primo segmento `sessioniA1.jsp` è semplice codice JSP che definisce tre variabili.
- 2 Il secondo segmento è `sessioniA2.jsp` che visualizza le tre variabili settate.



- 3 Puoi confrontare la tua soluzione con quella presente nei corrispondenti file.



# ESERCITAZIONI DI LABORATORIO 2

## JSP: JAVA SERVER PAGES CON PARAMETRI DA HTML

### ■ Gestione dell'input

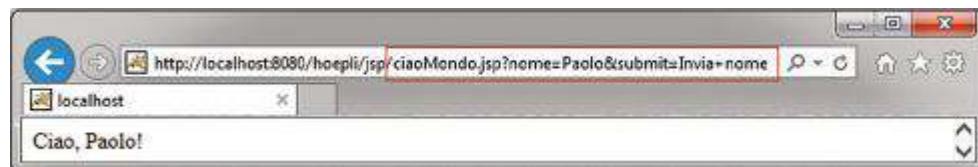
In questa esercitazione vedremo come passare i dati da una pagina **HTML** a una **JSP**: scriviamo una semplice pagina **ciaoUtente.html** che legge e invia il parametro **nome** alla pagina **ciaoMondo.jsp**.



La form richiama la pagina **JSP** mediante il metodo **GET**: nella pagina **JSP** leggiamo il parametro in modo identico a quello visto per le **servlet**.

```
ciaoMondo.jsp
<HTML><BODY>
<% String visitor = request.getParameter("nome");
if (visitor == null) visitor = "Mondo"; %>
Ciao, <%= visitor %>
</BODY></HTML>
```

L'oggetto隐式的 **request** della classe **HttpServletRequest** viene creato automaticamente e su di esso possiamo utilizzare i metodi già visti per le **servlet**: il risultato è il seguente, dove abbiamo evidenziato la stringa di query.





## Zoom su...

### METODI **HTTPSERVLETREQUEST**

La successiva tabella riporta alcuni metodi disponibili per la classe **HttpServletRequest**.

<code>public String getParameter(String name)</code>	Restituisce il valore di un parametro di richiesta di nome <i>name</i> come <i>String</i> , oppure il valore <i>null</i> se il parametro non esiste.
<code>public String[] getParameterValues(String name)</code>	Restituisce un array di oggetti <i>String</i> che contengono tutti i valori posseduti da un determinato parametro della richiesta, (cfr. checkbox) oppure il valore <i>null</i> se il parametro non esiste.
<code>public String getMethod()</code>	Restituisce il nome del metodo <b>HTTP</b> usato per effettuare la richiesta ( <b>GET</b> , <b>POST</b> ).
<code>public String getProtocol()</code>	Restituisce il nome e la versione del protocollo utilizzato per effettuare la richiesta.
<code>public String getRequestURI()</code>	Restituisce l'URL della richiesta corrente, esclusa la stringa di query.
<code>public String getQueryString()</code>	Restituisce la stringa di query contenuta nella richiesta

Nel secondo esempio introduciamo un **list box** contenente il nome di alcune città: ►

Esempio di Servlet con query

JSP con parametro passato da HTML

Inserisci la città dove vivi:

Como  
Milano  
Napoli  
Roma  
Torino

Submit    Reset

Nota: viene restituita una pagina HTML, generata da `parametro1.jsp`

La form richiama la pagina **JSP** mediante il metodo **GET** visualizza questa schermata: ►

Vivi a Como!

Informazioni sulla richiesta HTTP

Metodo richiesto = `request.getMethod()` : GET  
 URI = `request.getRequestURI()` : /hoepli/jsp/parametro1.jsp  
 Protocollo = `request.getProtocol()` : HTTP/1.1  
 QueryString = `request.getQueryString()` : scelta=Como

Oltre al parametro selezionato visualizziamo a titolo didattico anche il risultato di alcuni metodi dell'oggetto **request** della classe **HttpServletRequest**.

Il codice lo puoi trovare nel file **parametro1.jsp**.



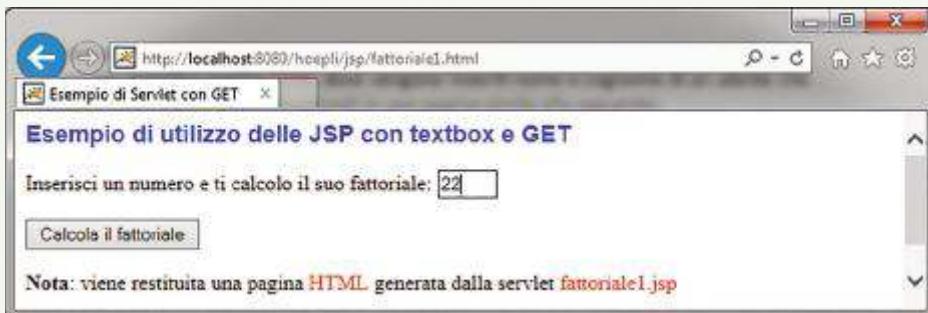
## Prova adesso!

• JSP con parametri

- 1 Scrivi una semplice pagina **HTML** dove vengono inseriti nome e cognome di un utente che vengono successivamente visualizzati in una pagina simile alla seguente:



- 2 Modifica lo **script** che esegue il calcolo del fattoriale passandogli come parametro il numero per il quale deve effettuare il calcolo:



- 3 Scrivi una pagina **HTML** che permetta di selezionare da una list box una categoria e un bottone che una volta cliccato reindirizzi il nome alla pagina **JSP** (**sezioni.jsp**) che si occuperà di scrivere **Sezione XXX**, dove **XXX** rappresenta la categoria presa in input. Le categorie possono essere ad esempio finanza, politica, sport.



### Elaborazione del codice

Naturalmente è possibile elaborare la variabile ricevuta come parametro per inserirla all'interno di codice **scriptlet**: scriviamo una applicazione con la quale un utente seleziona da una **list box** un colore che successivamente viene utilizzato come sfondo della pagina di risposta.

La pagina JSP analizza il parametro “scelta” passato col metodo **GET** dalla form iniziale: se è presente un parametro viene utilizzato come sfondo altrimenti viene definito il bianco come colore di default.



```

parametro2.jsp
1 <HTML>
2 <%
3 String bgColor = request.getParameter("scelta");
4 boolean coloreScelto;
5 if(bgColor != null){           // analisi del parametro
6     coloreScelto = true;
7 }else{
8     coloreScelto = false;
9     bgColor = "WHITE";
10 }
11 <%
12 <BODY BGCOLOR=<%= bgColor %>>    <!-- setto colore di sfondo-->
13 <H2>Colore come parametro</H2><B>
14 <%
15 if(coloreScelto){                  // visualizzo la scelta fatta
16     out.println("Hai scelto un bel colore: " + bgColor + ".");
17 }else{
18     out.println("Colore sfondo non scelto, WHITE di default.");
19 }
20 <%
21 </B></BODY>
22 </HTML>

```

Nella parte finale visualizza sullo schermo una scritta “costruita” in base alla scelta del colore.

Una possibile esecuzione è la seguente:



Confermando il giallo come colore preferito viene prodotta la seguente pagina JSP:





## Prova adesso!

- JSP con parametri
- Interazione HTML e JSP

- 1** Modifica la pagina **HTML** precedente trasformandola in una pagina di **LOGIN**: in caso di mancanza di un dato indica all'utente l'errore e richiedine l'inserimento richiamando la pagina di input ed evidenziando in essa il dato mancante, per esempio colorando lo sfondo del campo in giallo.

Dopo che l'utente ha effettuato la **login**, presentagli una pagina dove sono presenti tre oggetti:

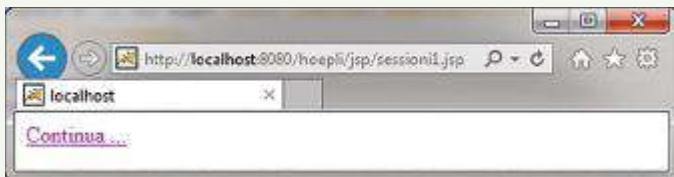
- ▶ un **listbox**, contenente per esempio le materie d'esame di maturità;
- ▶ un gruppo di **radio button**, per esempio per selezionare se scritta od orale;
- ▶ un gruppo di **check box**, per esempio per indicare i giorni della settimana nei quali la materia selezionata verrà studiata.

Dopo che l'utente ha effettuato la scelta ed effettuato la conferma, realizza una pagina **JSP** che ne sintetizzi a video i dati prescelti per tutte e sei le materie d'esame.

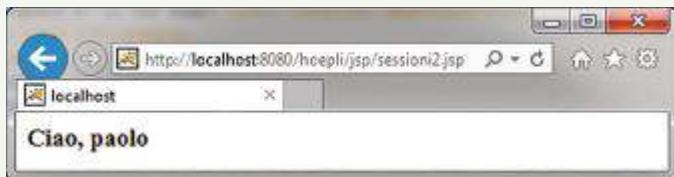
- 2** Scrivi tre segmenti di codice per definire e visualizzare il contenuto di una sessione.

1. Il primo segmento è un semplice codice **HTML** che legge un dato e lo invia come parametro.

2. Il secondo segmento è **sessioni1.jsp** che legge il parametro e setta l'attributo di sessione.



3. Il terzo segmento **sessioni2.jsp** visualizza il contenuto della variabile di sessione.



Puoi confrontare la tua soluzione con quella presente nei corrispondenti file.

# ESERCITAZIONI DI LABORATORIO 3

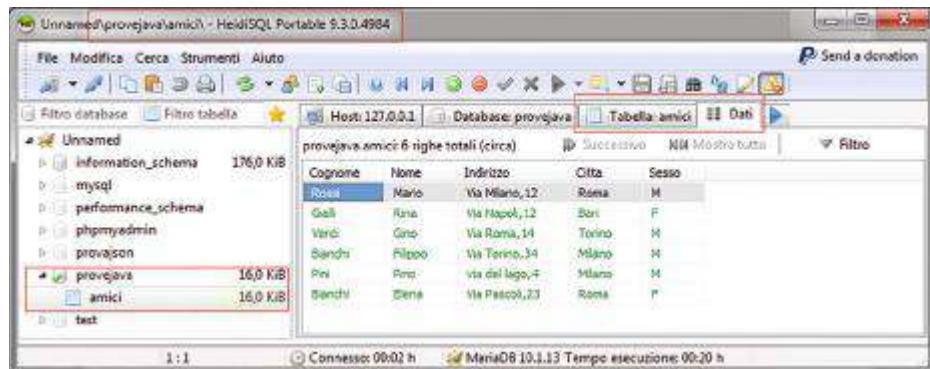
## JSP E DATABASE MYSQL

### ■ Predisposizione dell'archivio MySQL

L'archivio che utilizziamo in questa esercitazione è lo stesso che abbiamo già descritto nelle esercitazioni di laboratorio 6 e 7 dell'unità di apprendimento 4. Come "novità" in questa esercitazione utilizziamo HeidiSQL come "interfaccia" del database, che è prodotto **Open Source** scaricabile direttamente da <http://www.heidisql.com/> oppure dalla cartella **materiali** del **DVD** allegato al presente volume.

Utilizziamo la versione **HeidiSQL\_9.3\_Portable** che non richiede installazione: basta copiare il file **.zip** sulla penna **usb**, decomprimerlo e mandarlo in esecuzione cliccando sul file eseguibile.

Automaticamente Heidi stabilisce la connessione con **MySQL** visualizzando una schermata simile alla seguente, dove abbiamo individuato il nostro archivio e cliccato sulla tabella **Amici**.



### ■ Connessione con JSP

Per poter connettere una **JSP** a un database, come già visto per le **servlet** nella esercitazione 6 carichiamo il **driver** specifico per il database che si desidera utilizzare, nel nostro caso **MySQL**: è lo stesso driver **JDBC Connector/J** già utilizzato per le **servlet**!

Il primo passo è quello di includere la libreria **Java** con i metodi necessari alle operazioni **SQL**:

```
<%@ page import="java.sql.*" %>
```

Definiamo ora le stringhe con il nome del driver, l'indirizzo del **database** e la query di selezione:

```

7  // carico il driver per la connessione al DB MySQL
8  String DRIVER = "com.mysql.jdbc.Driver";
9  // riferimento al database: connessione MySql
10 String URL_mioDB = "jdbc:mysql://localhost:3306/proveJava"; // 3306 di default
11 // definizione della query
12 String query = "SELECT Cognome, Nome, Indirizzo, Citta FROM Amici";

```

La connessione al **DRIVER** la realizziamo in un costrutto **try.. catch ...**

```

14 try{
15     Class.forName(DRIVER);
16 }
17 catch (ClassNotFoundException e) {
18     System.err.println("Driver non trovato" + e);
19 }

```

Segue l'istruzione per la connessione al database con utente è "**root**" e password assente.

```

21 Connection connessione = null;
22 try{ // apro la connessione verso il database
23     connessione = DriverManager.getConnection(URL_mioDB,"root","");
24 }
25 catch (Exception e){
26     System.err.println("Errore nella connessione col database: " + e);
27 }

```

Analogamente a quanto già visto per le **servlet**, l'esecuzione della query ritorna un oggetto di tipo **ResultSet** che viene analizzato all'interno di un ciclo per visualizzare sullo schermo i dati che contiene in un segmento **HTML** pre-formattato:

```

29 try{
30     // ottengo lo Statement per interagire con il database
31     Statement statement = connessione.createStatement();
32     // interrogo il DBMS mediante una query SQL
33     ResultSet resultSet = statement.executeQuery(query);
34     // Scorro e mostro i risultati
35     out.println("<PRE>");
36     out.println("<B>cognome"+ "#9;"+"nome"+ "#9;"+"indirizzo"+ "#9;"+"citta</B><BR>");
37     while (resultSet.next()) {
38         String cognome = resultSet.getString(1);
39         String nome = resultSet.getString(2);
40         String indirizzo = resultSet.getString(3);
41         String citta = resultSet.getString(4);
42         out.println( cognome+ "#9;"+"nome"+ "#9;"+"indirizzo"+ "#9;"+"citta");
43     }
44     out.println("</PRE>");
45 }

```

Termina la pagina **JSP** il ramo **catch** e la chiusura della connessione:

```

49 try{ // chiusura connessione
50     connessione.close();
51 }
52 catch (Exception e) {
53     System.err.println(e);
54 }
55 out.close(); // chiusura oggetto di output
56 %>

```

Il codice completo è disponibile nel file **db1.jsp**.

Un'esecuzione produce il seguente risultato:

cognome	nome	indirizzo	citta
Rossi	Mario	Via Milano,12	Roma
Gialli	Rina	Via Napoli,12	Bari
Verdi	Gino	Via Roma,14	Torino
Bianchi	Filippo	Via Torino,34	Milano
Pini	Pino	via del lago,4	Milano
Bianchi	Elena	Via Pascoli,23	Roma



## Prova adesso!

- Connessione a MySQL

- 1 Modifica la **JSP** e adatta le pagine **HTML** già realizzate per le esercitazioni sulla connessione al database fatte con le servlet per realizzare le medesime applicazioni, che si riassumono in:
  - ▶ una pagina con form che permette l'inserimento dei dati nell'archivio;
  - ▶ una pagina che permette di effettuare la ricerca per città o per sesso.
- 2 Aggiungi inoltre una pagina che permette di cancellare un dato presente e una pagina che ne permette la rettifica in caso di errori di inserimento.

# ESERCITAZIONI DI LABORATORIO 4

## JSP E MDB

### ■ Connessione tra JSP e MDB

Il codice che effettua la connessione tra un pagina **JSP** e il database **MDB** è simile a quello già descritto per le **servlet** nell'esercitazione di laboratorio 7 dell'unità precedente. Utilizziamo anche in **JSP** le librerie **UCanAccess**. Per prima cosa importiamo la libreria **Java**:

```

1 <HTML>
2 <BODY><TITLE>JSP, UCanAccess e database MDB</TITLE></BODY>
3 <H1>JDBC e database MDB</H1>
4 <%@ page import="java.sql.*" %>

```

Definiamo le stringhe con il nome del driver, l'indirizzo del **database** e la query di selezione:

```

6 <%
7 // stringhe di inizializzazione driver
8 String DRIVER = "net.ucanaccess.jdbc.UcanaccessDriver";
9 // riferimento al database: connessione diretta
10 String protocollo = "jdbc:ucanaccess://"; // connessione libreria
11 String mdbpath = "tomcat/webapps/hoepli/esempioDB/"; // percorso relativo
12 String mdbName = "proveJava.mdb;memory=false"; // nome database
13 // riferimento al database
14 String URL_mioDB = protocollo + mdbpath + mdbName;
15 // definizione della query
16 String query = "SELECT Cognome, Nome, Indirizzo, Citta FROM Amici";

```

Carichiamo quindi il **driver** con la seguente istruzione:

```

18 try{
19     Class.forName(DRIVER);
20 }
21 catch (ClassNotFoundException e) {
22     System.err.println("Driver non trovato" + e);
23 }

```

Ed effettuiamo la connessione al **database**:

```

24 // apro la connessione verso il database
25 Connection connessione = null;
26 try{
27     connessione = DriverManager.getConnection(URL_mioDB);
28 }
29 catch (Exception e){
30     System.err.println(e);
31 }

```

Procediamo con l'interrogazione al database definendo dapprima il `resultSet`:

```

32 // interrogazione del database
33 try{
34     // ottengo lo Statement per interagire con il database
35     Statement statement = connessione.createStatement();
36     // interrogo il DBMS mediante una query SQL
37     ResultSet resultSet = statement.executeQuery(query);

```

e successivamente con la visualizzazione dei dati in codice **HTML** preformatto:

```

38 // Scorro il resultSet e mostro i risultati
39 out.println("<PRE>");
40 out.println("<B>cognome"+"&#9;"+"nome"&#9;"+"indirizzo"&#9;"+"citta</B>");
41 while (resultSet.next()) {
42     String cognome = resultSet.getString(1);
43     String nome = resultSet.getString(2);
44     String indirizzo = resultSet.getString(3);
45     String citta = resultSet.getString(4);
46     out.println(cognome+"&#9;"+"nome"+&#9;"+"indirizzo"+&#9;+"citta");
47 }
48 out.println("</PRE>");

```

Lo `script` prosegue con il ramo `catch` per visualizzare eventuali errori nell'interrogazione:

```

50 catch (SQLException e){    // visualizzazione dell'errore
51     out.println("Errore nella esecuzione della query:<br>" +query);
52     out.println(e);
53 }

```

Il programma termina con la chiusura della connessione:

```

54 try{                                // chiusura connessione
55     connessione.close();
56 }
57 catch (SQLException e){    // gestione dell'errore
58     out.println(e);
59 }
60 out.close();
61 %>

```



## Prova adesso!

Modifica la **JSP** e adatta le pagine **HTML** già realizzate per le esercitazioni sulla connessione al database **MySQL** per realizzare le medesime applicazioni, che si riassumono in:

- ▶ una pagina con form che permette l'inserimento dei dati nell'archivio;
- ▶ una pagina che permette di effettuare la ricerca per città o per sesso.

Aggiungi inoltre una pagina che permette di cancellare un dato presente e una pagina che ne permette la rettifica in caso di errori di inserimento.

# ESERCITAZIONI DI LABORATORIO 5

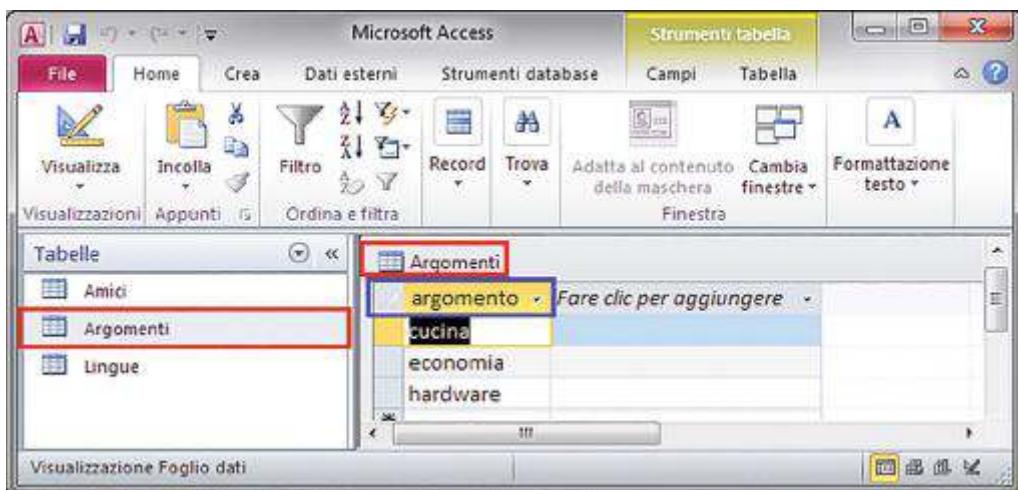
## JSP BEAN E DATABASE

### ■ Connessione tra Java Bean e Access

Vediamo un primo esercizio che ci mostra come realizzare l'interazione tra **Java Bean** e database **Access**: utilizziamo le librerie **UCanAccess** per realizzare la connessione al database **Access**.

A tal fine è necessario:

- A** modificare il database aggiungendo la tabella **Argomenti** con il campo **argomento**:



- B** Creare una classe **Java ScelteDBAccess** che:

- 1** si connette al database **proveJava.mdb**:

```

9 // dati per la connessione al database
10 private final String protocollo = "jdbc:ucanaccess://";           // connessione libreria
11 private final String mdbpath   = "tomcat/webapps/hoepli/esempioDB/"; // percorso relativo
12 private final String mdbName  = "proveJava.mdb;memory=false";      // nome database
13 // stringa completa di indirizzo per il database access
14 String URL_mioDB = protocollo + mdbpath + mdbName;

```

- 2** estraie dalla tabella **argomenti** l'elenco dei dati che contiene:

```

96 // definizione della query
97 String query = "SELECT argomento FROM argomenti";

```

- 3** predisponde il **combo box** da proporre all'utente affinché faccia la sua scelta:

```

108 // scorro e creo la stringa HTML che genera il combobox
109 elenco=<select name=\"argomentoScelto\">;
110 while (resultSet.next()){
111     String argomento = resultSet.getString(1);
112     opzione = "<option value = \"" + argomento + "\">" + argomento + "</option>";
113     elenco = elenco + opzione;
114 }
115 elenco = elenco + "</select>";

```

- C** registrare la classe **ScelteDBAccess** nel file **web.xml**:

```

<catalina>
    <Context>
        <!-- Il nome del database Access -->
        <param name="dbFile" value="C:\Users\luca\Desktop\Java\HoEpli\HoEpli\src\main\resources\HoEpli.mdb"/>
    </Context>
</catalina>
<servlet>
    <servlet-name>ScelteDBAccess</servlet-name>
    <servlet-class>ScelteDBAccess</servlet-class>
</servlet>
<mapping>
    <url-pattern>/servlets/servlet/mieiBean/ScelteDBAccess</url-pattern>
</mapping>

```

- D** creare la pagina JSP (ad esempio **scegliDBBean.jsp**) che carica l'elenco degli argomenti che gli viene passato dal **Bean** che interroga il database **Access**, estrae i dati, li organizza in un **combo box** "ritornagli" la stringa **HTML** che può essere visualizzata nel browser:

```

<HTML>
<BODY BGCOLOR="#ffffcc"><h3>Connessione a database Access </h3>
<h3><font color="#FF0000" face="Arial, Helvetica, sans-serif">Scegli l'argomento di tuo interesse </font></h3>
<FORM ACTION="scegliDBBean.jsp" METHOD="GET">
<jsp:useBean id="elenco" class="mieiBean.ScelteDBAccess" />
<H4><font color="#0000ff">
    Scegli tra questi argomenti:
    <!-- dispongo nel Bean un combobox come risultato del metodo getArgomenti della classe ScelteDBAccess -->
    <jsp:getProperty name="elenco" property="argomenti"/>
    <INPUT TYPE="submit" VALUE="Invia argomento">
</font></H4>
<p><strong>Nota:</strong> viene restituita una pagina <font color="#FF0000">scegliDBBean.jsp </font> con la scelta effettuata<font color="#ff0000"> </font></p>
</FORM></BODY>
</HTML>

```

La sua esecuzione è la seguente, dove possiamo vedere i dati presenti nel **combo box**:





## Prova adesso!

• Connessione a Access

- 1 Dopo aver aggiunto le tabelle **comuni**, **province** e **regioni** scrivi una applicazione che visualizza l'elenco delle **regioni** in modo che l'utente ne possa selezionare una: dalla scelte dell'utente si deve presentare una nuova combo box contenente l'elenco delle **province** presenti nella **regione** indicata.
- 2 Alla successiva scelta dell'utente si carichino in un ulteriore combo box **comuni** presenti nella provincia desiderata in modo che anche tra questi l'utente possa fare la sua scelta.
- 3 Si visualizzino infine i dati del **comune** prescelto: in particolare il **codice comune**, la sua **longitudine** e **latitudine**.

## ■ Connessione tra Java Bean e MySQL

Come esercizio riepilogativo sui **Java Bean** realizziamo una applicazione che seleziona da un database i libri presenti in archivio in base alle scelte operate da un utente sull'argomento e la lingua desiderata.

Come prima operazione aggiungiamo queste due tabelle al database **provejava**:

ID_LIBRO	argomento	titolo	ISBN	lingua
6	hardware	Hard Arduino	978-88-203-9998-7	inglese
7	hardware	Arduino Easy	978-88-203-9998-8	inglese
8	informatica	Il linguaggio in C++	978-88-203-9999-3	italiano
10	hardware	Informatic in Visual Basic	978-88-203-9999-1	italiano
11	informatica	Il linguaggio in Java	978-88-203-9999-2	italiano
12	linguaggi	Programming in C	978-88-203-9998-9	inglese
13	linguaggi	Le Mind Mapping	978-88-203-9998-0	francese
14	linguaggi	Le Mind Mapping vol.1	978-88-203-9998-8	francese
15	linguaggi	Le Mind Mapping vol.2	978-88-203-9998-5	francese
16	linguaggi	Le Mind Mapping vol.3	978-88-203-9998-8	francese
17	informatica	Traveller avec un ipad	978-88-203-9498-8	francese
18	informatica	Traveller avec un iphone	978-88-203-2998-8	francese
19	linguaggi	Travaux pratiques avec WordPress	978-88-203-9498-8	francese

- A Creare una classe **Java LibriBeanDB** che:
  - 1 si connette alla tabella **libri** del database **provejava**;
  - 2 estrae l'elenco **lingue** e setta un attributo **lingua** dopo che viene scelto dall'utente;
  - 3 estrae elenco **argomenti** e setta un attributo **argomento** dopo che viene scelto dall'utente;
  - 4 estrae l'elenco **libri** dalla tabella **libri** utilizzando una query con filtro;
- B registrare la classe **LibriBeanDB** nel file **web.xml**;
- C creare due pagine **JSP**:
  - 1 la prima, **libriDBBean.jsp**, che carica l'elenco delle lingue e degli argomenti in due **combo box**;
  - 2 la seconda, **caricaLibri.jsp**, che riceve i parametri scelti dal **combo box** e interroga il database visualizzando l'elenco dei libri che soddisfano le condizioni impostate dall'utente.

Dopo aver definito le stringhe con il nome del driver e l'indirizzo del **database** impostiamo la lingua di default e carichiamo il driver nel metodo **inizializza()** richiamato dal costruttore della classe: ►

```

24 private void inizializza() {
25     lingua = "italiano";
26     // carico il driver
27     try {
28         Class.forName(DRIVER);
29     } catch (ClassNotFoundException e) {
30         System.err.println("Driver non trovato" + e);
31     }
32 }
```

Il metodo **getElenco()** restituisce l'elenco dei libri ottenuto dalla interrogazione **SQL** costruita prendendo come parametri i valori dei due attributi **lingua** e **argomento**: ►

```

30 public String getElenco(){
31     String elenco = "";
32     String argomento = "";
33     // definizione della query
34     String query = "SELECT * FROM libri WHERE lingua = '" + lingua + "'";
35     if (argomento != null)
36         query = query + " AND argomento = '" + argomento + "'";
37 }
```

I valori di questi due attributi vengono scelti alla prima pagina JSP della nostra applicazione, **libriDBBean.jsp**, tramite due **combo box** realizzati con le seguenti istruzioni:

```

5 <FORM ACTION="caricaLibri.jsp" METHOD="GET">
6 <jsp:useBean id="libri" class="meibean.LibriBeanDB" />
7 <HR>
8 <H4><font color="#0000FF">
9 Scegli tra questi argomenti:
10 <!-- dispongo nel Bean un list box come risultato del metodo getArgomenti della classe LibriDBBean -->
11 <jsp:getProperty name="libri" property="argomenti"/>
12 </font></H4>
13 <H4>Scegli tra queste lingue:
14 <!-- dispongo nel Bean un list box come risultato del metodo getLingue della classe LibriDBBean -->
15 <jsp:getProperty name="libri" property="lingue"/>
16 </font></H4>
17 <HR>
18 <H4>
19 <font color="#FF0000" face="Arial, Helvetica, sans-serif">ricerca nel database</font>
20 <INPUT TYPE="submit" VALUE="Submit">
21 <HR>
22 </FORM>
```

Sappiamo che il contenuto del **combo box** viene “ritornato” dai due metodi **getArgomenti()** e **getLibri()**, come già descritto nel paragrafo precedente per il database Access.

Quando l'utente sceglie e conferma le sue preferenze dalla seguente videata, ►



viene richiamata la pagina **caricaLibri.jsp** che come prima cosa andrà a settare i due attributi richiamando i metodi:

```

48 public void setArgomento(String argomento) {
49     this.argomento = argomento;
50 }
51
52 public void setLingua(String lingua) {
53     this.lingua = lingua;
54 }
```

tramite le seguenti istruzioni che leggono il parametri passati dalla pagina precedente:

```
28 <!-- setto nel Bean la variabile argomento con la scelta dell'utente -->
29 <jsp:setProperty name="libri" property="argomento"
30           value="<% request.getParameter(\"argomentoScelta\") %>" />
31 <!-- setto nel Bean la variabile lingua con la scelta dell'utente -->
32 <jsp:setProperty name="libri" property="lingua"
33           value="<% request.getParameter(\"linguaScelta\") %>" />
```

Il metodo `getElenco()`, dopo aver eseguito la query `SQL` e ottenuto il `resultSet`, predisponde l'elenco con tutti i libri individuati che passa come parametro di ritorno:

```
    // apro la connessione verso il database.
    connessione = DriverManager.getConnection(URL_mioDB,userName,password);
    // ottengo lo Statement per interagire con il database
    Statement statement = connessione.createStatement();
    // interroga il DBMS mediante una query SQL
    ResultSet resultSet = statement.executeQuery(query);
    // scorre e predispongo l'elenco con i risultati
    elenco = "<PRE><B>Titolo#9c#9c#9c#ISBN</B><BR>";
    while (!resultSet.next()){
        String id_libro = resultSet.getString(1);
        String argomento = resultSet.getString(2);
        String titolo = resultSet.getString(3);
        String ISBN = resultSet.getString(4);
        String lingua = resultSet.getString(5);
        opzione = titolo + "#9c#ISBN#9c#<BR>";
        elenco = elenco + opzione;
    }
    elenco = elenco + "</PRE><BR>";
}
```

che viene richiamato nella pagina JSP dalla seguente istruzione:

```
26
27  <!-- ricevo dal Bean la stringa di risposta con l'elenco dei libri -->
28  <jsp:getProperty name="libri" property="elenco"/>
```

che contemporaneamente visualizza l'attributo **elenco** sullo schermo: ►

Il codice completo è disponibile nei file `libriDBBean.jsp` e `caricaLibri.jsp`.



**Prova adesso!**

- Connessione a MySQL

Modifica l'archivio aggiungendo la tabella **Autori** e collegala alla tabella **Libri**; quindi aggiungi la possibilità di selezionare i libri anche per autore, in modo da avere le seguenti alternative:

- ▶ libri di un autore di qualunque argomento;
  - ▶ libri di un autore di un argomento particolare;
  - ▶ libri di un autore in una lingua prescelta;
  - ▶ libri di tutti gli autori in una lingua e/o argomento prescelto.

# 6

# Applicazioni lato server in PHP e AJAX

L1 I file e l'upload in PHP

L2 Gli oggetti in PHP

L3 La connessione ai database object oriented

L4 Le API di Google

## Esercitazioni di laboratorio

1 Installazione di EasyPHP; 2 Comunicazione client-server con AJAX; 3 Connessione FTP con uno script PHP; 4 Inviare una mail con PHP connesso a MySQL; 5 Creare file pdf con PHP; 6 Creare file di Excel e Word con PHP; 7 Le API di Google: interazione tra JavaScript, Ajax e PHP; 8 Inviare un file con i socket in PHP

### Conoscenze

- Conoscere i file e l'upload in PHP
- Conoscere la programmazione a oggetti di PHP
- Apprendere il ruolo del Web server
- Comprendere il ruolo di AJAX nel dialogo client-server

### Competenze

- Realizzare applicazioni client-server in PHP con l'uso dei socket
- Realizzare la connessione a MySQL in PHP con la OO MySQLi
- Realizzare un servizio di mailing con PHP
- Utilizzare AJAX con PHP
- Realizzare applicazioni Web dinamiche che realizzino interazione con le mappe di Google

### Abilità

- Applicare le API di Google in pagine Web dinamiche
- Scrivere pagine Web con socket
- Realizzare server FTP con PHP
- Realizzare pagine in formato PDF con PHP
- Realizzare file in formato Excel e Word da PHP

## AREA *digitale*



Esercizi



Why in the World Would PHP do this?

- ▶ Come ottenere la API key
- ▶ Vantaggi offerti dal formato PDF



Esempi proposti

Consulta il DVD in allegato al volume



Soluzioni

Puoi scaricare il file anche da hoepliScuola.it

# I file e l'upload in PHP

In questa lezione impareremo...

- a verificare il contenuto di una directory
- a gestire i file di testo
- a fare l'upload di file su un server con le funzioni PHP

## ■ L'apertura di un file

La funzione che PHP mette a disposizione per aprire un file è `fopen()`, con la quale viene associato al file aperto un puntatore che rappresenta l'identificatore del file per poter successivamente effettuare le operazioni di lettura e scrittura.

```
2 $idmiofile = fopen(<nome del file>, <modalità di apertura>);
```

Il `<nome del file>` è rappresentato dal percorso del file da aprire, mentre la `<modalità di apertura>` può essere una delle seguenti:

<code>r</code>	sola lettura
<code>r+</code>	lettura e scrittura
<code>w</code>	sola scrittura: il contenuto viene perso (*)
<code>w+</code>	lettura e scrittura, ma perdendo il contenuto
<code>a</code>	solo per aggiunta (modalità append) (*)
<code>a+</code>	per lettura e aggiunta (*)
<code>b</code>	da aggiungere ai precedenti per trattare il file come file binario

(\*) se il file non esiste PHP tenta di crearlo

Per verificare se l'apertura del file è andata a buon fine è necessario verificare il contenuto del puntatore al file, se esso è falso significa che si è verificato un problema.

```
3 $nomefile = "accessi.txt";           // nome del file da aprire
4 $idfile = fopen($nomefile, "r+");    // tentativo di apertura file
5 if (!$idfile) die ($msg = "il file $nomefile non è stato aperto <br>");
```

La tabella seguente riepiloga le principali funzioni per la gestione dei files:

```

34 fopen($filedaaprire, $mode)           // apre un file
35 fread($filedaleggere)                // legge da un file
36 unlink($daeliminare)                 // elimina un file
37 file_exists($dacontrollare)          // controlla se un file esiste
38 is_writable($dacontrollare)          // controlla se un file è riscrivibile
39 is_readable($dacontrollare)           // controlla se un file è leggibile
40 fwrite ($testodascrivere, $fileincuiscrivere) // scrive in un file eliminandone il
41   // contenuto precedente
42 file_get_contents($filedaleggere)    // legge da un file senza doverlo aprire
43 fputs()                            // identico a fwrite
44 fclose($filedachiudere)             // chiude un file

```

Al termine di ogni elaborazione il file deve essere chiuso con la funzione `fclose()` indicando come parametro il puntatore relativo al file da chiudere.

## ■ Lettura e scrittura in un file di testo

La funzione che consente di leggere un file è `fread()`, che possiede due parametri, <identificatore del file> e <numero di byte> da leggere. In questo esempio vogliamo leggere da un file `contaccessi.txt` i primi "n" caratteri. Il codice che segue mostra come utilizzare la funzione `fread()`:

```

9   // apertura file in lettura
10  $idmiofile = fopen("/percorso/contaccessi.txt", "r");
11  $n = 100;
12  // lettura di cento caratteri
13  $datiletti = fread($idmiofile, $n);

```

La funzione che permette di scrivere all'interno di un file di testo è `fwrite()`; anch'essa possiede due parametri, <identificatore del file> e <stringa da scrivere>. Restituisce il numero di byte che ha effettivamente scritto, mentre in caso di errore restituisce il valore `-1`. Per usare questa funzione il file deve essere aperto in una modalità `scrittura`.

```

16 // apertura file in lettura e scrittura con percorso relativo
17 // in questo caso il file deve trovarsi nella cartella del codice
18 $idmiofile = fopen("contaccessi.txt", "w");
19 // scrive la stringa indicata nel file eliminandone il contenuto precedente
20 $bytescritti = fwrite($idfile, "ali baba");

```

### ESEMPIO Contatore di accessi in file

In questo esempio utilizziamo un file per memorizzare il numero di accessi a una determinata pagina. Come possiamo notare la prima operazione effettuata riguarda la verifica dell'esistenza del file mediante la funzione `file_exists()` (riga 3). Quando il file esiste lo script provvede ad aprire il file (riga 5) e, dopo aver verificato se si riesce ad aprirlo (riga 7), legge il valore in esso contenuto (riga 9) e lo chiude con la successiva istruzione.

```

1  <?php
2  $nomefile = "accessi.txt"; // nome del file dove viene memorizzato il totale accessi
3  if (file_exists($nomefile)) // verifica dell'esistenza del file
4  {
5      $idfile = fopen($nomefile, "r+"); // se esiste già viene aperto in lettura
6      // se non si riesce ad aprirlo viene creato un messaggio di errore
7      if (!$idfile) die ($msg = "il file $nomefile non è stato aperto <BR>");
8      // se il file viene aperto si leggono dieci caratteri e messi in $conta accessi
9      $conta_accessi = (int)fread($idfile, 10); // il casting da stringa in intero
10     fclose($idfile); // chiusura file
11 }

```

Se il file non esiste viene creato memorizzando 0 al suo interno (righe 15-19).

```

12     else:
13     {
14         // se il file non esiste viene creato e aperto contemporaneamente
15         $idfile = fopen($nomefile, "w+");
16         if (!$idfile) die ($msg = "il file &namefile non è stato aperto<BR>");
17         // se il file viene aperto correttamente
18         $conta_accessi = 0;                      // si inizializza la variabile conta_accessi
19         fclose($idfile);                         // chiusura del file
20     }

```

A questo punto la variabile che contiene il totale di visite (`$conta_accessi`) viene incrementata ([riga 23](#)), quindi dopo aver riaperto il file in scrittura distruttiva ([riga 23](#)), viene scritto il nuovo valore ([riga 27](#)); infine viene visualizzato il totale sul browser ([riga 29](#)).

```

22     $conta_accessi++;                      // incremento del contatore di accessi
23     $idfile = fopen($nomefile, "w+");        // apertura del file in scrittura
24     if (!$idfile) die ($msg = "il file &namefile non è stato aperto<BR>");
25
26     // se il file aperto correttamente si scrive nel file del contatore di accessi
27     fwrite($idfile, $conta_accessi);
28     fclose($idfile);                        // chiusura file
29     echo($conta_accessi);                  // visualizza il contatore accessi
30 ?>

```

Per verificare il funzionamento dello script si deve simulare l'apertura della pagina da utenti diversi, richiamando più volte la pagina: purtroppo il contatore si incrementa anche quando la pagina viene aperta dallo stesso client, e questo inconveniente deve essere eliminato altrimenti il contatore non effettua il conteggio delle visite ma di quante volte si è richiesta la pagina.



## Prova adesso!



**APRI IL FILE** `contaAccessi.php`

Modifica il codice dell'esempio in modo tale che mostri dei numeri grafici al posto del numero per il totale delle visite effettuate alla pagina. Per fare questo utilizza le immagini presenti nella sottodirectory `\cifre` presente nella cartella degli esempi (`0.gif`, `1.gif`, ... `9.gif`).

- Apertura di un file
- Caricamento di immagini



## ■ L'array associativo `$_FILES`

La variabile predefinita `$_FILES` contiene i nomi e le informazioni che riguardano i file provenienti dal campo di tipo file di un form. In tal modo possiamo effettuare l'upload di file dal **client** al **server**. In una comunicazione client e server che consenta l'upload dei file gli elementi fondamentali sono i seguenti:

- ▶ **form html** con campo file e proprietà `enctype = multipart/form-data`;
- ▶ funzione **PHP move\_uploaded\_file**(nome del file da caricare, percorso in cui caricarlo);
- ▶ array associativo `$_FILES`.

L'array `$_FILE` possiede la seguente struttura:

```

23  $_FILES['nomecampofile']['name'] // nome ed estensione del file caricato
24  $_FILES['nomecampofile']['type'] // tipo di file caricato (in formato MIME type)
25  $_FILES['nomecampofile']['size'] // dimensione del file caricato
26  $_FILES['nomecampofile']['tmp_name'] // percorso completo del file temp sul server
27  $_FILES['nomecampofile']['error'] // codice numerico compreso fra 0 e 8 indicante
28  // il tipo di errore che si è verificato
29  // oppure 0 nessun//errore

```

### ESEMPIO *Upload di file*

Vediamo come effettuare l'upload di un file di tipo immagine da un **client** a un **server** mediante un form **HTML**: lo script si compone di due parti, secondo la tecnica **postback** (riga 3).

La prima parte viene mostrata al primo accesso quando cioè il campo `$_POST['invia']` non è stato ancora settato (righe 4-10), e contiene un pulsante di invio (**submit**) e un campo di tipo **file** per selezionare il file da inviare al **server**.

```

caricaFile.php
1 <?php
2 if (!isset($_POST['invia'])){           //verifica campo - tecnica POSTBACK
3     //viene mostrato il form
4     echo "<FORM ACTION='".$SERVER['PHP_SELF']."' ENCTYPE='multipart/form-data' ";
        METHOD='post'>";
5     echo "<INPUT TYPE='file' NAME='file_caricato'>";
6     echo "<INPUT TYPE='submit' value='Upload file' NAME='invia'>";
7     echo "</FORM>";
8 }

```

La seconda parte racchiude lo script che viene elaborato alla pressione del tasto di **submit**, in questo caso **Upload file**. Innanzitutto i dati del file proveniente dal **client** vengono salvati nella variabile di comodo (righe 11-13).

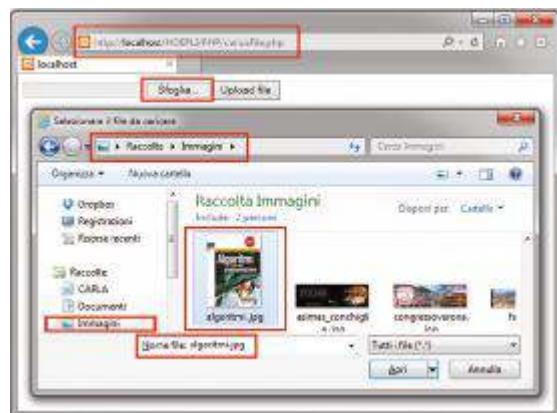
```

9 else{
10     //metto il tipo, il percorso completo e il nome del file in 3 variabili di comodo
11     $tipo=$_FILES['file_caricato']['type'];
12     $nome=$_FILES['file_caricato']['name'];
13     $nome_tmp=$_FILES['file_caricato']['tmp_name'];
14     //verifico il tipo di file, se si tratta di un'immagine (jpg, gif o png)
15     if (($tipo=="image/jpeg")||($tipo=="image/gif")||($tipo=="image/png")){
16         // il file verrà caricato dal client al server nel percorso specificato
17         move_uploaded_file($nome_tmp,"C:/WWW/$nome");
18         echo "il file è stato caricato correttamente!";
19     }
20     else{ // se tipo file non consentito
21         echo "File con estensione non consentita";
22     }
23 }
24 ?>

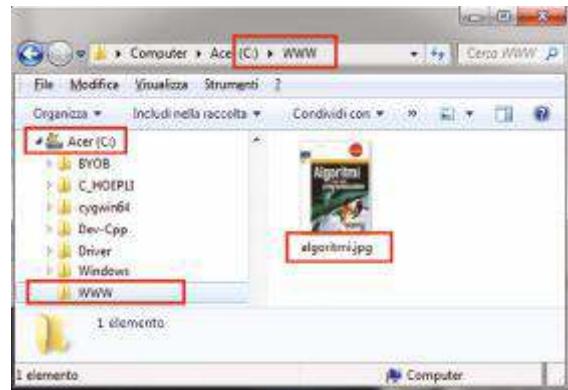
```

Quindi viene verificato il tipo di file da caricare (riga 15): se il tipo di file non è una immagine viene segnalato un errore e lo script termina altrimenti viene usata la funzione `move_uploaded_file()` che carica effettivamente il file dal **client** al **server** e lo memorizza nel percorso specificato come secondo parametro. In questo caso nella directory "`C:/www`" del server e il file mantiene il suo nome originale memorizzato in `$nome`.

Mandiamo in esecuzione il programma e scegliamo una immagine dalla directory **Immagini** del disco locale:



Quindi confermiamo con **[Upload file]** e andiamo a verificare la presenza della immagine nel percorso che abbiamo indicato come destinazione.



## Prova adesso!



**APRI IL FILE** `caricaFileSol.php`

Il programma carica i file all'interno della sottodirectory `./programmi`. Indenta il codice e modificalo in modo tale che si possano caricare i file in tre cartelle: **programmi**, **lavori** e **immagini**. Inserisci quindi un secondo contatore che conti i download effettuati dai file della cartella **lavori**.

- Appicare le sessioni e i file
- Utilizzare le funzioni di upload

**UPLOAD:** to transfer (data or files) from a computer to the memory of another device (such as a larger or remote computer).



**DOWNLOAD:** an act or instance of transferring something (such as data or files) from a usually large computer to the memory of another device (such as a smaller computer).

## Verifichiamo le conoscenze



### 1. Risposta multipla

**1 Per creare un file utilizzo il codice:**

- a. \$idfile=fopen ("miofile.txt","w+");
- b. \$idfile=fopen ("miofile.txt","c");
- c. \$idfile=fopen ("miofile.txt","c+");
- d. \$idfile=fopen ("miofile.txt","w");

**2 La funzione fsize():**

- a. ritorna la dimensione del file in Kbyte
- b. ritorna la dimensione del file in Mbyte
- c. ritorna la dimensione del file in bit
- d. ritorna la dimensione del file in byte

**3 Il puntatore al file è:**

- a. un indirizzo di memoria
- b. un numero intero
- c. un riferimento all'inizio del file
- d. nessuna delle precedenti

**4 La modalità di apertura di un file w+:**

- a. è di lettura e scrittura
- b. è di lettura
- c. è di scrittura
- d. è di lettura e scrittura, ma perdendo il contenuto

**5 Il comando \$datiletti = fread(\$idfile,6);:**

- a. legge il puntatore del file
- b. legge i primi 6 caratteri dell'idfile
- c. legge i primi 6 caratteri del file
- d. legge 6 caratteri del file a partire dalla posizione corrente

**6 Il comando \$bytescritti = fwrite(\$idfile,"ali baba");:**

- a. provoca un errore
- b. scrive "ali baba" all'inizio del file indicato
- c. in \$bytescritti assegnerà valore 8

- d. scrive "ali baba" all'inizio del file indicato solo se aperto in mutua esclusione

**7 Il comando fgetc(\$idfile);:**

- a. restituisce TRUE se il file è terminato
- b. restituisce FALSE se il file è terminato
- c. restituisce 0 se il file è terminato
- d. restituisce -1 se il file è terminato

**8 Il comandofeof(\$idfile);:**

- a. restituisce TRUE se il file è terminato
- b. restituisce FALSE se il file è terminato
- c. posiziona il file a eof
- d. chiude il file

**9 La variabile predefinita \$\_FILEs contiene:**

- a. i nomi dei file più recenti caricati sul server
- b. le informazioni sui file da caricare presenti nei campi TEXT di un form
- c. le informazioni provenienti dal campo file di un form
- d. i nomi dei file di cui effettuare l'upload

**10 L'istruzione: move\_uploaded\_file(\$\_FILES['file'],'./files/file.dat');**

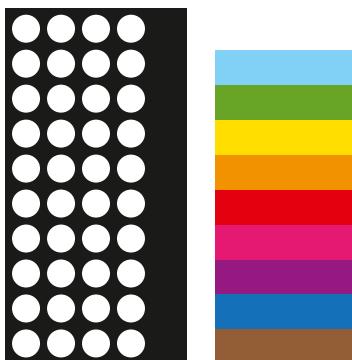
- a. carica sul server nel percorso indicato come primo parametro il file presente come secondo parametro
- b. legge dal form il nome del file da caricare (secondo parametro)
- c. legge dal form il nome del file da caricare (primo parametro)
- d. carica sul server nel percorso indicato come secondo parametro il file presente come primo parametro

## Verifichiamo le competenze

### Problemi

- 1 Mostra un form composto da due list box, regione e provincia di nascita, quindi trasferiscili in una seconda pagina mostrandoli a video.
- 2 Scrivi uno script che verifichi la presenza di un file inserito dall'utente e se presente lo carichi sul server. Se non presente lo crei scrivendoci al suo interno il tuo nome.
- 3 Scrivi uno script che legga il contenuto del file nani.txt e lo mostri a video, all'interno di una tabella con righe a colori di sfondo alternati.
- 4 Scrivi uno script che legga il contenuto del file nani.txt e lo riscriva a video alla rovescia.
- 5 Scrivi uno script che mostri a video il nome, il percorso, l'estensione, la data e l'ora di creazione del file nani.txt.
- 6 Visualizza il contenuto di una cartella a piacere. Per ogni file (di testo) crea un link che consenta di visualizzarne il contenuto.
- 7 Crea N numeri causali e memorizzali su di un file di testo numeri.txt. Quindi stampa a video i numeri pari e dispari in due colonne all'interno di una tabella.
- 8 Crea uno script che mostri l'elenco dei file della cartella /programmi e consenti all'utente di caricarne uno sul server.
- 9 Crea uno script per simulare il gioco di forza4 con il computer, utilizzando le sessioni e i file per memorizzare i risultati parziali dei due giocatori.
- 10 Crea uno script per simulare il gioco del Master Mind utilizzando i file per memorizzare le soluzioni e le impostazioni, come indicato nella seguente figura:

Indovina la combinazione segreta



- Un colore corretto, ma nella posizione sbagliata
- Un colore corretto, nella posizione giusta

# Gli oggetti in PHP

**In questa lezione impareremo...**

- a conoscere il paradigma a oggetti in PHP
- a creare classi, metodi e proprietà
- a istanziare oggetti

## ■ Il paradigma a oggetti in PHP

PHP implementa il **paradigma degli oggetti** (**Object Oriented Paradigm**) anche se è un linguaggio a carattere misto, sia procedurale che a oggetti.

Per la creazione di una nuova **classe** di oggetti si deve utilizzare il costrutto **class**, all'interno del quale possiamo collocare i **metodi** e le **proprietà**, rappresentate rispettivamente da funzioni e variabili: tuttavia le proprietà devono essere dichiarate anteponendo la chiave **var**.

### ESEMPIO **Una classe carrello della spesa**

Nel codice che segue vediamo come realizzare una classe **Carrello** con due proprietà rappresentate da due array che contengono gli articoli presenti nel carrello e la relativa quantità:



```

carrello.php
1 <?php
2 class Carrello
3 {
4     //definizione delle proprietà articoli e quantità come array
5     var $articoli = array();
6     var $quantita = array();

```

Nella classe sono definiti quattro metodi:

- **aggiungi\_carrello()**: aggiunge al carrello un ordine di un nuovo articolo;
- **togli\_carrello()**: elimina un elemento dal carrello;
- **aggiorna()**: effettua l'aggiornamento di un prodotto in termini di quantità;
- **stampa()**: mostra a video tutti gli elementi dell'oggetto carrello, cioè quanto inserito nella nostra borsa della spesa virtuale.

Il metodo `aggiungi_carrello()` riceve come parametro il nome dell'articolo da aggiungere e la quantità: dopo aver letto la dimensione dell'array attraverso la funzione predefinita `count()` (riga 11) viene effettuato un ciclo (riga 14) per verificare se l'elemento che si vuole aggiungere è già presente (riga 15) e in caso positivo viene chiamato il metodo `aggiorna()` (riga 19) che provvederà solamente ad aggiornare la quantità del prodotto aggiungendo la nuova quantità. Se l'elemento non è presente, questo viene aggiunto ai due array (righe 21-22).

```

8 //metodo che aggiunge $a articoli di tipo $q al carrello
9 function aggiungi_carrello($a, $q){
10    //ottengo la lunghezza dell'array
11    $position = count($this->articoli);
12    $trovato = 0;
13    //ciclo di ricerca elemento nell'array
14    for ($i = 0; $i < count($this->articoli); $i++){
15        if ($this->articoli[$i] == $a) //ctr se il prodotto è presente nel carrello
16            $trovato=1;
17    }
18    if ($trovato == 1)           //se il prodotto è già presente aggiorno quantità
19        $this->aggiorna($a,$q);
20    else{                      //altrimenti aggiungo il prodotto al carrello
21        $this->articoli[$position] = $a;
22        $this->quantita[$position] = $q;
23    }
24}

```

Come possiamo notare nella riga 19 davanti al nome del metodo `aggiorna()` è stato anteposto l'operatore di **auto referenza** `$this` e la notazione freccia (`>`): l'operatore di **auto referenza** serve per indicare che il metodo appartiene alla classe. La notazione rappresentata da una freccia formata da un trattino e dal simbolo maggiore (`>`) è chiamata **brackets notation**.

Il metodo `togli_carrello()` riceve come parametro il nome dell'articolo da eliminare (riga 27) che viene cercato nell'array `articoli` (riga 31): se l'elemento è presente viene salvata la posizione nella variabile `posizione` (riga 33) e successivamente (righe 38- 42) si copiano gli elementi degli array `articoli` e `quantita` negli array temporanei (`app_ar` e `app_qt`) tranne l'elemento da cancellare (riga 390): ultime operazioni sono quelle di ripristinare il contenuto degli array-attributi partendo dagli array temporanei appena aggiornati.

```

26 //metodo che elimina l'articolo di nome $a
27 function togli_carrello($a){
28    //variabile utilizzata per cercare posizione elemento: settata a non trovato (-1)
29    $posizione = -1;
30    //ciclo di ricerca elemento nell'array
31    for ($i = 0; $i < count($this->articoli); $i++){
32        if ($this->articoli[$i] == $a) //ctr se il prodotto è presente nel carrello
33            $posizione = $i;          //se trovato salvo la posizione
34    }
35    if ($posizione != -1){         //se trovato procedo a toglierlo
36        $cont = 0;
37        // copia negli array temporanei gli elementi eccetto quello da cancellare
38        for ($i = 0; $i < count($this->articoli); $i++){
39            if ($this->articoli[$i] != $a){
40                $app_ar[$cont] = $this->articoli[$i];
41                $app_qt[$cont] = $this->quantita[$i];
42                $cont++;
43            }
44        }
45        //elimino i "vecchi" array
46        unset($this->articoli);
47        unset($this->quantita);
48        //rimetto i valori dagli array temporanei agli array articoli e quantita
49        for ($i = 0; $i < count($app_ar); $i++){
50            $this->articoli[$i] = $app_ar[$i];
51            $this->quantita[$i] = $app_qt[$i];
52        }
53    }
54    else echo "Prodotto non trovato!<BR>";
55}

```

Il metodo `aggiorna()` riceve due parametri (`$n` e `$q`) che rappresentano il nome del prodotto da cercare e la quantità alla quale aggiornarlo. Il ciclo (riga 60) ricerca la presenza dell'elemento nell'array, quando l'elemento è stato trovato (righe 62-63) viene salvata la posizione.

Se l'elemento è presente si aggiorna la quantità oppure viene segnalato l'errore.

```

57     //metodo che aggiorna l'array
58     function aggiorna($n,$q){
59         $posizione = -1;
60         for ($i = 0; $i < count($this->articoli); $i++){
61             //prelevo la posizione del prodotto nell'array
62             if ($this->articoli[$i] == $n)
63                 $posizione = $i;
64         }
65         //aggiorna le informazioni del prodotto
66         if ($posizione == -1)
67             echo "Prodotto non trovato!<BR>";
68         else
69             $this->quantita[$posizione] = $q;
70     }

```

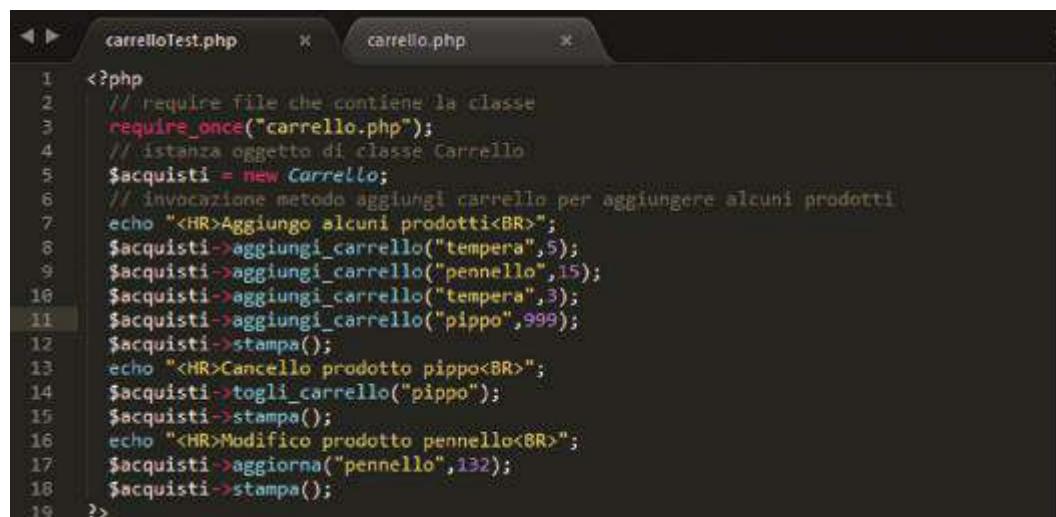
Il codice seguente è relativo al metodo `stampa()` e non necessita di chiarimenti:

```

71     //metodo di stampa a video dei prodotti
72     function stampa(){
73         for ($i = 0; $i < count($this->articoli); $i++){
74             echo "<B>Articoli:</B> " . $this->articoli[$i] . "<BR>";
75             echo "<B>Quantità:</B> " . $this->quantita[$i] . "<BR>";
76         }
77     }
78
79 } // fine definizione classe
80 ?>

```

Scriviamo ora un programma di prova `carrelloTest.php` dove utilizziamo la classe `Carrello`: è necessario includere il file `carrello.php` che contiene il codice relativo alla classe da istanziare (riga 3). L'istanza della classe `Carrello` avviene per mezzo dell'operatore `new` (riga 5) e l'invocazione dei metodi avviene nel codice sull'oggetto `$acquisti` con la brackets notation (`->`).

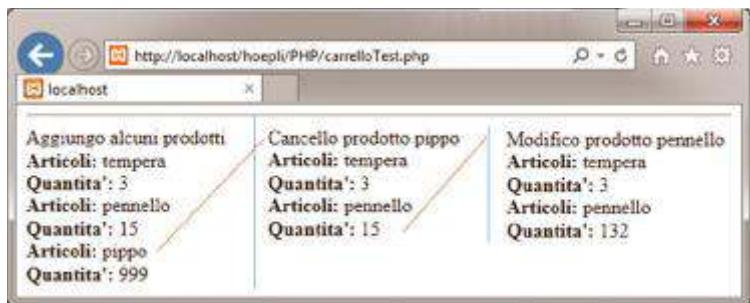


```

1 <?php
2     // require file che contiene la classe
3     require_once("carrello.php");
4     // istanza oggetto di classe Carrello
5     $acquisti = new Carrello();
6     // invocazione metodo aggiungi_carrello per aggiungere alcuni prodotti
7     echo "<HR>Aggiungo alcuni prodotti<BR>";
8     $acquisti->aggiungi_carrello("tempera",5);
9     $acquisti->aggiungi_carrello("pennello",15);
10    $acquisti->aggiungi_carrello("tempera",3);
11    $acquisti->aggiungi_carrello("pippo",999);
12    $acquisti->stampa();
13    echo "<HR>Cancello prodotto pippo<BR>";
14    $acquisti->togli_carrello("pippo");
15    $acquisti->stampa();
16    echo "<HR>Modifico prodotto pennello<BR>";
17    $acquisti->aggiorna("pennello",132);
18    $acquisti->stampa();
19 ?>

```

Eseguiamo tre operazioni dello script ottenendo il risultato mostrato nella videata a fianco:



## Prova adesso!



**APRI IL FILE** `carrelloTest.php`

Modifica il file in modo che mostri all'utente una pagina in cui può scegliere se aggiungere, cancellare o aggiornare i prodotti dall'elenco della spesa virtuale, quindi a ogni aggiunta o eliminazione visualizzare quanto presente nel carrello.

- Utilizzare le classi
- Istanziare oggetti
- Usare metodi e proprietà

## ■ Il costruttore

I **costruttori** sono metodi di una classe aventi il nome della stessa classe: un costruttore viene richiamato automaticamente quando viene creata una nuova istanza.

In PHP è possibile definire un **unico costruttore**, a differenza degli altri linguaggi **OOP** dove i costruttori si distinguono per il tipo e il numero dei parametri.

Il codice seguente riporta un costruttore per la classe **Carrello** che assieme alla creazione aggiunge un prodotto:

```
81     function Carrello($a,$b){
82         $this->aggiungi_carrello($a,$b);
83     }
```

All'atto della creazione dell'oggetto, che viene effettuata con l'operatore **`new()`**, vengono passati i valori dei parametri, che in questo caso sono l'**articolo** e le **quantità**:

```
33 // istanza creata con il costruttore al quale vengono passati due parametri
34 $carrello = new Carrello("Cornice", 3);
35
```

### ESEMPIO **Una classe cronografo**

In questo esempio utilizziamo un classe per verificare la durata di esecuzione di uno script, che utilizza due metodi **StartStop()** e **LapReset()**. L'implementazione di tali metodi della classe si basa sulla funzione **time()** che restituisce il **Timestamp** di **UNIX**.



◀ **Timestamp** è una sequenza di caratteri che rappresentano una data in modo da renderla comparabile con un'altra per stabilirne l'ordine temporale. Nei sistemi operativi Unix-like, come ad esempio Linux il tempo viene rappresentato come offset in secondi rispetto alla mezzanotte (UTC) del 1° gennaio 1970, chiamata **UNIX epoch**. ►

```

1 <?php
2 class Cronometro {
3     var $inizio,$fine;
4     function Cronometro(){
5         $this->azzer();
6     }
7     function azzer(){
8         return $this->inizio = $this->fine = 0;
9     }
10    function vai(){
11        $this->inizio += time() - $this->fine;
12        return $this->fine;
13    }
14    function parziale(){
15        return time() - $this->inizio;
16    }
17    function fermati(){
18        return $this->fine = time() - $this->inizio;
19    }
20 }

```

La risoluzione di questo cronometro non è molto elevata (essendo di ben 1 secondo), e inoltre esso non tiene conto del tempo di esecuzione del solo processo ma del tempo complessivo di esecuzione: infatti il metodo **fermati()** calcola la differenza tra l'ora di sistema di inizio processo e l'ora di fine.

```

23 //istanza di una classe Cronometro
24 $chrono = new Cronometro;
25 echo 'Vai: ', $chrono->vai(), '<BR>';
26 sleep(2);
27 echo 'parziale: ', $chrono->parziale(), '<BR>';
28 sleep(2);
29 echo 'fermati: ', $chrono->fermati(), '<BR>';
30 echo 'azzer: ', $chrono->azzer(), '<BR>';
31 echo 'Vai: ', $chrono->vai(), '<BR>';
32 sleep(1);
33 echo 'parziale: ', $chrono->parziale(), '<BR>';
34 sleep(2);
35 echo 'fermati: ', $chrono->fermati(), '<BR>';
36
37 ?>

```

Questo script crea un nuovo oggetto **chrono** di classe **Cronometro**, che all'inizio è automaticamente resettato: successivamente viene avviato il cronometro, che restituisce il valore 0 all'inizio, proprio come un vero cronometro, e di seguito sono richiamati tutti i metodi.



**Prova adesso!**

- Metodo costruttore
- Costruttore multiplo

Utilizzando la funzione **func\_num\_args()**, **func\_get\_args()** e **func\_get\_arg()** crea un "costruttore multiplo", cioè un costruttore che individui il numero di parametri e si comporti di conseguenza in modo da simulare l'esistenza di più costruttori diversi.  
Una soluzione è riportata nel file **costruttoriMultipli.php**.

## ■ L'ereditarietà

Secondo il concetto di **ereditarietà** della OOP (Object Oriented Programming) le classi possono essere estese ad altre classi, le classi “estese” possiedono tutte le proprietà e i metodi della classe base e in più quello che viene aggiunto nella definizione estesa. PHP non possiede tuttavia, come ad esempio in C++, l'**ereditarietà multipla**. La parola riservata per definire una estensione di classe, come in Java, è **extends**. La sintassi seguente mostra una classe **MioCarrello** che eredita dalla classe **Carrello** e la estende aggiungendone il metodo **setPrezzo()**:

```
carrelloEredità.php
1 <?php
2 include "Carrello.php";
3 class MioCarrello extends Carrello{
4 {
5     //definizione di una nuova proprietà
6     var $prezzi = array();
7
8     function setPrezzo($a,$p){
9         //ciclo di ricerca del prodotto
10        for ($i = 0; $i < count($this->articoli); $i++){
11            if ($this->articoli[$i] == $a) //verifico se è presente nel carrello
12                $this->prezzi[$i] = $p; //salvo la posizione
13        }
14    }
}
```

Per poter visualizzare il contenuto del carrello compreso il nuovo attributo riserviamo il metodo **stampa()**:

```
15 //metodo di stampa a video dei prodotti
16 function stampa(){
17     for ($i = 0; $i < count($this->articoli); $i++){
18         echo "<B>Articoli:</B> " . $this->articoli[$i] . "<BR>";
19         echo "<B>Quantità:</B> " . $this->quantita[$i] . "<BR>";
20         echo "<B>Prezzo:</B> " . $this->prezzi[$i] . "<BR>";
21     }
22 }
```

Una sequenza di attivazione è la seguente, dove viene richiamato il metodo della classe padre per inserire un prodotto nel carrello:

```
25 //istanzio un nuovo oggetto della nuova classe derivata
26 $nuovo_c = new MioCarrello;
27 //aggiungo un articolo
28 $nuovo_c->aggiungi_carrello("Acquarello", 3);
29 //aggiungo il prezzo all'articolo Acquarello
30 $nuovo_c->setPrezzo("Acquarello", 12);
31 $nuovo_c->stampa();
```

Nelle classi derivate il costruttore della classe padre non viene chiamato automaticamente quando viene chiamato il costruttore della classe derivata, quindi se vogliamo richiamare il costruttore della classe ereditata dobbiamo indicarlo esplicitamente.

**OOP:** Stands for “Object-Oriented Programming.” OOP (not Ooops!) refers to a programming methodology based on objects, instead of just functions and procedures. These objects are organized into classes, which allow individual objects to be grouped together. Most modern programming languages, including Java, C/C++, and PHP, are object-oriented languages. Object-oriented programming makes it easier for programmers to structure and organize software programs.

**OBJECT:** An “object” in an OOP language refers to a specific type, or “instance,” of a class. Each object has a structure similar to other objects in the class, but can be assigned individual characteristics.

An object can also call functions, or methods, specific to that object. For example, the source code of a video game may include a class that defines the structure of characters in the game.



# Verifichiamo le competenze

## Problemi

- 1 Definisci una classe Convertitore, in grado di convertire valute diverse, ad esempio Dollaro-Euro. Per il programma si richiede la scrittura di almeno un metodo per il calcolo, un metodo di stampa dei valori risultanti e la definizione di uno o più attributi privati ove memorizzare i dati.
- 2 Definisci una classe Libro contenente i seguenti attributi: nome del libro (array di caratteri), prezzo, numero di scaffale, numero di pagine, casa editrice. Inoltre definisci i seguenti metodi:
  - inizializza
  - mostra
  - applicaScontole quali, rispettivamente, hanno i seguenti compiti:
  - inizializzare i campi dati dell'oggetto classe
  - stampare tutti i dati dell'oggetto
  - diminuire del 10% il prezzo del libro in oggetto
- 3 Crea la classe Massimi che contiene i metodi max() e min(), che calcolano rispettivamente il valore massimo e minimo di due numeri passati come parametri.
- 4 Crea una classe Negozio con due costruttori che inizializzino lo stato dell'oggetto, dotato dei seguenti parametri:
  - proprietario
  - nomeNegozio
- 5 Crea una classe ContoCorrente che contiene tre costruttori che inizializzino lo stato dell'oggetto, dotato dei seguenti parametri:
  - nome
  - codiceConto
  - saldo
- 6 Crea la classe Prodotto utilizzando meno codice possibile con tre costruttori che inizializzino lo stato dell'oggetto, dotati dei seguenti parametri:
  - codice
  - descrizione;
  - codice
  - descrizione
- 7 Crea la classe Fattura utilizzando meno codice possibile con tre costruttori, che inizializzino lo stato dell'oggetto, dotati dei seguenti parametri:

cliente	numeroProdotti
cliente	numeroProdotti
- 8 Definisci una classe LineaBus per rappresentare oggetti linea di autobus urbano con il numero identificativo, il nome dei due capolinea e con opportuni metodi d'istanza per la descrizione del suo percorso.
- 9 Definisci una classe Auto per rappresentare oggetti automobile con il nome della marca, il nome del modello, la targa e l'anno di immatricolazione e con opportuni metodi d'istanza.
- 10 Definisci una classe Studente per rappresentare oggetti studente con il cognome, il nome, il codice fiscale, il numero di matricola e con opportuni metodi d'istanza.

# La connessione ai database object oriented

In questa lezione impareremo...

- ▶ l'utilizzo della classe MySQLi
- ▶ La connessione MySQLi procedurale e OOP MySQLi

## ■ Premessa

Negli ultimi anni l'evoluzione del linguaggio **PHP**, se da una parte ha portato notevoli benefici per gli sviluppatori in termini di efficienza e funzionalità, dall'altra ha obbligato gli stessi a effettuare notevoli interventi di manutenzione sul codice esistente per portare aggiornamenti "obbligatori", necessari per adeguare i propri script alle nuove release per sostituire funzioni e istruzioni **deprecate** nel corso degli anni.

Nella versione **PHP7** viene "rimossa" la funzione **mysql\_connect()** che da sempre ha permesso ai programmatori di interagire con il database **MySQL**: tale funzione era stata deprecata 6 anni fa!

I meccanismi presenti nell'ultima release del linguaggio per potersi connettere a MySQL sono sostanzialmente due:

- ▶ **OOP MySQLi**: alternativa che richiede poche linee di codifica;
- ▶ **PHP Data Objects (PDO)**: interfaccia che definisce un livello di astrazione a prescindere dal database che si sta utilizzando e quindi permette di utilizzare le stesse funzioni per eseguire query e recuperare i dati anche utilizzando altri database.

**AREA** *digitale*



Why in the World Would PHP do this?

Se da un parte tutti i programmatori gradirebbero avere un'unica interfaccia per potersi connettere a qualsiasi database in modo “trasparente” la storia (e l'esperienza) ci suggerisce di dubitare di questi “miracoli” soprattutto sapendo quali (e quante!) differenze sono presenti nelle diverse versioni di **SQL** specifiche per ogni database!

Noi abbiamo scelto di utilizzare **MySQLi** per un insieme di motivi che riportiamo sinteticamente:

- 1 non è necessario convertire tutto il vecchio codice di **MySQL** in una sola volta;
- 2 per le stringhe di query “quasi” non ci sono cambiamenti obbligatori mentre nel caso di **PDO** praticamente ogni stringa deve essere “ritoccata”;
- 3 con **MySQLi** sono mantenute la totalità delle funzionalità delle precedenti implementazioni: in **PDO**, ad esempio, non è presente la funzione **data\_seek()**;
- 4 è disponibile anche una versione di **MySQLi procedurale**, oltre alla **OOP MySQLi**, ed in essa i cambiamenti di codice necessari per la migrazione sono veramente pochi.

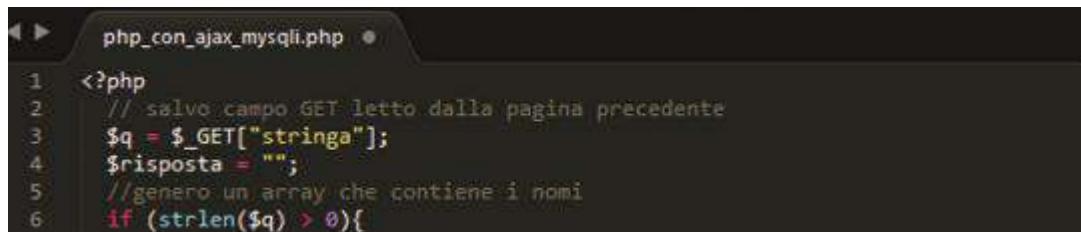
## ■ Connessione a MySQL con MySQLi

Nel primo esempio di **MySQLi** utilizziamo entrambe le notazioni di scrittura, cioè sia quella **OOP** che quella **procedurale**, per confrontarle tra loro e con il “vecchio” codice scritto in **PHP5**.

### ESEMPIO

Scriviamo la parte di connessione al database utilizzabile in una semplice applicazione che visualizza l'elenco dei comuni italiani (con altitudine e longitudine) in base a una richiesta effettuata dall'utente in stile **Google Suggest**.

Le prime istruzioni prendono la stringa di ricerca passata dalla form **html**:



```

1 <?php
2 // salvo campo GET letto dalla pagina precedente
3 $q = $_GET["stringa"];
4 $risposta = "";
5 //genero un array che contiene i nomi
6 if (strlen($q) > 0){

```

La connessione al database avviene dapprima creando un oggetto della relativa classe con la **riga 8**: il costruttore **mysqli()** utilizzerà i parametri da noi forniti per aprire la connessione con il **DBMS**.

I parametri da passare al costruttore sono soltanto 4 più un argomento opzionale:

- 1 l'**host**, cioè l'indirizzo della macchina in cui è in esecuzione **MySQL** (nome o l'indirizzo IP);
- 2 il **nome utente** dell'utilizzatore abilitato a inviare istruzioni al **DBMS**;
- 3 la **password** associata al nome utente utilizzato;
- 4 il **nome del database**.

I metodi **connect\_error()** e **connect\_errno()** sono utili per controllare l'avvenuta connessione:



```

7 // creo un oggetto con i parametri della connessione
8 $mysql = new mysqli("localhost", "root", "", "provephp");
9 // effettuo il tentativo di connessione al database
10 if (mysqli_connect_errno()) {
11     printf("Connessione non effettuata: %s\n", mysqli_connect_error());
12     exit();
13 }

```

Si noti che in questo caso abbiamo adottato la notazione procedurale:

- ▶ `mysqli_connect_error()` // notazione procedurale
- ▶ `$mysql->connect_error()` // notazione a oggetti

All'oggetto creato abbiamo dato volutamente il nome `$mysql`, così che le due istruzioni diventassero il più possibile simili.

Ricordiamo la precedente sintassi, osservando le “analogie”:

```

8 //apertura connessione al database
9 $db = mysql_connect("localhost","root","");
10 //selezione database utenti
11 mysql_select_db("provephp");
12

```

Usiamo invece la notazione a oggetti per effettuare la query, come si può osservare nella riga 15.

```

14     $query = "SELECT name,lat,lng FROM comuni WHERE name like '$q%'";
15     if ($res = $mysql->query($query)) {           // esecuzione query
16         printf("La select ha individuato %d comuni.<br>", $res->num_rows);
17         printf("(comune,latitudine,longitudine)<br><br>");
18     }

```

Anche in questo caso riportiamo la precedente notazione per poter confrontare le istruzioni:

```

17 //esecuzione query. Il risultato è nella recordset $res
18 $res = mysql_query($query) or die ("query fallita!");
19 //ciclo che legge il recordset con la funzione mysql_fetch_array
20 while ($riga = mysql_fetch_array($res)) {

```

Risulta diversa l'istruzione che ci permette di scorrere i dati ottenuti come risultato della query: è comodo utilizzare la funzione `foreach()` che scorre ogni elemento dell'array associativo e lo trasferisce riga per riga in una variabile di servizio, da noi indicata col nome `$riga`:

```

19     if ($res->num_rows > 0){           // se presente almeno un comune
20         foreach( $res as $riga ) { // per estrarre i record restituiti da query
21             // $riga è un array associativo con chiave il nome del campo
22             $risposta .= $riga["name"].",".$riga["lat"].",".$riga["lng"]."<BR>";
23         }
24     }

```

Da ogni riga vengono estratti i singoli campi e accodati alla variabile `$risposta`.

Terminato l'utilizzo del database è opportuno chiudere la connessione per rilasciare la risorsa.

```

25     $mysql->close();
26 }
27
28 //chiusura connessione
29 mysql_close();

```

Il nostro script si conclude con l'istruzione che comunica la risposta alla pagina **HTML**:

```

27     //verifico se $risposta è vuota restituisco stringa "Nessun nome trovato"
28     if ($risposta == ""){
29         $response = "Nessun nome trovato";
30     }
31     else{                                //assegnazione stringa di risposta
32         $response = $risposta;
33     }
34     echo $response;                      //invio risposta alla pagina chiamante
35 ?>

```

Questa pagina PHP viene richiamata da una pagina HTML dove l'utente inserisce il nome (o parte di esso) del comune desiderato: ad esempio può essere utilizzata per sostituire l'array dei nomi dell'esempio proposto sull'utilizzo di Ajax ([lezione 2](#) di laboratorio) e ottenere quanto segue:

Nella [lezione 7](#) di laboratorio verrà realizzato un esempio riepilogativo che utilizza Ajax, la connessione al database e la geolocalizzazione su GoogleMaps dei comuni selezionati.



## MySQL & PHP

There are (more than) three popular ways to use MySQL from PHP.

- 1 (DEPRECATED)** The [MySQL functions](#) are procedural and use manual escaping.
- 2 MySQLi** is a replacement for the mysql functions, with object-oriented and procedural versions. It has support for prepared statements.
- 3 PDO** (PHP Data Objects) is a general database abstraction layer with support for MySQL, among many other databases. It provides prepared statements, and significant flexibility in how data is returned.

I would recommend using PDO with prepared statements. It is a well-designed API and will let you move more easily to another database (including any that supports [ODBC](#)) if necessary.



# Verifichiamo le competenze

## Esercizi

**1** Una galleria d'arte ha deciso di creare un sistema che consenta ai suoi clienti di consultare da casa il catalogo dei quadri, tramite accesso a una pagina web che la galleria può creare presso un fornitore di servizi telematici. Per ogni quadro è compilata una scheda che riporta l'autore, il titolo, la tecnica (olio, tempera ecc.), le dimensioni, il prezzo. Dopo aver progettato il database, realizza una pagina php che permetta di visualizzare i quadri in base a un artista preferito oppure alla tecnica utilizzata. (estratto maturità 1998)

**2** Uno studio medico associato, in cui operano medici di base e medici specialisti, dà incarico a una società d'informatica di progettare e realizzare un database contenente i dati anagrafici e professionali dei medici, l'orario delle visite, i tempi medi previsti per ogni visita, il costo delle singole prestazioni, i dati anagrafici dei pazienti e le prestazioni richieste.

Realizza la pagina web con la quale la segreteria visualizza le visite prenotate per ciascun medico e lo storico delle visite effettuate da ogni paziente. (estratto maturità 2007)

**3** Un'agenzia immobiliare intende potenziare la sua attività per offrire, nella città dove si trova, affitti di case per brevi periodi. In tale città è infatti forte la richiesta di questo servizio in ogni momento dell'anno e anche in relazione a diversi eventi internazionali che richiamano un forte flusso turistico, che non trova accoglienza nelle strutture alberghiere. L'agenzia intende realizzare un sistema, anche accessibile dal suo sito web, che renda pubbliche le offerte di affitto di appartamenti di proprietari privati, consentendo al tempo stesso le prenotazioni e la conferma delle transazioni di affitto.

Realizzare la pagina web con la quale se un cliente seleziona una località e la tipologia di appartamento desiderato il sistema gli propone l'elenco delle soluzioni possibili presenti in archivio con collegamento alle schede, con i dettagli di ogni alternativa. (estratto maturità 2011).

**4** Si vuole realizzare una web community per condividere dati e commenti relativi a eventi dal vivo di diverse categorie, ad esempio concerti, spettacoli teatrali, balletti ecc. che si svolgono in Italia. Gli eventi vengono inseriti sul sistema direttamente dai membri stessi della community, che si registrano sul sito fornendo un nickname, nome, cognome, indirizzo di e-mail e scegliendo una o più categorie di eventi a cui sono interessati. (...) I membri registrati possono interagire con la community sia inserendo i dati di un nuovo evento, per il quale occorre specificare categoria, luogo di svolgimento, data, titolo dell'evento e artisti coinvolti, sia scrivendo un post con un commento e un voto (da 1 a 5) su un evento.

Il sito della community offre a tutti, sia membri registrati sia utenti anonimi, la consultazione dei dati online, tra cui:

- visualizzazione degli eventi di un certo tipo in ordine cronologico, con possibilità di filtro per territorio di una specifica provincia;
- visualizzazione di tutti i commenti e voti relativi a un evento.

Viene richiesta la codifica di una applicazione Web che consente l'interazione con la base di dati per offrire queste prestazioni. (estratto maturità 2015)

# Le API di Google

In questa lezione impareremo...

- ▶ a conoscere la geolocalizzazione e le API di Google Maps
- ▶ a utilizzare le classi di `google.maps` associando eventi
- ▶ a gestire i market, gli infoWindow, i percorsi e street-view panorama

## ■ La geolocalizzazione

La **geolocalizzazione** è l'identificazione della posizione geografica, mediante le sue coordinate di **latitudine** e **longitudine**, di un dato oggetto nel mondo reale secondo varie possibili tecniche. **Google Maps** (il cui nome originario era **Google Local**) è

un servizio accessibile dal relativo sito web che consente la ricerca e la visualizzazione di ▶ mappe topografiche ▶ di quasi tutto il pianeta.



◀ **Topografia** La parola topografia deriva dal greco τόπος ("luogo") e γραφεῖν ("scrivere") e in senso ampio è la scienza che si occupa di determinare con precisione la posizione di entità sulla superficie terrestre e successivamente di rappresentare tali entità sulla carta. ►

Con **Google Maps** è anche possibile ricercare servizi in particolari luoghi, tra cui ristoranti, monumenti, negozi, trovare un possibile percorso stradale tra due punti e visualizzare foto satellitari (statiche) di molte zone con diversi gradi di dettaglio. Oltre a queste funzioni, **Google Maps** offre anche una ricerca di attività commerciali sulle stesse mappe e mette a disposizione delle **API** che consentono di localizzare elementi presenti sulle mappe in base alle ▶ coordinate geografiche ▶.



◀ **Coordinate geografiche**. Servono per identificare la posizione di un punto sulla superficie terrestre, si dividono in **latitudine**, **longitudine** e **altitudine**:

- ▶ la **latitudine** è la distanza angolare del punto dall'equatore;
- ▶ la **longitudine** è la distanza angolare del punto dal meridiano di riferimento (Greenwich) lungo lo stesso parallelo del luogo;
- ▶ l'**altitudine** è la distanza, misurata lungo la verticale del punto considerato sulla superficie terrestre dal livello del mare. ▶





## APPLICATION PROGRAMMING INTERFACE API

In informatica con il termine **Application Programming Interface API** (Interfaccia di Programmazione di un'Applicazione) si indica ogni insieme di procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici, per l'espletamento di un determinato compito all'interno di un certo programma.

La finalità è ottenere un'astrazione, di solito tra l'hardware e il programmatore o tra software a basso e ad alto livello, semplificando così il lavoro di programmazione. Le **API** permettono infatti di evitare ai programmatori di riscrivere ogni volta tutte le funzioni necessarie dal nulla, ovvero dal basso livello, rientrando quindi nel più vasto concetto di "riuso di codice". Le **API** stesse rappresentano quindi un livello di astrazione intermedio: il software che fornisce una certa **API** è detto **implementazione dell'API**.

Le **API** di **Google Maps** utilizzano **JavaScript** come linguaggio predefinito, necessario per la loro implementazione, come molte altre applicazioni web di **Google**.

Le **API** di **Google Maps** nascono nel giugno 2005 per consentire agli sviluppatori di integrare l'applicazione **Google Maps** nei loro siti web: è un servizio gratuito che attualmente non contiene pubblicità anche se **Google**, nei suoi termini di utilizzo, si riserva il diritto di visualizzare annunci in futuro. Con le **API** di **Google Maps** possiamo incorporare mappe all'interno del nostro sito.

Le **API** di **Google Maps** sono gratuite anche per uso commerciale, a condizione che il sito su cui vengono utilizzate sia accessibile al pubblico gratuitamente e non è generi più di 25.000 accessi alle mappe al giorno. I siti che non soddisfano questi requisiti possono acquistare **Google Maps API Premier**.

Le **open API** sono le interfacce di programmazione di un'applicazione e rappresentano uno strumento adatto per rendere disponibile ad altri programmatore le funzionalità di un programma. Il termine **Open** indica che sono aperte, cioè rese disponibili a tutti. Nell'ambito del Web 2.0 le **Open API** sono disponibili sul web e sfruttano le tecnologie e i protocolli, come ad esempio l'**HTTP**: in tal modo un programmatore può includere nel suo programma funzionalità offerte da altri programmi formando quello che viene chiamato un **mash-up**.

**Google Maps** consente di visualizzare la mappa di un luogo prescelto e richiedere informazioni di vario tipo, per esempio dove si trova un'azienda o una località, insieme ai dati di contatto e alle indicazioni stradali per raggiungerli.

L'utente può effettuare varie operazioni sulla mappa, come ad esempio il trascinamento che mostra altre zone adiacenti senza ricaricare la nuova area, oppure l'ingrandimento o la riduzione dello zoom, oppure ancora la funzionalità **Street View** che verrà mostrata più avanti. L'utente può anche scegliere se visualizzare una semplice mappa con le sole indicazioni stradali, oppure una carta **satellitare** o **ibrida**.



## La API key

Per poter utilizzare le **Google API** e i relativi servizi dobbiamo ottenere una **API key**, una chiave che ci permetta di poter accedere al servizio, univocamente per ogni dominio. Tali servizi vengono offerti gratuitamente per un numero limitato di accessi al sito: esiste un limite di 15.000 generazioni di **geocode** nella pagina con la mappa generata dalle **API**.



◀ **Geocode** È la conversione di un indirizzo reale, come ad esempio "Via Milano, 132, Roma, Italia", in coordinate geografiche formate da latitudine e longitudine. Google impone che le proprie coordinate siano utilizzate solo su **Google Maps** e non per altri scopi. È possibile visualizzare un **geocode** solo su **Google Maps**. ▶

La procedura per ottenere la **Google API** è molto semplice: basta seguire le indicazioni riportate a partire dall'indirizzo: <https://console.developers.google.com>.

Per i nostri esempi utilizzeremo le **Google Maps JavaScript API** in quanto proporremo codice per dispositivi fissi.

### AREA digitale



Come ottenere la API key

La **Google key** è una stringa di circa 50 caratteri, come la seguente:

```
AIzaSyCFEZBrASwKXkxsxKoFw_aoSAYeqRf0VQE
```

Per poter utilizzare le **API** nei nostri esempi è sufficiente indicare l'**API key** ottenuta all'interno del tag **<SCRIPT>** della **<HEAD>** del documento **HTML**:

```
mappaConKey.html
1 <HTML><HEAD>
2 <LINK HREF="/maps/documentation/javascript/examples/default.css" REL="stylesheet">
3 <SCRIPT TYPE="text/javascript"
4 SRC="https://maps.googleapis.com/maps/api/
js?key=AIzaSyChTsh9mTgKr8BzRAdmRVFLBTDWG_SaIk&sensor=false">
5 </SCRIPT>
```

Il parametro **sensor** indica se l'applicazione usa un sensore per determinare la locazione dell'utente.

È anche possibile utilizzare le **API** senza inserire l'**API key**, come si può vedere dal segmento di codice riportato in figura, ma in tal caso è possibile effettuare un numero limitato di accessi.

```
mappaSenzaKey.html
1 <HTML><HEAD>
2 <LINK HREF="/maps/documentation/javascript/examples/default.css" REL="stylesheet">
3 <SCRIPT TYPE="text/javascript"
4 SRC="https://maps.googleapis.com/maps/api/js?key=0&sensor=false">
5 </SCRIPT>
```

## ■ Usare le API di Google Maps

Incominciamo a vedere come includere una mappa di **Google**: a tal fine è utile che questa venga collocata all'interno di un **<DIV>** di **HTML**, associando un ID a esso (nel nostro esempio con identificatore **mappa**), come illustrato dal seguente codice:

```
24 <BODY ONLOAD="inizializza();">
25 <DIV ID="mappa" STYLE="position:absolute;left:5px;top:9px;height:250px;width:550px"></DIV>
26 </BODY></HTML>
```

Abbiamo inserito una funzione `inizializza()` che viene eseguita al caricamento della pagina: in essa è presente il codice per visualizzare la **mappa** desiderata sullo schermo con le **opzioni** che volta per volta gli passeremo. A tal fine vengono definite e utilizzate due variabili globali:

- **coordinate**: oggetto di tipo `LatLng` che contiene la definizione delle coordinate del **punto centrale** (riga 8);
- **opzioni**: array associativo necessario per definire alcune proprietà che dovrà avere la nostra mappa (ad esempio zoom, tipologia ecc.).

```

6 <SCRIPT>
7 // definizione coordinate di centratura della mappa
8 var coordinate = new google.maps.LatLng(45.812, 9.0855);
9 // dichiarazione opzioni della mappa
10 var opzioni = {
11   // livello di zoom della mappa mostrata
12   zoom: 10,
13   center: coordinate,
14   // tipo di mappa (strade=ROADMAP, oppure HYBRID, SATELLITE, TERRAIN)
15   mapTypeId: google.maps.MapTypeId.ROADMAP
16 }

```

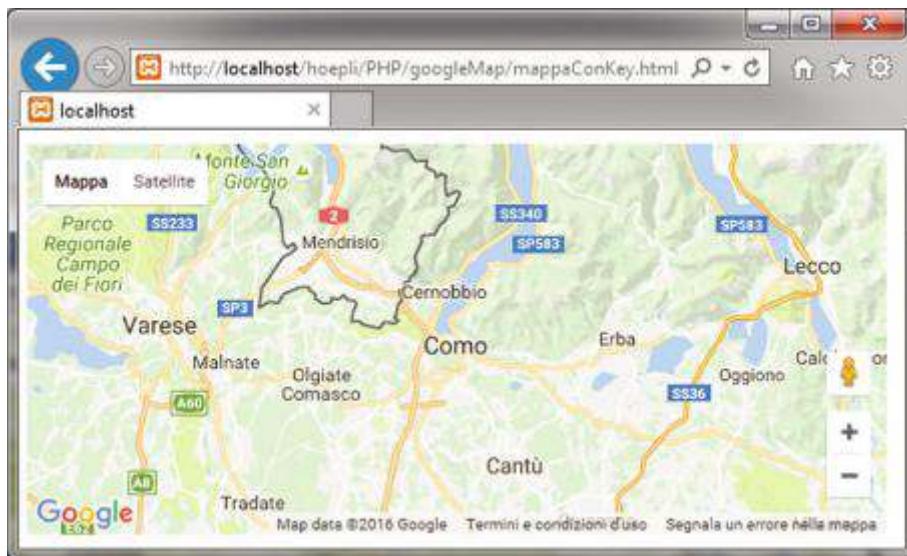
Questo array deve essere passato come parametro al costruttore dell'oggetto nella funzione `inizializza()`:

```

18 function inizializza(){
19   // creazione dell'oggetto mappa
20   var mappa = new google.maps.Map(document.getElementById('mappa'), opzioni);
21 }

```

L'esecuzione di questo semplice script ci visualizza la seguente schermata:



## google.maps.LatLng

La classe `LatLng` rappresenta una coordinata geografica, cioè formata da una longitudine e una latitudine, necessari per la geolocalizzazione (**geocoding**). Il costruttore `LatLng()` possiede due parametri: **latitudine** e **longitudine**

```
// definizione coordinate di centratura della mappa
var coordinate = new google.maps.LatLng(45.812, 9.0855);
```

Il database **localita** presente nella cartella **materiali** contiene la tabella “comuni” in cui sono memorizzati i nomi e le **geolocalizzazioni** di tutti i comuni italiani:

ID	name	lat	lng
1	Agliè	45.3681	7.7681
2	Airasca	44.9181	7.4855
3	Ala di Stura	45.3154	7.3026
4	Albano d'Ivrea	45.4339	7.9517
5	Alice Superiore	45.4599	7.7774
6	Almese	45.1169	7.3954
7	Alpette	45.4106	7.5808
8	Alpignano	45.0943	7.5244
9	Andezeno	45.0373	7.8731
10	Andrate	45.5285	7.8814
11	Annunziata	44.8438	7.2246

## google.maps.Map

Come abbiamo visto la seguente istruzione rappresenta una singola mappa all'interno di una pagina web:

```
28 // creazione dell'oggetto mappa vero e proprio
29 var map = new google.maps.Map(document.getElementById('mappa'),opzioni);
30
```

Il **costruttore** possiede due parametri:

- un **elemento HTML** che conterrà la mappa, generalmente un contenitore come ad esempio <SPAN> oppure <DIV>:

```
document.getElementById("mappa")
```

- la variabile strutturata **opzioni** (di tipo **object literal**), che contiene le **proprietà** della mappa:

```
// definizione coordinate di centratura della mappa
var coordinate = new google.maps.LatLng(45.812, 9.0855);
// dichiarazione opzioni della mappa
var opzioni = {
    // proprietà zoom indica di quanto zoomare la mappa
    zoom: 10,
    // proprietà center a cui assegnare un oggetto di classe LatLng per centrare la mappa
    center: coordinate,
    // proprietà mapTypeId definisce il tipo di terreno:
    // possibili alternative: ROADMAP | HYBRID | SATELLITE | TERRAIN
    mapTypeId: google.maps.MapTypeId.ROADMAP
}
```

## google.maps.Marker

Questa classe rappresenta un **marker** che può essere aggiunto alla mappa. Il **marker** è il segnale che mostra una posizione sulla mappa stessa:

Il costruttore **Marker()** possiede un solo parametro rappresentato dalle proprietà del marker, racchiuse in oggetto **literal**.



Il **marker** di default standard è quello che tutti conosciamo, rosso con la punta che indica la posizione. È anche possibile utilizzare **marker** personalizzati da definendo una variabile immagine nel modo seguente:

```
18 var immagine = { // nuova immagine per il marker (se vogliamo cambiarla)
19   url: 'icona.png', // file dell'immagine
20   size: new google.maps.Size(32, 32), // dimensione del marker in pixel
21   origin: new google.maps.Point(0, 0), // origine
22   anchor: new google.maps.Point(16, 16) // punto marker collocato in LatLang
23 };
```

Abbiamo diverse possibilità per collocare un **marker** sulla mappa: vediamo di seguito due metodi alternativi.



- A** creiamo dapprima la mappa e il **marker** e successivamente lo aggiungiamo sulla mappa mediante la funzione **addMarker()**:

```
33 marker = new google.maps.Marker({
34   map: map, // la mappa appena creata
35   title:"il castello", // stringa con descrizione del marker
36   position: castello, // posizioniamo il marker a queste coordinate
37   icon: immagine
38 });
39 map = new google.maps.Map(document.getElementById('map-canvas'), mapOptions);
40 addMarker(marker, map); // aggiunge il marker sulla mappa
```

- B** posizioniamo direttamente il **marker** con il metodo di **map setMap()**:

```
29 map = new google.maps.Map(document.getElementById('map-canvas'), mapOptions);
30 marker.setMap(map);
```

Naturalmente la posizione del **marker** viene definita tra le variabili iniziali dello script:

```
16 var napoli = new google.maps.LatLng(40.8333,14.25);
17 var castello = new google.maps.LatLng(40.8277,14.2478);
```

Per rimuovere il **marker** dalla mappa si richiama la funzione **setMap(null)**.

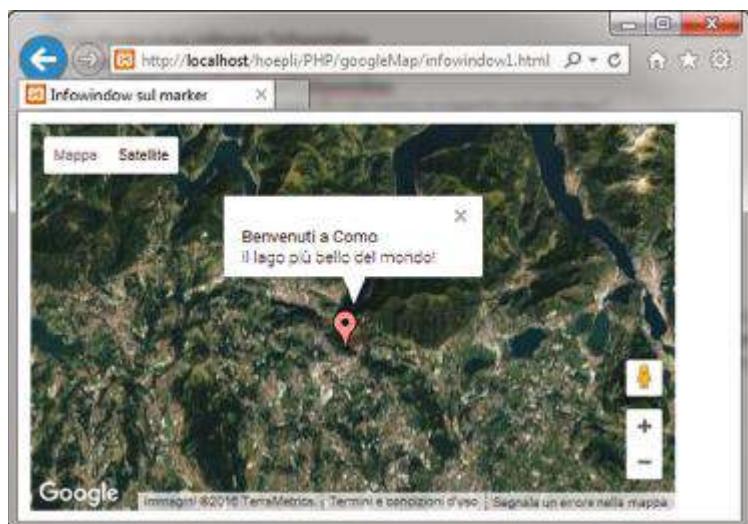
## google.maps.InfoWindow

La classe **InfoWindow** rappresenta una finestra di informazioni che può essere aggiunta direttamente sulla mappa oppure su un marker. Il **costruttore InfoWindow()** possiede un unico parametro: si tratta delle proprietà della InfoWindow, contenute in un **object literal**.

```
41 // testo html da visualizzare
42 stringa='<br><b>Benvenuti a Como</b><br>il lago più bello del mondo!';
43 // creazione istanza della InfoWindows
44 infowindow = new google.maps.InfoWindow({
45   content: stringa, // stringa HTML mostrata nella infowindow
46   position: coordinate // anche differenti da quelle del marker
47 });
48 // metodo open che mostra la marker con infowindow
49 infowindow.open(map, marker);
```

Per rendere visibile la **infoWindow** abbiamo utilizzato nelle istruzione di riga 49 il metodo **open()**, a cui passare due parametri: la **mappa** ed eventualmente il **marker** a cui ancorarla.

Eseguendo il codice completo otteniamo:



## ■ Associare eventi agli oggetti Google Maps

Alcuni oggetti delle **Google Maps API** sono costruiti in modo tale da rispondere a eventi legati in genere al mouse (ad esempio click, mouseover, dblclick ecc.). La classe **google.maps.Map** risponde a molti eventi tra i quali citiamo:

Evento	Argomenti	Descrizione
<b>click</b>	MouseEvent	Questo evento è scatenato dal click del mouse sulla mappa, ma non dal click su di un marker o su una <b>infoWindow</b>
<b>dblclick</b>	MouseEvent	Questo evento è scatenato dal doppio click sulla mappa.
<b>rightclick</b>	MouseEvent	Questo evento è scatenato dal click con il tasto destro del mouse.

L'argomento **MouseEvent** possiede la proprietà **LatLng**.

Per gestire tali eventi occorre registrare un ascoltatore (**listener**), mediante il metodo **addListener()** della classe **google.maps.event**.

```
20 // associazione tra evento click e marker
21 // in questo caso al click sul marker appare una infowindow
22 google.maps.event.addListener(marker, 'click', function(){
23   infowindow.open(map, marker);
24 });


```

Il metodo **addListener()** possiede 3 parametri:

- un oggetto (ad esempio di classe **Map**, oppure **Marker**);
- un evento da “catturare” (ad esempio **'click'**);
- una funzione (**anonima**) da invocare quando l’evento viene catturato (ad esempio per mostrare una **infoWindow**).

La documentazione completa delle classi di **Google Maps**, dei relativi metodi e proprietà è consultabile all'indirizzo:

<https://developers.google.com/maps/documentation/javascript/reference>

### ESEMPIO

Vediamo un esempio cliccando sulla mappa vengono visualizzate le coordinate del punto selezionato.

```
30  <TABLE><TR><TD>Latitudine</TD><INPUT TYPE="text" ID="lat" SIZE=15>
31  <TR><TD>Longitudine</TD><INPUT TYPE="text" ID="lng" SIZE=15></TABLE>
32  <DIV ID="mappa" STYLE="position:absolute;left:10px;top:70px;height:300px;width:500px"></DIV>
```

In questo esempio vogliamo mostrare una mappa di **Google** all'interno di un **<DIV>** di nome identificativo **id="mappa"** (riga 32). A ogni click sulla mappa viene mostrata la coordinata all'interno di due caselle di testo chiamate **lat** e **lng** (righe 30-31).

```
6  <SCRIPT>
7  var map;           // dichiarazione oggetto map di tipo globale
8  var opzioni = {   // dichiarazione opzioni della mappa
9    zoom: 10,
10   center: new google.maps.LatLng(45.8109,9.0885), // coordinate di Como
11   mapTypeId: google.maps.MapTypeId.ROADMAP          // tipo di mappa
12 }
13 //funzione richiamata all'apertura del body
14 function inizializza(){
15   // creazione dell'oggetto mappa vero e proprio
16   map = new google.maps.Map(document.getElementById('mappa'),opzioni);
17   // aggiunta di un ascoltatore di evento per il click
18   google.maps.event.addListener(map,'click', function(event){
19     // scrivo nella casella di testo la latitudine utilizzando i metodi lat()
20     document.getElementById('lat').innerText=event.latLng.lat();
21     // scrivo nella casella di testo la longitudine utilizzando i metodi lng()
22     document.getElementById('lng').innerText=event.latLng.lng();
23   });
24 }
25 </SCRIPT>
```

Dopo aver dichiarata la **API key** inizia il codice **JavaScript** con la dichiarazione una variabile globale di nome **map** che rappresenta l'oggetto della mappa e **opzioni** che è un array (**oggetto literal**) che contiene le proprietà necessarie per definire la mappa.

È utile dichiararla come variabile **globale** per renderla visibile a tutte le funzioni.

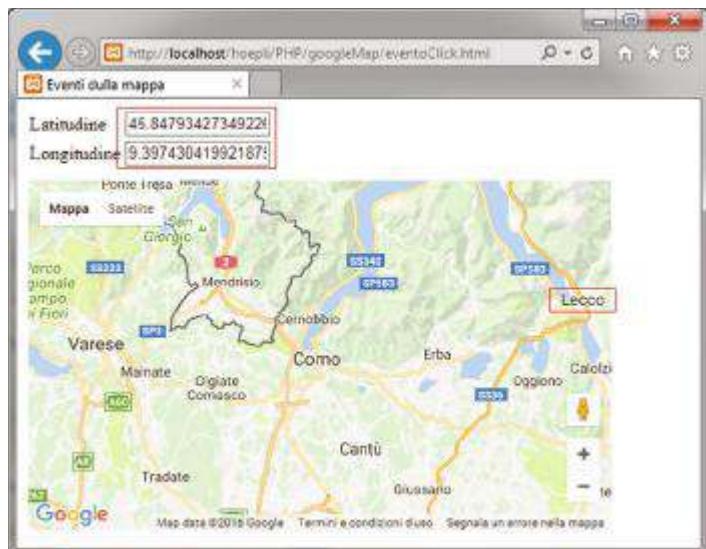
La funzione **inizializza()** (righe 14-24) serve per inizializzare la mappa di **Google** ed è associata all'evento **Load** del tag **<BODY>** di **HTML**.

L'istanza dell'oggetto **google.maps.Map** possiede due parametri:

- dove collocare la mappa (elemento **<DIV>** di nome '**mappa**');
- le **opzioni** contenute nell'array che descrivono la tipologia di mappa da visualizzare.

Aggiungiamo l'ascoltatore alla mappa per l'evento '**click**' e dichiariamo una **funzione anonima** che riceve come parametro l'argomento di nome **evento** di classe **MouseEvent** (riga 18): all'interno della funzione andiamo a leggere la latitudine e la longitudine del punto in cui è avvenuto il click sulla mappa attraverso i metodi **lat()** e **lng()** dell'oggetto **latLng** di **event** (riga 20-22).

Nell'esecuzione del codice possiamo notare che vengono inserite le coordinate nelle caselle di testo relative al punto dove è avvenuto il click sulla mappa, nel nostro caso sulla citta di [Lecco](#):



## ESEMPIO

In questo esempio vogliamo inserire un [marker](#) all'interno della mappa per poi associarvi un [evento](#) che mostri a video una finestra di informazioni (di classe [InfoWindow](#)). Lo [script](#) è formato dalla funzione [inizializza\(\)](#) che si occupa di mostrare la mappa, il marker e ascoltare l'evento.

Dopo l'intestazione e la [API key](#) vengono definite le variabili globali tra cui le [coordinate](#) da utilizzare per centrare la mappa ([riga 10](#)): come al solito viene creata la variabile [literal](#) chiamata [opzioni](#) che contiene le opzioni necessarie per la classe [Map](#) ([righe 11-15](#)) e un'altra variabile [literal](#) chiamata [image](#) necessaria per la creazione del marker ([righe 17-21](#)).

```

1 <HTML><HEAD>
2 <LINK HREF="/maps/documentation/javascript/examples/default.css" REL="stylesheet">
3 <SCRIPT TYPE="text/javascript"
4 SRC="https://maps.googleapis.com/maps/api/
js?key=AIzaSyChTsh9mTgKrBBzRADMRVFLTDWG_SaWk&sensor=false">
5 </SCRIPT>
6 <SCRIPT>
7 var marker;           // dichiarazione oggetto map di tipo globale
8 var map;              // dichiarazione variabile infowindow
9 var infowindow;       // funzione richiamata all'apertura del body
10 var coordinate = new google.maps.LatLng(43.1656,13.7242);
11 var opzioniMappa = {
12   zoom: 10,
13   center: coordinate,
14   mapTypeId: google.maps.MapTypeId.ROADMAP
15 }
16 var image = {
17   url: 'mark.png',           // simbolo del marker
18   size: new google.maps.Size(32, 32),
19   origin: new google.maps.Point(0, 0),
20   anchor: new google.maps.Point(0, 32)
21 };

```

La parte restante del codice istanzia un oggetto `map` di classe `google.maps.Map` e gli associa l'elemento HTML '`mappa`' quindi crea un oggetto `Marker`, in questo caso chiamato (con molta fantasia) `marker` e contestualmente ne inizializza gli attributi:

- ▷ le coordinate (proprietà `position`);
- ▷ la mappa in cui collocarlo (proprietà `map`);
- ▷ l'icona da mostrare (proprietà `icon`);
- ▷ il nome che appare passando il mouse sopra di esso (proprietà `title`) (righe 28-31).

```

23  function inizializza() {
24      // creazione dell'oggetto mappa vero e proprio
25      map = new google.maps.Map(document.getElementById('mappa'),opzioniMappa);
26      // creazione del marker
27      marker = new google.maps.Marker({
28          position: coordinate,
29          map: map,
30          icon: image,
31          title: 'Fermo'
32      });
33      // aggiunta ascoltatore sul marker
34      google.maps.event.addListener(marker,'click', function() {
35          // definizione della stringa HTML da visualizzare
36          var stringa=<br><b>Benvenuti a ">+marker.getTitle()+"</b><br>";
37          stringa+="Dista circa 6 km dal mare Adriatico, in una zona ad alta concentrazione ";
38          stringa+="demografica e incluso in un'area calzaturiera<BR><img src='fermo.jpg'>";
39          // istanza della InfoWindows
40          var infowindow = new google.maps.InfoWindow({
41              content: stringa      // stringa HTML mostrata nella infowindow
42          });
43          // metodo open che mostra la infowindow sul marker
44          infowindow.open(map, marker);
45      });
46  }
47  </SCRIPT>

```

La parte interessante è sicuramente legata all'utilizzo del metodo `ascoltatore addListener()` della classe `google.maps.event` (riga 34), che riceve tre parametri:

- ▷ l'oggetto `marker` sul quale intercettare gli eventi;
- ▷ l'evento da intercettare (in questo caso il `click`);
- ▷ la funzione anonima di risposta.

La funzione anonima si occupa di mostrare a video una `InfoWindow`: a tal fine dapprima memorizza in una stringa (`stringa`) il testo `HTML` da mostrare successivamente nella `InfoWindow` (righe 36-38). Successivamente avviene l'istanza dell'oggetto `InfoWindow` di classe `InfoWindow` (righe 40-42) dove nel costruttore inseriamo un unico argomento (`content`) che è la stringa da mostrare (riga 44).

La riga 44 conclude la funzione anonima legata all'evento sul marker: mostra la finestra invocando il metodo `open(map,marker)` sull'oggetto `InfoWindow`.

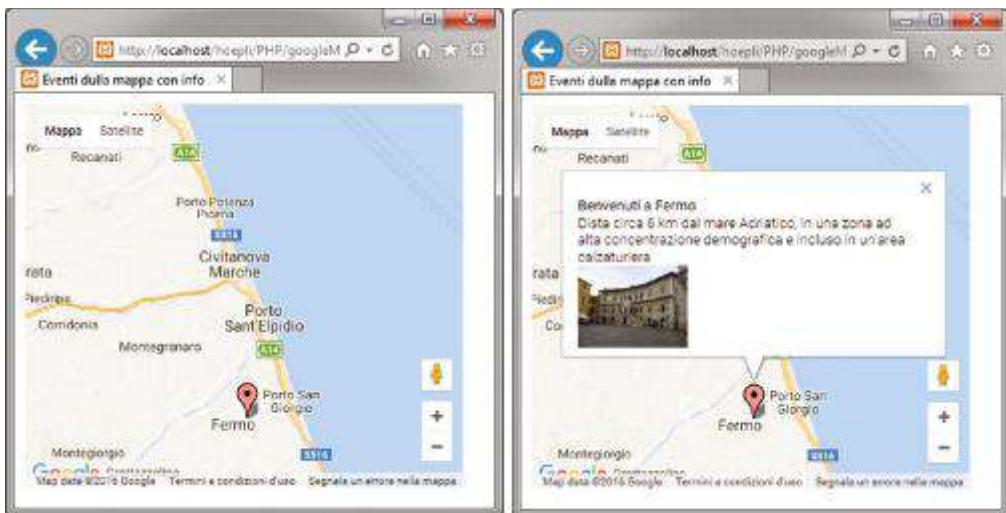
L'ultima parte riguarda la sezione `HTML` (righe 48-52). Il tag `<BODY>` richiama subito la funzione `inizializza()`, quindi mostra un riquadro `<DIV>` (`id='mappa'`) al quale è associato uno stile che lo identifica in una parte dello schermo.

```

48  </HEAD>
49  <TITLE>Eventi sulla mappa con info</TITLE>
50  <BODY onload="inizializza()">
51  <DIV ID="mappa" STYLE="position:absolute;left:10px;top:10px;height:350px;width:400px"></div>
52  </BODY></HTML>

```

L'esecuzione mostra la seguente mappa e cliccando sul marker appare una **InfoWindow**:



## Prova adesso!



**APRI IL FILE** `eventiClickInfo.html`

- 1 Modifica il codice dell'esempio in modo che possa mostrare i marker per tutti i capoluoghi della tua regione. A ogni click sul marker devono apparire le coordinate in cui si trova e alcune immagini inserite all'interno di una **InfoWindow**.
- 2 Prova poi a verificare le coordinate di casa tua e aggiungi un marker che mostri la sua foto e un messaggio all'interno di un <DIV> diverso da quello della mappa.

- Utilizzare le classi `Map`, `Marker`, `event`, `InfoWindow`
- Utilizzare gli ascoltatori di evento
- Leggere le coordinate geografiche



◀ **Calcolo del percorso** Il calcolo del percorso minimo tra due punti della mappa avviene per mezzo dell'algoritmo di **Dijkstra** che assegna come pesi degli archi il tempo di percorrenza in relazione a diversi elementi tra cui:

- ▶ orario;
- ▶ traffico;
- ▶ tipo di strada;
- ▶ lunghezza in km. ▶

Una delle funzionalità più interessanti delle **API di Google Maps** è certamente legato alla misura della distanza tra due località, chiamato **directions**, che effettua il ▲ **calcolo del percorso** ▼ tra due località e visualizza sulla mappa una linea, che indica il cammino, corredata da una serie di marker con descrizione testuale.

La ricerca del percorso migliore avviene per mezzo di due classi, **Directions Renderer** e **DirectionsService**: la prima è delegata alla visualizzazione della direzione, mentre la seconda allo svolgimento del calcolo delle direzioni.

Dopo aver creato una nuova istanza della classe **DirectionsService**, dobbiamo utilizzare il metodo **DirectionsService.route()** che possiede due argomenti, la variabile **literal DirectionsRequest** e la funzione anonima eseguita non appena la richiesta è stata completata.

L'oggetto `DirectionsRequest` possiede le seguenti proprietà:

- ▶ `origin` (obbligatorio) specifica l'indirizzo di partenza da cui calcolare il percorso;
- ▶ `destination` (obbligatorio) specifica l'indirizzo di arrivo del percorso;
- ▶ `travelMode` (obbligatorio) specifica la modalità di trasporto (`DRIVING`, `WALKING`, `BICYCLING`);
- ▶ `waypoint [ ]` (opzionale) specifica un array (`DirectionsWaypoints`) di punti di passaggio obbligati nel calcolo del percorso.

Le proprietà `origin` e `destination` possono contenere valori specificati come stringhe oppure come oggetti di classe `latLng`.

La funzione anonima restituisce un valore numerico (codice) `DirectionsStatus` con l'esito della richiesta e un oggetto di classe `DirectionsResult`, in formato `JSON`, che presenta una struttura articolata contenente le indicazioni per la localizzazione dei punti di arrivo e partenza, oltre ad altre informazioni come ad esempio la durata e la lunghezza del percorso.

La classe `DirectionsRenderer` viene utilizzata per visualizzare il percorso ottenuto: l'istanza di questa classe, tramite il metodo `setMap()`, associa a una mappa i risultati da mostrare. Sempre su questa istanza dobbiamo chiamare il metodo `setDirections()`, che accetta come argomento l'oggetto della risposta `DirectionsResult` e che disegnerà fisicamente il percorso sulla mappa.

Il metodo `setPanel()` infine mostra le indicazioni testuali del percorso, collocabili all'interno di un riquadro (`<DIV>`).

### ESEMPIO

In questo esempio vogliamo effettuare una ricerca di un percorso stradale utilizzando le classi messe a disposizione dalle [API di Google](#). Dopo aver inizializzato la pagina e inserito le [API Key](#):

```
percorso.html
1 <HTML><HEAD>
2 <LINK HREF="/maps/documentation/javascript/examples/default.css" REL="stylesheet">
3 <SCRIPT TYPE="text/javascript"
4 SRC="https://maps.googleapis.com/maps/api/
js?key=AIzaSyChTsh9mTgKrBBzRAdmRVFLBTDWG__5awK&sensor=false">
5 </script>
```

nella prima parte dell'esempio utilizziamo i fogli di stile necessari per mostrare due `<DIV>` 'mappa' e 'box' che conterranno rispettivamente la mappa di [Google](#) e il percorso in formato testuale.

```
6 <STYLE>
7 div.box {
8   position: absolute;
9   left: 3px;
10  top: 10px;
11  height: 540px;
12  width: 400px;
13  background-color: azure;
14  overflow: auto;
15 }
16 div.mappa {
17   position: absolute;
18   left: 420px;
19   top: 10px;
20   height: 540px;
21   width: 550px;
22   background-color: yellow;
23 }
24 </STYLE>
```

Lo script **JavaScript** è racchiuso in una funzione chiamata **init()** che viene richiamata all'apertura della pagina. Inizialmente vengono istanziati gli oggetti di classe **DirectionService** e **DirectionRenderer**, necessari rispettivamente per la ricerca del percorso e per la collocazione dei risultati nella mappa (**righe 28-30**). Successivamente viene dichiarata la variabile **literal** che contiene le **opzioni** della mappa (**righe 32-34**) e quindi la variabile **literal request** che contiene le proprietà di inizio e fine percorso di ricerca e di tipo di percorso (**origin, destination, travelMode**) (**righe 38-42**).

```

25 <SCRIPT TYPE="text/javascript">
26 function init(){
27   // dichiarazione oggetto DirectionService, per la ricerca del percorso
28   var direzione = new google.maps.DirectionsService();
29   // dichiarazione oggetto DirectionRenderer, per visualizzare i risultati
30   var mostra = new google.maps.DirectionsRenderer();
31   // dichiarazione opzioni della mappa
32   var opzioni = {
33     zoom: 10,
34     mapTypeId: google.maps.MapTypeId.HYBRID
35   }
36   // definizione delle proprietà necessarie per l'oggetto direzione:
37   // inizio, fine percorso e metodo di ricerca (WALKING o DRIVING)
38   var request = {
39     origin: 'Milano',
40     destination: 'Brescia',
41     travelMode: google.maps.DirectionsTravelMode.DRIVING
42   };

```

Il metodo **route()** calcola il percorso ricevendo due parametri, la variabile **literal request** e la funzione anonima che riceve due parametri: **response** e **status** (**righe 46-51**).

```

43   // metodo route che calcola il percorso ricevendo due parametri
44   // la variabile literal request che contiene da inizio e fine percorso
45   // e la funzione anonima che riceve due parametri: response e status
46   direzione.route(request, function(response, status){
47     if (status == google.maps.DirectionsStatus.OK){           //se lo status è OK
48       //viene inserita la traccia del risultato con il metodo setDirections
49       mostra.setDirections(response);
50     }
51   });

```

A questo punto viene istanziato l'oggetto di classe **Maps** e associato al **<DIV>** di nome '**mappa**' (**riga 53**) e viene invocato:

- il metodo **setMap()** che mostra nella mappa l'output dei risultati (**riga 55**);
- il metodo **setPanel()** che mostra nel **<DIV>** l'output dei risultati in forma testuale (**riga 57**).

```

52   //definizione oggetto di classe Maps e associazione al <div> di nome 'mappa'
53   map = new google.maps.Map(document.getElementById("mappa"), opzioni);
54   //metodo setMap mostra nella mappa l'output dei risultati
55   mostra.setMap(map);
56   //metodo setPanel mostra nel <div> l'output dei risultati in forma testuale
57   mostra.setPanel(document.getElementById('elenco'));
58 }
59 </SCRIPT>

```

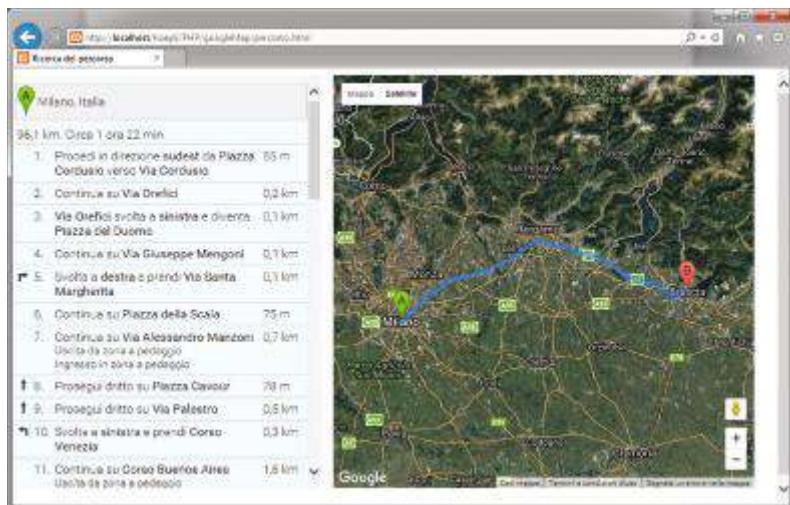
La parte finale della pagina è assai semplice e si compone di due **<DIV>** che conterranno i risultati:

```

60 </HEAD>
61 <TITLE>Ricerca del percorso</TITLE>
62 <BODY onload="init()">
63 <div class="box" id="elenco"></DIV></P>
64 <DIV id="mappa" class="mappa"></DIV>
65 </BODY></HTML>

```

L'esecuzione dello script come possiamo notare mostra il percorso stradale necessario per collegare le due località (Milano e Brescia):



## Prova adesso!



**APRI IL FILE** `percorso.html`

- Modifica il codice in modo che l'utente possa selezionare attraverso 3 campi radio se effettuare il percorso di tipo stradale, pedonale oppure ciclabile.
- Aggiungi due caselle di testo in modo che l'utente possa selezionare l'origine e la destinazione.
- Aggiungi due caselle a tendina, usando il tag SELECT, che mostrino i due comuni leggendoli dal database `localita`.

- Usare la API di Google Maps
- Usare le directions
- Usare MySQL e PHP

## ■ Lo Street View panorama

Le **API di Google Maps** possiedono un oggetto assai utile per fornire una visione stradale delle mappe: si tratta di **Google Street View**.



◀ **Google Street View** È una caratteristica di **Google Maps** che fornisce viste panoramiche a 360° in orizzontale e a 290° in verticale lungo le strade, a una distanza di circa 10 metri l'una dall'altra, consentendo agli utenti di vedere parti di varie città del mondo a livello del terreno. ►

La classe esatta si chiama `google.maps.StreetViewPanorama` e consente di posizionare l'omino giallo in modo tale che esso possa essere trascinato in un punto qualsiasi della mappa. Le proprietà di questa classe sono molteplici, le abbiamo sintetizzate come segue.

### position

È la proprietà di tipo `LatLng` e rappresenta la coordinata geografica esatta in cui vogliamo mostrare la mappa di **Street View**.

### navigationControl

È una proprietà di tipo `boolean`, che di default è impostata a `true` e permette di attivare o disattivare i controlli per la navigazione.

**pov**

Significa **Point Of View** e rappresenta la proprietà obbligatoria che indica da quale punto stiamo osservando la scena nella mappa **Street View**. È formata a sua volta dalle seguenti proprietà obbligatorie:

**heading**

Imposta l'angolazione della telecamera, di default è impostata a nord (0 gradi).

**pitch**

Imposta l'altezza della telecamera 90° al massimo -90° al minimo.

**zoom**

Imposta lo zoom con un valore compreso tra 0 e 20.

**ESEMPIO**

In questo esempio collocheremo due finestre nel nostro `<DIV>` in cui mostrare all'utente una mappa di **Google Maps** e accanto una mappa di **Street View**. La navigazione dell'omino giallo avverrà nella mappa di **Google Maps**, mentre la mappa di **Street View** si adatterà di conseguenza. Vediamo il codice, assai semplice, di questo esempio, prima di tutto vediamo la dichiarazione degli stili assegnati ai due riquadri `<DIV>` che si chiamano, rispettivamente 'street' e 'mappa':

```

1 <HTML><HEAD>
2 <STYLE>
3 div.street {
4   position:absolute;
5   left:10px;
6   top:16px;
7   height:440px;
8   width:400px;
9   background-color:azure;
10 }
11 div.mappa {
12   position:absolute;
13   left:420px;
14   top:16px;
15   height:440px;
16   width:550px;
17   background-color:azure;
18 }
19 </STYLE>

```

Il codice è racchiuso nella funzione `init()` richiamata all'avvio della pagina. La funzione innanzi tutto dichiara le opzioni della mappa (righe 29-34), quindi viene, come di consueto, istanziato l'oggetto di classe `Map` (riga 35).

```

24 <SCRIPT TYPE= "text/javascript">
25 //funzione eseguita all'apertura della pagina HTML
26 function init(){
27   var coordinate = new google.maps.LatLng(41.8914,12.4923);
28   //Definisco le opzioni della mappa
29   var opzioniMap = {
30     zoom: 13,
31     center: coordinate,
32     mapTypeId: google.maps.MapTypeId.ROADMAP
33   }
34   // istanzio l'oggetto di classe Map
35   var map = new google.maps.Map(document.getElementById('mappa'),opzioniMap);

```

A questo punto vengono dichiarate le opzioni della **StreetView** (righe 37-42).

L'oggetto di classe **StreetViewPanorama** è istanziato nella **riga 45** e collegato alla mappa alla **riga 47**.

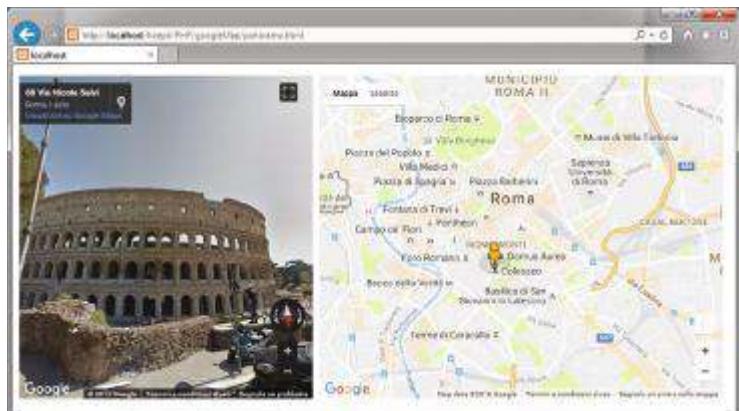
```

36 // opzioni della StreetView
37 var opzioniSV = {
38   position: coordinate,
39   pov: {
40     heading: 34,
41     pitch: 10
42   }
43 };
44 // istanza oggetto di classe StreetViewPanorama
45 var panorama = new google.maps.StreetViewPanorama(document.getElementById('strada'), opzioniSV);
46 // inserimento della street view nella mappa
47 map.setStreetView(panorama);
48 }
49 </SCRIPT>

```

Se togliessimo la riga 47 non avremmo più il collegamento tra le due mappe, cioè la mappa di destra diventerebbe indipendente rispetto a quella di sinistra.

L'esecuzione dello script mostra due riquadri, muovendo l'omino giallo nel riquadro di destra otteniamo la corrispondente mappa di **Street View** nel riquadro di sinistra:



Google Maps is a web mapping service developed by Google. It offers satellite imagery, street maps, 360° panoramic views of streets (Street View), real-time traffic conditions (Google Traffic), and route planning for traveling by foot, car, bicycle (in beta), or public transportation.

Google Maps began as a C++ desktop program designed by Lars and Jens Eilstrup Rasmussen at Where 2 Technologies. In October 2004, the company was acquired by Google, which converted it into a web application. After additional acquisitions of a geospatial data visualization company and a realtime traffic analyzer, Google Maps was launched in February 2005. The service's front end utilizes JavaScript, XML, and Ajax. Google Maps offers an API that allows maps to be em-

bedded on third-party websites, and offers a locator for urban businesses and other organizations in numerous countries around the world. Google Map Maker allows users to collaboratively expand and update the service's mapping worldwide.

Google Maps' satellite view is a "top-down" or "birds eye" view; most of the high-resolution imagery of cities is aerial photography taken from aircraft flying at 800 to 1,500 feet (240 to 460 m), while most other imagery is from satellites. Much of the available satellite imagery is no more than three years old and is updated on a regular basis.

Google Maps uses a close variant of the Mercator projection, and therefore cannot accurately show areas around the poles.



# Verifichiamo le competenze

## Problemi

### Crea gli script che risolvano i problemi proposti

- 1 Mostra, all'interno di una pagina Web, due <DIV>: entrambi mostrano una mappa, a ogni click sul primo deve comparire nella seconda mappa un marker nella stessa posizione.
- 2 Mostra un form e una mappa di Google. Nel form appare una casella a discesa (<SELECT>) che mostra all'utente l'elenco delle regioni. Quando l'utente seleziona una regione mostra, sulla mappa, un elenco di marker su tutti i capoluoghi di provincia di quella regione prelevandoli dal database **localita**.
- 3 Crea una tabella, prendendo spunto dal database **localita**, che contenga le coordinate delle abitazioni dei tuoi compagni di classe. Mostra sulla mappa un marker per ciascuna abitazione della tua classe, facendo click su di essi deve apparire un InfoWindow con la foto e alcuni dati dell'alunno.
- 4 Mostra un form che consenta di effettuare la ricerca di una città dal database **localita**. Quando la località viene trovata deve apparire in un <DIV> che contiene una mappa di Google la visione dello Street View.
- 5 Mostra un form che consenta all'utente di cercare due località dal database **localita**. Nella mappa posta in un <DIV> devi mostrare il percorso stradale che collega i due luoghi.
- 6 Modifica l'esempio precedente cercando il percorso da effettuare a piedi.
- 7 Realizza uno script dove l'utente seleziona tre tappe di una gita cliccando sulle tre località: indica il percorso più corto per collegare le tre località visualizzandolo sulla cartina.
- 8 Modifica lo script precedente facendo selezionare dall'utente cinque città dal database **localita**, controllando che appartengano a regioni differenti; indica successivamente:
  - Il percorso più corto per collegare tutte le località;
  - un percorso alternativo facendo selezionare la città di partenza e quella di arrivo.Naturalmente devono essere escluse anche le città che appartengono alle isole.

# ESERCITAZIONI DI LABORATORIO 2

## COMUNICAZIONE CLIENT-SERVER CON AJAX

**AREA** *digitale*



Lab. 1 Installazione di EasyPHP

### ■ AJAX

AJAX, acronimo di **Asynchronous JavaScript and XML**, è una tecnica di sviluppo per la realizzazione di applicazioni Web interattive. Mediante **AJAX** possiamo consentire l'aggiornamento di una sezione di pagina **HTML** senza che questa venga caricata nuovamente dal server tramite richieste http effettuate in JavaScript.

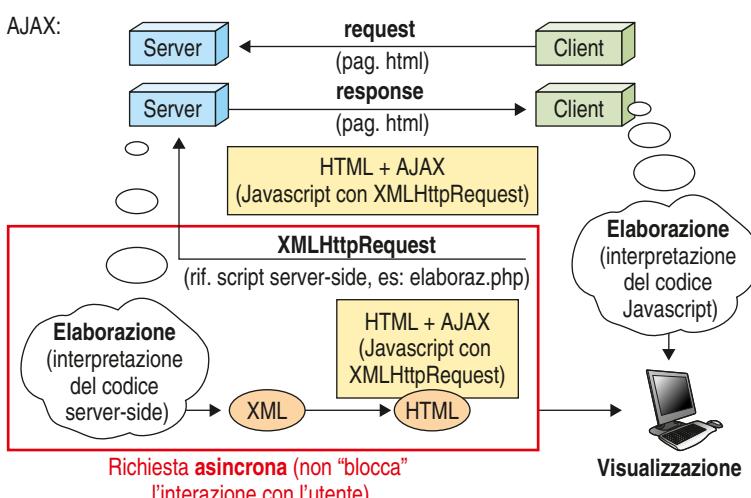
Il core di ▶ **AJAX** ▶ è rappresentato da una specifica classe chiamata **XMLHttpRequest**: l'istanza di questa classe varia purtroppo però a seconda del browser utilizzato, impareremo nell'esempio che segue, come istanziare un oggetto di questa classe per versioni di browser differenti.



- ◀ **AJAX** Questo termine è stato coniato per la prima volta nel febbraio del 2005 da **Jesse James Garrett**, per descrivere un insieme di applicazioni web dinamiche basate sull'interazione tra diverse tecnologie:
- ▶ **(X)HTML** e **CSS** per la visualizzazione della pagina;
- ▶ **DOM**, modificato attraverso Javascript, per offrire dinamicità alle pagine web;
- ▶ **XMLHttpRequest**, che consente al browser e al server di comunicare senza che la pagina venga ricaricata, permettendo la creazione di pagine web dinamiche più veloci. ►

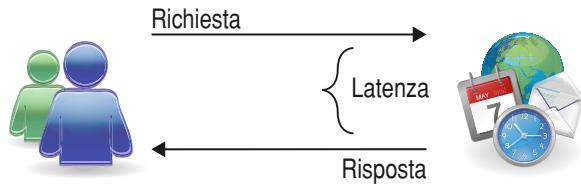
Lo schema a fianco illustra il ruolo di questa tecnologia nel tipico dialogo **client-server** attraverso il Web e il protocollo **HTTP**.

La classe principale di **AJAX** è **XMLHttpRequest** che consente di effettuare, con il protocollo **HTTP**, la richiesta di una risorsa a un **Web server** in modo indipendente dal browser. Nella richiesta è possibile inviare informazioni, ove opportuno, sotto forma di variabili di tipo **GET** o di tipo **POST** in maniera simile all'invio dati di un form.

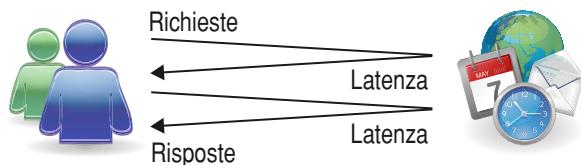


La richiesta è **asincrona**, il che significa che non bisogna necessariamente attendere che sia stata ultimata per effettuare altre operazioni, stravolgendo sotto diversi punti di vista il flusso dati tipico di una pagina Web.

Generalmente infatti il flusso è racchiuso in due passaggi alla volta, **richiesta** dell'utente (link, form o refresh) e **risposta** da parte del **server**, per poi passare, eventualmente, alla nuova richiesta da parte dell'utente. La figura a fianco rappresenta una richiesta a un server con **latenza** e **risposta**.



Inevitabile l'inconveniente del tempo di attesa/latenza necessario al server per produrre la risposta. Con **AJAX** si perde questo schema: l'utente all'interno della stessa pagina può effettuare richieste multiple completamente indipendenti; teoricamente è possibile effettuare decine di richieste simultanee al server per operazioni differenti con o senza controllo da parte del navigatore. Il nuovo schema è quello a fianco.



Rispetto al modello precedente rimane solo il tempo di **attesa della risposta**, che passa tuttavia in secondo piano rispetto ai vantaggi offerti da questa tecnologia. Questo tempo è anche uno dei maggiori problemi dell'utilizzo di **AJAX**, sia per gli sviluppatori sia per i navigatori: i primi potrebbero trovarsi in difficoltà qualora l'operazione asincrona dovesse attendere assolutamente una risposta al fine di completare una serie di operazioni più o meno sensibili, mentre i secondi potrebbero non avere idea di cosa stia accadendo alla pagina chiudendola ignari di aver richiesto un'informazione.

L'oggetto di classe **XMLHttpRequest** si aspetta una risposta dal server, dopo una chiamata, che non deve essere necessariamente di tipo **XML** ma che può essere semplicemente testuale (come nel nostro esempio). Una delle prime applicazioni a utilizzare questo componente è stato il servizio **Google suggest** che consente di ricercare una parola attraverso continui suggerimenti ricevuti in modo asincrono, proprio grazie alla tecnologia **AJAX**.

Riportiamo in una tabella i principali **metodi** della classe **XMLHttpRequest**.

NOME METODO	DESCRIZIONE
open( metodo, URL ) open( metodo, URL, async ) open( metodo, URL, async, userName ) open( metodo, URL, async, userName, password )	Specifica il metodo, l'URL e altri parametri opzionali per la richiesta. Il parametro <b>metodo</b> può assumere valore di "GET", "POST", oppure "PUT" ("GET" è utilizzato quando si richiedono dati, mentre "POST" è utilizzato per inviare dati). Il parametro <b>URL</b> può essere sia relativo che assoluto. Il parametro " <b>async</b> " specifica se la richiesta deve essere gestita in modo asincrono oppure no: ▶ con "true" lo script può proseguire l'elaborazione senza aspettare la risposta dopo il metodo <code>send()</code> ; ▶ con "false" lo script è costretto ad aspettare una risposta dal server prima di continuare.
send( content )	Invia la richiesta.
setRequestHeader( chiave, valore)	Aggiunge la coppia chiave/valore alla richiesta da inviare.

La seguente tabella riporta i principali **attributi** della classe **XMLHttpRequest**.

NOME PROPRIETÀ	DESCRIZIONE
onreadystatechange	Gestore dell'evento lanciato a ogni cambiamento di stato.
readyState	Restituisce lo stato corrente dell'istanza di XMLHttpRequest: 0 = non inizializzato, 1 = aperto, 2 = richiesta inviata, 3 = risposta in ricezione, 4 = risposta ricevuta.
responseText	Restituisce la risposta del server in formato stringa.
status	Restituisce il codice HTTP restituito dal server (per esempio 404 per "Not Found" e 200 per "OK").

### ESEMPIO *Simulazione di Google suggest*

Si vuole realizzare un programma che aiuta l'utente a ricercare un nome compreso in un insieme di nomi, visualizzando a ogni carattere inserito solo l'elenco di quelli che “sono possibili soluzioni”.

Il programma è diviso in due sezioni, **client** e **server**, la prima contenuta nel file **esempioAjax.html** che utilizza **HTML**, **JavaScript** e **AJAX** e la seconda nel file **php\_con\_ajax.php** che utilizza il solo linguaggio **PHP**.

Il funzionamento è piuttosto semplice: la pagina **HTML** riceve dall'utente una stringa che viene ricercata nell'array di nomi presente nella seconda pagina attraverso **AJAX**, dove viene effettuata la ricerca nell'array e restituito un valore: con **AJAX** la comunicazione è assai efficiente in quanto la richiesta al **server** avviene a ogni tasto premuto, simulando il funzionamento di **Google suggest**.

Descriviamo il codice a partire della pagina **esempioAjax.html**: nella ultime righe del file abbiamo la sezione **HTML** che mostra una casella di testo in cui l'utente deve inserire la stringa da cercare (**riga 45**). A ogni pressione del tasto, mediante l'evento **onkeyup**, viene richiamata la funzione **mostra()** alla quale viene passato il testo digitato fino a quell'istante; un tag contenitore (**riga 47**) servirà per posizionare i risultati.

```

42 <BODY>
43 <><B>Scrivere nella casella le iniziali:</B></P>
44 <FORM>
45 Nome: <INPUT TYPE="text" onkeyup="mostra(this.value)">
46 </FORM>
47 <P>Nominativi trovati: <DIV ID="risposta"></DIV></P>
48 </BODY>
49 </HTML>
```

La funzione **mostra()**, se la stringa ricevuta non esiste (**righe 5-8**), colloca una stringa vuota nel **<DIV> 'risposta'**. A questo punto viene istanziato l'oggetto di classe **XMLHttpRequest** nei diversi modi a seconda delle versioni del browser (**righe 10-15**).

```

3 // funzione attivata ad ogni pressione di un tasto sulla casella di testo
4 function mostra(str){
5   if (str.length == 0){ // se stringa letta vuota viene inviata risposta blank
6     document.getElementById("risposta").innerHTML = "";
7     return;
8   }
9   // creazione oggetto XMLHttpRequest per vari tipi di Browser
10  if (window.XMLHttpRequest){ // browser IE7+, Firefox, Chrome, Opera, Safari
11    xmlhttp = new XMLHttpRequest();
12  }
13  else{ // browser IE6, IE5
14    xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
15  }
```

Nella “*funzione anonima*” (righe 18-31) della proprietà `onreadystatechange` viene scritto il codice che verrà processato quando il server risponderà alla chiamata: si tratta di un codice che viene eseguito in modo **asincrono**, solo quando il server restituisce qualcosa, come una sorta di interrupt.

```

16 // dopo una richiesta serve una funzione x ricevere i dati restituiti dal server
17 // onreadystatechange contiene la funzione che processera la risposta del server
18 xmlhttp.onreadystatechange = function(){
19 // readyState contiene lo stato della response del server. Ogni volta che
20 // readyState cambia, la funzione onreadystatechange viene eseguita.
21 // 0 La richiesta non è inizializzata
22 // 1 È stabilita la connessione col server
23 // 2 La richiesta è ricevuta
24 // 3 La richiesta è processata
25 // 4 La richiesta è completa
26 // se status=200 OK - 404 Not Found
27 if (xmlhttp.readyState == 4 && xmlhttp.status == 200){
28 // i dati trasmessi dal server sono ricavati da responseText (tipo stringa)
29 document.getElementById("risposta").innerHTML = xmlhttp.responseText;
30 }
31 }
```

Dopo aver verificato le proprietà `readyState` e `status`, che ci informano se il server ha risposto (riga 27) viene collocato nel `<DIV>` il risultato ottenuto rappresentato dalla proprietà `responseText` (riga 29).

L'ultimo segmento di codice deve essere posizionato dopo la sezione `onreadystatechange` ma viene eseguito per primo.

```

32 // il metodo open() ha 3 parametri:
33 // -il primo definisce quale metodo usare (GET/POST)
34 // -il secondo è l'url dove risiede lo script server-side
35 // -il terzo (booleano) specifica che la richiesta deve essere asincrona
36 xmlhttp.open("GET", "php_con_ajax.php?stringa=" + str, true);
37 // send() trasmette effettivamente la richiesta al server.
38 // senza parametri si indica POST - per GET mettere i parametri tra parentesi
39 xmlhttp.send();
40 }
```

Il metodo `open()` (riga 36), infatti, imposta l'invio della richiesta al **server**: possiamo notare che tra i suoi parametri vi sono il metodo di invio (**GET** o **POST**) e la pagina a cui inviare la richiesta, in questo caso `php_con_ajax.php`; il metodo `send()` (riga 39) realizza infine l'invio effettivo della richiesta.

Descriviamo ora il codice della pagina `php_con_ajax.php` che riceve la richiesta asincrona.



```

1 <?php
2 //array che contiene i nomi
3 $a = array("Anna", "Brigitte", "Claudio", "Carmela", "Evelina", "Franco", "Giovanna"
, "Luca", "Matteo", "Gabriele", "Marta", "Martina", "Manuela", "Michela", "Monica", "
Nanni", "Pamela", "Paolo", "Paola", "Pietro", "Vincenzo", "Alessandro", "Riccardo",
"Chiara", "Eva", "Marina", "Maddalena", "Marco", "Michelle", "Antonio", "Giuseppe",
"Lisa", "Elisabetta", "Giuliana", "Arnoldo", "Uberto", "Umberto");
4 // salvo campo GET letto dalla pagina precedente
5 $q = $_GET["stringa"];
6 $response = "";
```

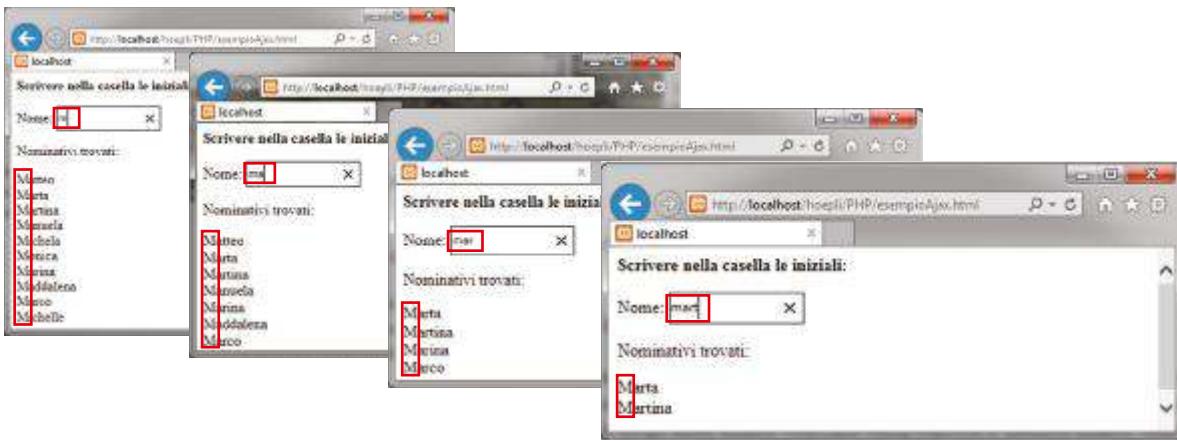
Nella prima parte viene dichiarato un array che contiene l'elenco dei nomi su cui cercare (riga 3).

Successivamente viene effettuata la ricerca (righe 8-18) (su cui non ci soffermeremo con la descrizione) e restituito l'elenco dei nomi separati dal tag <BR>: come possiamo notare la restituzione avviene con una normale echo (riga 27).

```

7 // verifica se campo esiste
8 if (strlen($q) > 0){
9     $risposta = "";
10    // ciclo di ricerca nell'array
11    for($i = 0; $i < count($a); $i++){
12        // controllo se nome trasformato in minuscolo è stato trovato
13        if (strtolower($q) == strtolower(substr($a[$i], 0, strlen($q)))){
14            // viene aggiunto il nome alla stringa separandolo dal tag di invio a capo
15            $risposta .= $a[$i] . "<BR>";
16        }
17    }
18 }
19 // verifico se $risposta è vuota restituisco stringa "Nessun nome trovato"
20 if ($risposta == ""){
21     $response = "Nessun nome trovato";
22 }
23 else{ //assegnazione stringa di risposta
24     $response = $risposta;
25 }
26 // invio risposta alla pagina chiamante
27 echo $response;
28 ?>
```

Caricando la pagina [esempioAjax.html](#) osserviamo come a ogni cambiamento della parola scritta nella casella di testo venga mostrato un elenco che si aggiorna in tempo reale:



**Prova adesso!**



[APRI IL FILE](#) `esempioAjax.html` e `php_con_ajax.php`

- Usare l'oggetto XMLHttpRequest
- Usare PHP

- 1 Modifica il codice dell'esempio in modo tale che il nome venga selezionato da un elenco a tendina.
- 2 Modifica il codice della pagina PHP dichiarando a inizio codice una matrice formata da nome e cognome. Il nome ricevuto dallo script deve essere ricercato nella matrice e devono essere restituiti i cognomi corrispondenti.

# ESERCITAZIONI DI LABORATORIO 3

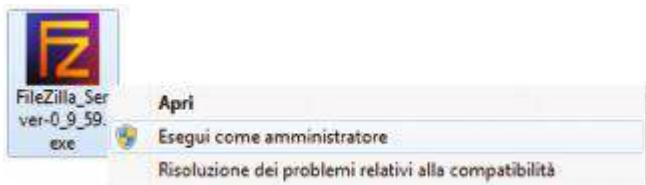
## CONNESSIONE FTP CON UNO SCRIPT PHP

### ■ Il server FTP FileZilla

In questa lezione di laboratorio installiamo un **server FTP** (**FileZilla Server**) per connetterci a esso mediante uno script **PHP**. Lo script mostra sul browser un form che consente, all'utente che vuole caricare dei file sul server, di selezionare il file ed effettuare l'**Upload** dopo aver fornito i dati necessari per la sua autenticazione (**indirizzo del server, porta FTP, nome utente e password**).

Descriviamo la procedura per installare **FileZilla** in versione **server**.

- 1 Per prima cosa procuriamoci **FileZilla** in versione **Server** scaricabile gratuitamente da Internet (o dal sito [hoepliscuola](#), nella cartella **materiali**). Una volta scaricato facciamo click sull'icona del programma di installazione procedendo come amministratore della macchina:



- 2 Nella prima schermata il programma mostra il contratto, procediamo facendo click su **I Agree**, in quanto il software è di tipo **freeware**:



- 3 Selezioniamo i componenti da installare: lasciamo quanto proposto di default.



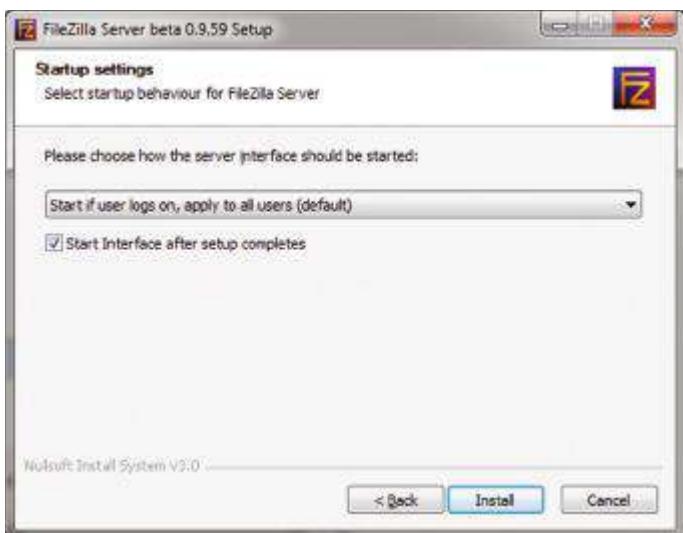
- 4 Adesso scegliamo dove installare il pacchetto, in questo caso lasciamo la directory indicata.



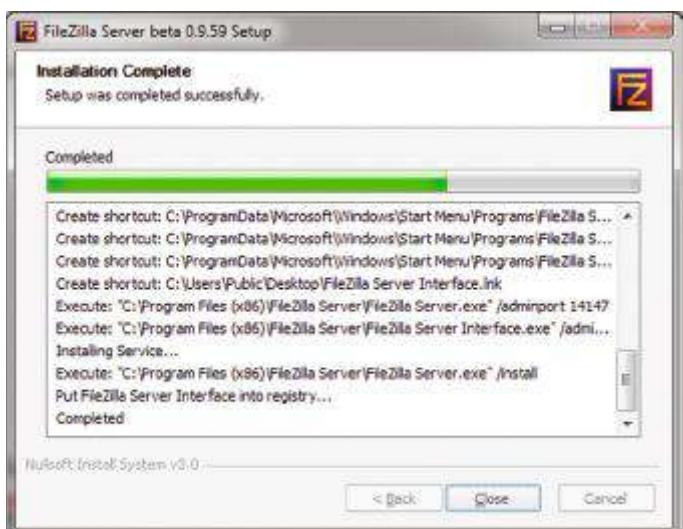
- 5 Viene indicata la porta TCP di default sulla quale connetterci al server in qualità di amministratori del server stesso (anche in questo caso lasciamo lo stesso valore proposto).



- 6 Confermiamo con **[Install]** se vogliamo che il programma sia utilizzabile per tutti gli utenti.



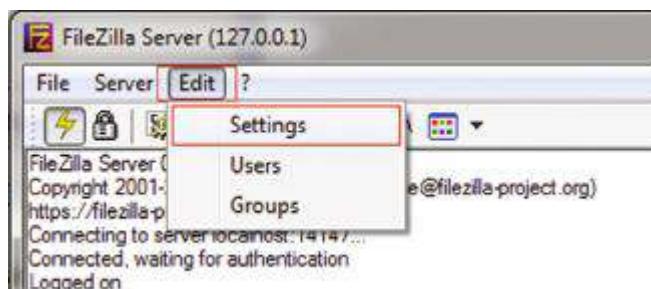
- 7 Inizia la copia dei file e l'installazione vera e propria:



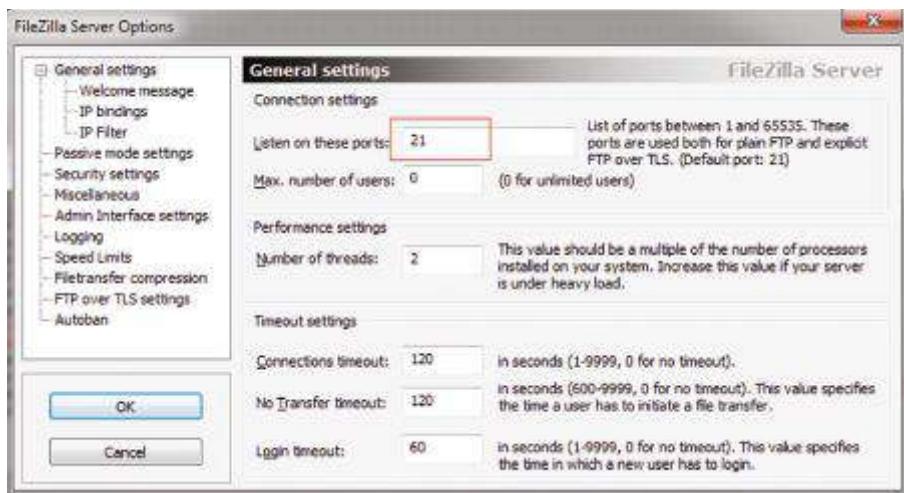
- 8 Al termine viene avviato il server **FTP**, appare una finestra che indica come connetterci a esso in qualità di amministratore, assegniamo a esso l'indirizzo di **loopback** (127.0.0.1) per effettuare le prove in **locale**:



- 9 A questo punto dobbiamo effettuare il settaggio del server. Per fare questo facciamo click su **Edit / Settings**:



- 10 Per prima cosa andiamo alla voce **General settings** e verifichiamo il port TCP sul quale il server è in ascolto (21):



A volte, soprattutto quando ci sono molte connessioni remote, può essere utile aumentare i tempi di Timeout in quanto i tempi indicati di default sono molto restrittivi.

- 11 A questo punto passiamo a definire l'indirizzo del server (**IP bindings**): potremmo anche lasciare l'asterisco che indica qualunque indirizzo, tuttavia, per maggiore sicurezza, assegniamo al server il solo indirizzo di **loopback** (ricordiamo che effettueremo le prove di connessione in locale):



- 12 Dobbiamo ora definire gli **utenti** che hanno accesso al server. Per fare questo facciamo click sul pulsante indicato a lato:



- 13 Mediante il pulsante **Add** passiamo ad aggiungere un nuovo utente:

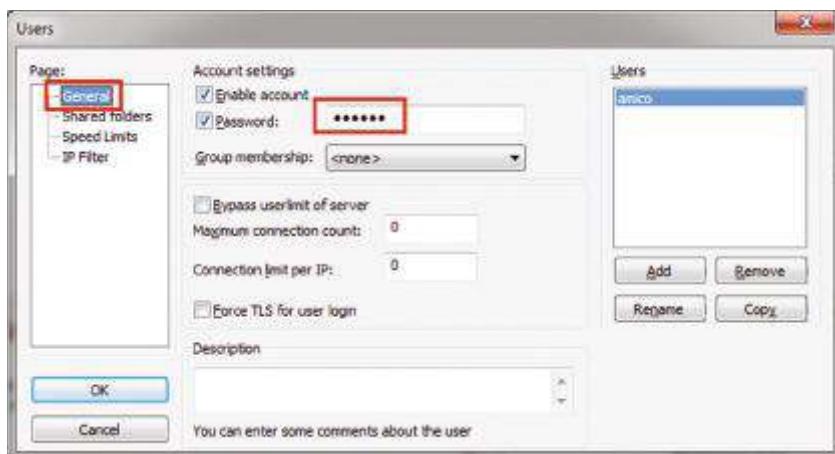


- 14 Inseriamo il nome dell'utente, in questo caso lo chiameremo **Amico**:

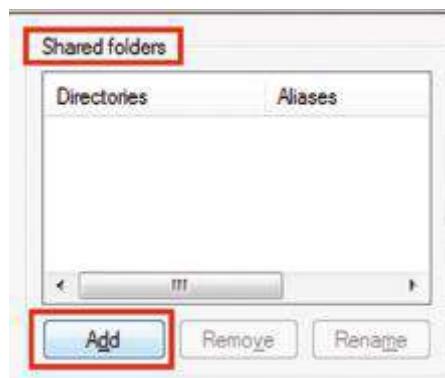


- 15 Aggiungiamo i parametri a esso associati: prima di tutti dobbiamo attivare **Enable account**, quindi digitare la **password**. Confermare con **OK** alla fine.

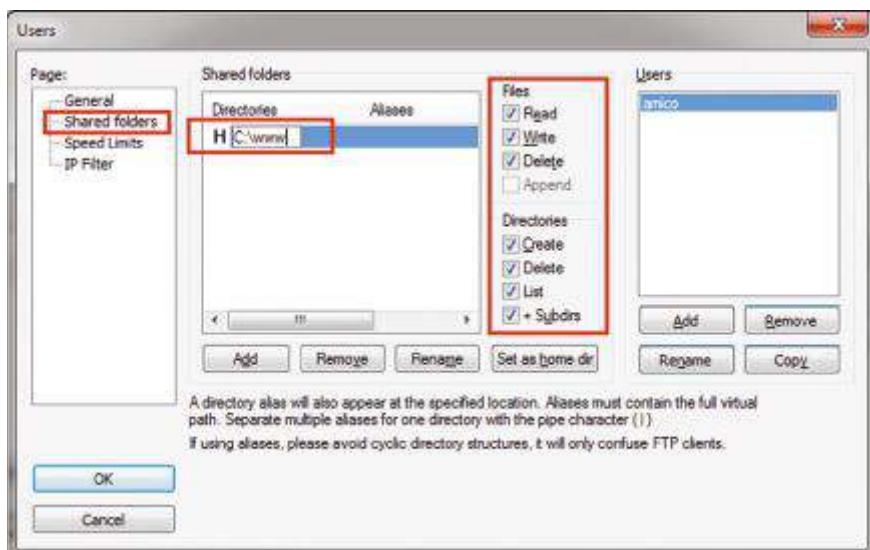
Queste operazioni devono essere ripetute per tutti gli utenti che vogliamo aggiungere al nostro server **FTP**.



- 16 Procediamo con la definizione delle cartelle sulle quali potranno caricare e scarica file gli utenti: nella sezione chiamata **Shared folders** facciamo click su **Add** per creare una nuova cartella per gli utenti.



- 17 Viene visualizzata una finestra nella quale dobbiamo selezionare la directory che diverrà la cartella condivisa: è chiaro che la cartella deve essere stata creata prima di effettuare questa operazione e noi abbiamo creato per questo scopo una apposita directory con percorso **e:\www** che selezioniamo.



Come possiamo notare, è stata associata la cartella **www** all'utente **amico**, con permessi assai privilegiati (**lettura, scrittura, cancellazione**) oltre a permessi specifici per le cartelle (**creazione, cancellazione, elenco, creazione di sottodirectory**).

- 18 A questo punto confermiamo con **OK**: il server è stato impostato per poter correttamente rispondere alle richieste che proverranno dal nostro script.  
Puoi anche testare il server utilizzando il software **FileZilla Client**, scaricabile sempre allo stesso indirizzo.

Procediamo con la scrittura dello **script** che dovrà effettuare la connessione al **server** per consentire di effettuare l'**upload** dei file: innanzitutto bisogna conoscere i dati necessari alla connessione, cioè l'indirizzo del server **FTP**, il numero di porta **TCP**, il nome dell'utente autorizzato e relativa password.

Definiamo inoltre due variabili per memorizzare il nome del file da caricare e il nome che lo stesso file acquisirà una volta inviato in remoto.

La connessione al server **FTP** avviene attraverso una funzione nativa apposita denominata **ftp\_connect()** (riga 22), che richiede come parametri il nome dell'host a cui si desidera connettersi e la porta

TCP che il **server** utilizza per le chiamate (generalmente si tratta della porta 21); **ftp\_login()** (riga 24) è invece la funzione che ci permette di autenticarci con il server **FTP**: essa accetta come parametri i dati di connessione definiti all'inizio del nostro listato.

Una volta connessi e autenticati, la funzione **ftp\_put()** (riga 28) si occuperà dell'invio del file a destinazione; da notare l'utilizzo del parametro **FTP\_BINARY** che specifica il tipo di file da caricare, in questo caso il tipo può essere qualunque. Una volta impostati i permessi desiderati, sarà possibile chiudere la connessione al server **FTP** con l'apposita funzione **ftp\_close()** (riga 32).

La pagina sfrutta il metodo del **postback** che consente di racchiudere sia il form da compilare che lo script di upload vero e proprio sempre nella stesso file **PHP**. Possiamo così suddividere la pagina in due parti, la prima contiene il codice **PHP** che consente l'upload del file secondo il protocollo **FTP**, mentre la seconda mostra a video il form che dovremo compilare per autenticarci e selezionare il file da caricare.

Il nome completo del file e del percorso in cui è memorizzato sul client è contenuto nell'array associativo **\$\_FILES['file']['tmp\_name']**, mentre il solo nome senza percorso completo è contenuto nell'array associativo **\$\_FILES['file']['name']**.

Di seguito è riportato il codice completo della nostra applicazione contenuta nel file **connessioneFTP.php**:

```

<?php
// tecnica Postback per verificare se è primo accesso
if (isset($_POST['send_file'])){
    // se è secondo accesso vengono letti i campi ricevuti dal Form
    $ftp_server = $_POST['ftp_server'];
    $porta = $_POST['port'];
    $username = $_POST['username'];
    $password = $_POST['password'];
    // verifica se i campi sono stati tutti compilati e validazione server
    if ($ftp_server != 'ftp://server' && $ftp_server != ''){
        // Validazione nome utente
        if ($username != 'username' || $username == ''){
            // validazione password
            if ($password != 'password' && $password != ''){
                // validazione nome del file da caricare
                if (is_uploaded_file($_FILES['file']['tmp_name'])){
                    // se tutto ok lettura nome del file con percorso completo
                    $file = $_FILES['file']['tmp_name'];
                    // lettura nome del file senza percorso
                    $new_file = $_FILES['file']['name'];
                    // connessione via FTP
                    $conn = ftp_connect($ftp_server, $porta) or die ('Impossibile connettersi al server.');
                    // login sul server
                    ftp_login($conn, $username, $password) or die('Username o password errati.');
                    // modalità passiva
                    ftp_pasv($conn, true);
                    // upload del file
                    $invia = ftp_put($conn, $new_file, $file, FTP_BINARY);
                    // esito dell'upload
                    echo ($invia) ? 'Upload fallito' : 'Upload completato<br>';
                    // chiusura della connessione
                    ftp_close($conn);
                }else{
                    echo "<font color=red><b>Inserire file</b></font><br>";
                }
            }else{
                echo "<font color=red><b>Inserire password</b></font><br>";
            }
        }else{
            echo "<font color=red><b>Inserire username</b></font><br>";
        }
    }else{
        echo "<font color=red><b>Inserire server ftp</b></font><br>";
    }
}
?>

```

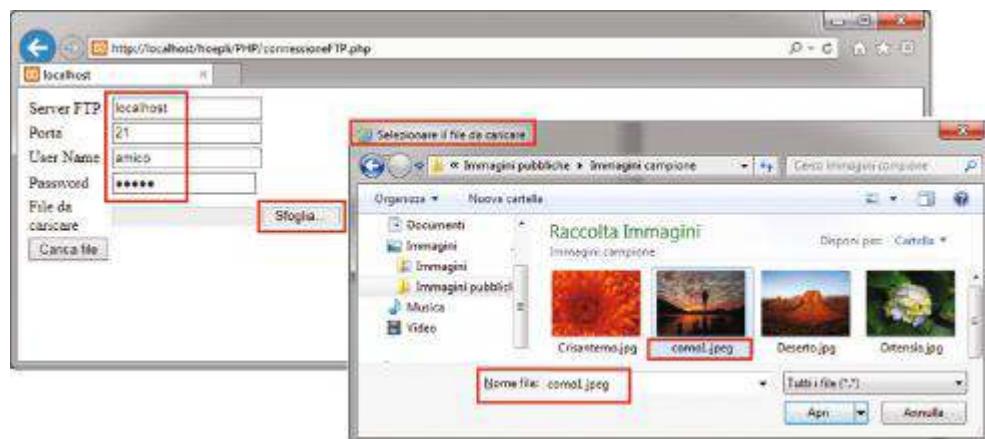
Conclude lo script la parte di **HTML** che visualizza il form che deve essere completato al primo caricamento della pagina:

```

47: <FORM ENCTYPE="multipart/form-data" NAME="modulo_ftp" ACTION="php echo $_SERVER['PHP_SELF'];??" METHOD="POST">
48: <TABLE WIDTH=30%><TR><TD>Server FTP</TD>
49: <INPUT TYPE="text" NAME="ftp_server" VALUE="ftp server"/>
50: </TR><TD>Porta</TD>
51: <INPUT TYPE="text" NAME="port" VALUE="21"/>
52: </TR><TD>User Name</TD>
53: <INPUT TYPE="text" NAME="username" VALUE="username"/>
54: </TR><TD>Password</TD>
55: <INPUT TYPE="password" NAME="password" VALUE="password"/>
56: </TR><TD>File da caricare</TD>
57: <INPUT TYPE="file" NAME="file"/>
58: </TR><TD>
59: <INPUT TYPE="submit" NAME="send_file" VALUE="Carica file"/>
60: </TABLE>
61: </FORM>
```

Come possiamo notare stiamo effettuando un test dello script in maniera locale in quanto il nome del server è **localhost**; la porta **TCP** da utilizzare è la **21** mentre il nome dell'utente è **amico** e la password è quella scelta per lo stesso utente, ovviamente.

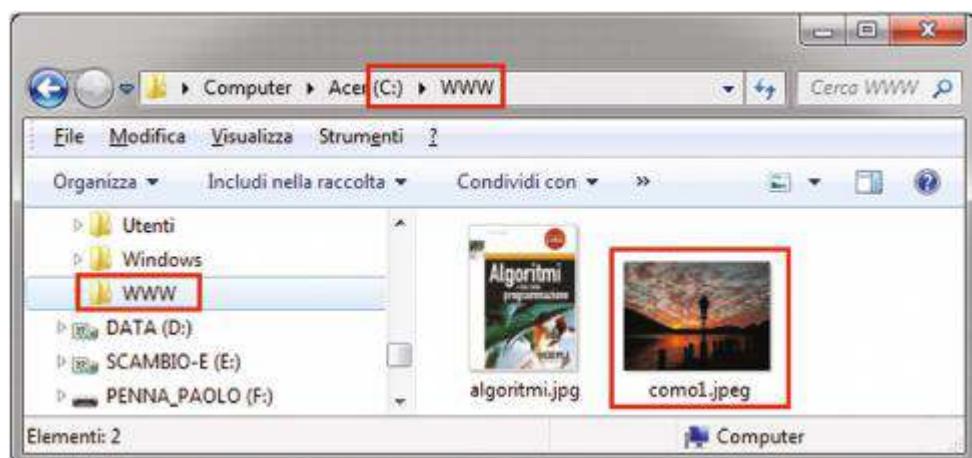
Cliccando sul pulsante **Sfoglia...** (rappresentato dal tag **HTML** di riga 57 `<INPUT TYPE="file">`) appare la finestra in cui selezionare il file da caricare. In questo caso possiamo notare che viene selezionato un file presente in una cartella qualsiasi del computer client:



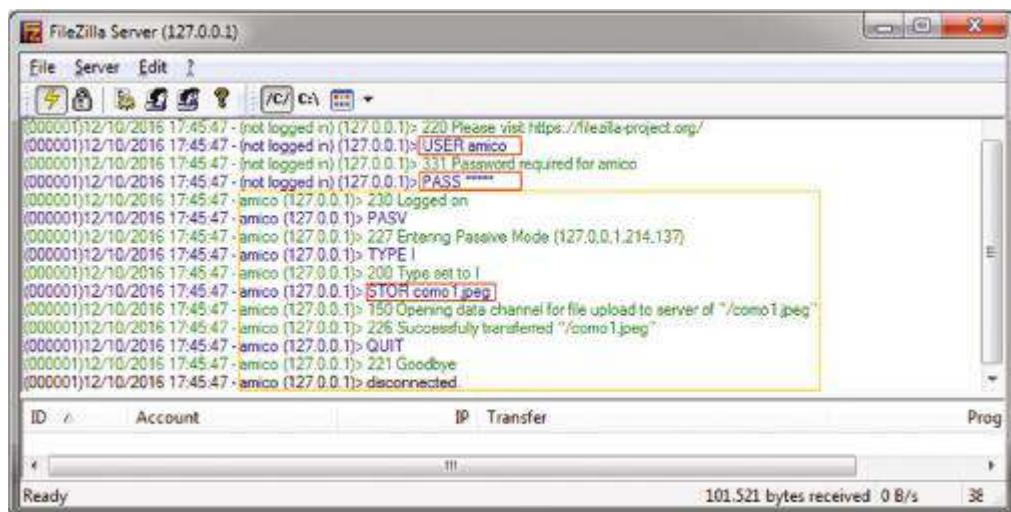
Una volta selezionato il file, facendo click su **[Carica file]** viene richiamata la stessa pagina che adesso esegue il codice necessario per effettuare l'upload vero e proprio del file. Al termine appare il messaggio **"Upload completato"**.



Verifichiamo se la cartella (`c:\www`) dell'utente chiamato **amico** contiene il file selezionato:



Al termine delle operazioni puoi verificare il log del server **FTP**, come puoi notare sono state indicate tutte le operazioni che lo script ha effettuato da remoto:



**Prova adesso!**

- Utilizzare un server FTP
- Effettuare una connessione da PHP a un server FTP

Prova a modificare il codice dell'esempio proposto in modo da poter caricare più di un file alla volta e verificarne il funzionamento installando il server FTP su di un host e facendo eseguire lo script da host differenti all'interno della stessa rete LAN.

# ESERCITAZIONI DI LABORATORIO 4

## INVIARE UNA MAIL CON PHP CONNESSO A MYSQL



Questa lezione richiede l'utilizzo di un **provider** con il servizio di **mail server** per effettuare l'invio della posta: in alternativa, se si volesse utilizzare questa possibilità in ambito locale, è necessario scaricare e installare un **mail server** da integrare a **xampp**, come ad esempio **Mercury**.

### ■ La funzione mail()

In **PHP** esiste una particolare funzione chiamata **mail()** che consente l'invio di un messaggio di posta elettronica; questa funzione possiede tre parametri, secondo la seguente sintassi:

```
mail($destinatario, $oggetto, $testo);
```

La funzione **mail()** restituisce un valore **false** nel caso l'operazione di invio messaggio non sia andata a buon fine, così possiamo verificare l'esito attraverso una semplice condizione:

```
mail($destinatario, $oggetto, $testo) or die ("spedizione non avvenuta");
```

Esiste anche un quarto parametro opzionale che consente l'aggiunta di una **header** (intestazione) supplementare, come possiamo vedere nel codice di esempio seguente:

```
$destinatario = 'amico@mail.it';
$oggetto      = 'Testing di mail()';
$testo        = 'Ciao come stai?\nprova di invio mail!\nSaluti!';
$mittente     = 'io@mail.it';
mail($destinatario, $oggetto, $testo, "$from\r\nX-Priority: 1 (Highest)");
```

È importante sottolineare che tra un **header** aggiuntivo e un altro bisogna inserire i caratteri **\r\n**, come indicato nell'**RFC** dedicato al protocollo **SMTP** e **POP**.

Vediamo un codice di esempio che mostra come utilizzare altre intestazioni:

► **Reply-To:** utilizzato dai client di posta come Eudora per le funzioni di risposta;

- **X-Mailer:** relativo al tipo di client utilizzato per l'invio dell'email.

```
mail($destinatario, $oggetto, $testo, "From: io@mail.it\r\n"
 . "Reply-To: io@mail.it\r\n"
 . "X-Mailer: PHP/" . phpversion());
```



◀ **Header** Si tratta dell'intestazione di un messaggio di posta elettronica; è formato da campi che descrivono i parametri del messaggio, come il mittente, il destinatario, la data ecc.  
Ogni intestazione possiede il formato:

Nome: Valore
--------------

Una intestazione comprende almeno le tre intestazioni seguenti:

- **From:** indirizzo elettronico del mittente;
- **To:** indirizzo elettronico del destinatario;
- **Date:** data di creazione del messaggio.

Può inoltre contenere le intestazioni facoltative seguenti:

- **Received:** informazioni diverse sui server intermedi e la data di elaborazione del messaggio associato;
- **Reply-To:** un indirizzo per la risposta;
- **Subject:** l'oggetto del messaggio;
- **Message-ID:** un identificativo unico del messaggio.

Tuttavia il destinatario e l'oggetto vengono inseriti a parte come argomenti della funzione **mail()**. ►

Altre intestazioni aggiuntive che possono tornare utili sono **Cc:** e **Bcc:**, necessarie per inviare copie conoscenza ad altri indirizzi.

Il quarto argomento della funzione **mail()** consente inoltre di passare dei parametri a riga di comando al programma server utilizzato per l'invio della posta elettronica: questi parametri sono strettamente legati al tipo di Mail server utilizzato (ad esempio **SendMail**, utilizzato da quasi tutti i server **Linux**).

Il codice che segue mostra come inviare email in formato **HTML**:

```
$testo = "<html><head>
<title>Messaggio ai lettori</title>
</head><body>
Caro lettore, cosa ne pensi di questo libro? Saluti, Mario Rossi.
</body></html>";
$dest="indirizzo_1@email.com";
$oggetto ="Feedback";
// impostare intestazione Content_type
$header = "MIME-Version: 1.0\r\n";
$header .= "Content-type: text/html; charset=iso-8859-1\r\n";
$header .= "From: MarioRossi <mariorossi@hoepli.it>\r\n";
$header .= "Cc: indirizzo_2@email.com \r\n";
$header .= "Bcc: indirizzo_3@email.com \r\n";
mail($dest, $oggetto, $testo, $header);
```

La funzione **mail()** restituisce un valore **false** nel caso l'operazione di invio messaggio non sia andata a buon fine, così possiamo verificare l'esito attraverso una semplice condizione:

La funzione **mail()** in **Linux** è sostituita dalla funzione **sendmail()**.



## Zoom su...

### IL FILE PHP.INI E LE MAIL

La funzione `mail()` innanzi tutto analizza la direttiva **SMTP** presente nel file `php.ini` contenente il programma **server** che si desidera utilizzare per l'invio delle email.

La sezione **[mail function]** del file `php.ini`, riportata di seguito, determina il server **SMTP** in uso sul sistema:

```
[mail function]
; For Win32 only.
SMTP = localhost
; For Win32 only.
sendmail_from = me@localhost.com
; For Unix only. You may supply arguments as well (default: 'sendmail -t -i').
;sendmail_path =
```

Come possiamo notare la prima direttiva è valida soltanto su sistemi **Windows** a 32 bit, e dovrebbe essere commentata se si lavora su sistemi **Unix**. Essa specifica l'indirizzo del server mail che si vuole utilizzare per inviare email con la funzione `mail()`.

La seconda direttiva è anch'essa utilizzata su sistemi **Windows** a 32 bit, specifica il campo **from**. La terza direttiva, qui commentata perché il file `php.ini` di esempio è quello per un ambiente **Windows**, specifica il percorso fisico del programma di invio email per **Linux**.

## ■ Una mailing list proveniente da MySql

In questo esempio vogliamo inviare una email a un gruppo di indirizzi, simulando l'invio a una mailing list.

Lo script utilizza una tabella di **MySQL** chiamata **Indirizzi** che contiene gli indirizzi a cui inviare (in simulazione) lo stesso messaggio:

The screenshot shows the MySQL Workbench interface. On the left, the database tree shows 'Unnamed' (16,0 KiB), 'provephp' (16,0 KiB), and 'indirizzi' (16,0 KiB). On the right, the 'indirizzi' table is displayed with the following data:

ID_CLIENTE	Email	cognome	nome
1	indirizzo_1@email.com	Rossi	Mario
2	indirizzo_2@email.com	Verdi	Mario
3	indirizzo_3@email.com	Rossi	Luigi
4	indirizzo_4@email.com	Verdi	Luigi
5	indirizzo_5@email.com	Galli	Mario
6	indirizzo_6@email.com	Galli	Pino

La struttura della tabella è molto semplice e contiene quattro campi: il primo è la chiave primaria di tipo **AUTO\_INCREMENT**, il secondo è l'indirizzo email dell'utente a cui inviare la newsletter, i successivi il cognome e nome. Lo script di amministrazione si limiterà a presentare un FORM **HTML** in cui inserire il testo e l'oggetto della mail. Al click sul pulsante di invio, lo script richiama se stesso in **postback** ed esegue l'invio delle email.

Analizziamo il codice: innanzi tutto viene verificato il contenuto dei campi POST inviati (**riga 3**) per decidere se si tratta del primo accesso oppure se l'utente ha già immesso i dati nei campi. In

questo caso passiamo a effettuare la connessione con il database **MySQL** per estrarre i nomi delle mail (**righe 11-13**).

```

1 <?php
2 // tecnica postback per rientrare nella stessa pagina
3 if (isset($_POST['messaggio']) && isset($_POST['oggetto'])){ // seconda volta
4 //accesso dopo inserimento dati - connessione a MySQL
5 $db = new mysqli("localhost", "root", "", "provephp");
6 // prova a connettersi al database
7 if (mysqli_connect_errno()){
8 printf("Connessione non effettuata: %s\n", mysqli_connect_error());
9 exit();
10 }
11 $query = "SELECT * FROM Indirizzi"; // predisposizione della query
12 if ($res = $db->query($query)) { // esecuzione query
13 printf(" - la select ha individuato %d righe.\n", $res->num_rows);
14 }

```

Possiamo ora iniziare a caricare nella variabile **\$intestazione** la prima riga d'intestazione che prevede i dati del mittente (**riga 17**); vengono successivamente salvati i dati dei campi **POST** relativi all'oggetto e al testo da inviare (**righe 19-20**).

```

16 // scrittura intestazione
17 $intestazioni = "From:mittente@hoepli.com \r\nReply To:mittente@hoepli.com";
18 $i = 0; // contatore di mail inviate
19 $oggetto = $_POST['oggetto']; // lettura oggetto e testo messaggio da POST
20 $messaggio = $_POST['messaggio'];
21 foreach( $res as $row ) { // ciclo che estrae i record restituiti da SQL
22 $dest = $row['Email']; // $dest contiene l'email del destinatario
23 $i++;
24 $x = mail($dest, $oggetto, $messaggio, $intestazioni); // invio della mail
25 }
26 echo "<HR>Sono state inviate $i email<HR>";
27 echo "<a href='".$ _SERVER['PHP_SELF']."'>Torna allo script</A>";
28 $res->close();
29 $db->close();

```

Il ciclo **foreach** (**righe 21-25**) legge per ogni riga **\$row** della tabella il campo [**Email**] invia la mail (**riga 24**).

Al termine di questa sezione avviene la comunicazione di quante mail inviate mediante la stampa a video del contatore **\$i** (**riga 26**) e viene proposta la possibilità di tornare alla pagina stessa.

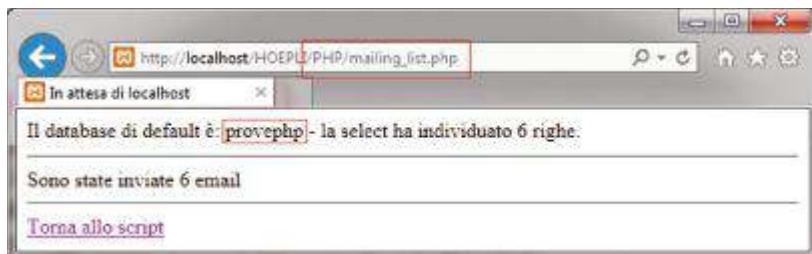
La seconda parte dello script mostra il form necessario per l'inserimento dei dati (**righe 33-38**).

```

31 else{ // mostro il form
32 echo "
33 <FORM name='form' method='post' action='$_SERVER['PHP_SELF']'>
34 <B>OGGETTO</B><INPUT TYPE='text' NAME='oggetto'><BR>
35 <TEXTAREA NAME='messaggio' cols='70' rows='5'></TEXTAREA><BR>
36 <INPUT TYPE='submit' VALUE='Invia Email ai clienti'>
37 </FORM></BODY></HTML>";
38 }
39 ?>

```

Mandando in esecuzione lo script otteniamo il risultato aspettato, cioè che vengono inviate 6 email, una per ogni record presente nella nostra tabella del database:



## Prova adesso!



**APRI IL FILE** mailing\_list.php

- Utilizzare la funzione mail()
- Connessione ai database MySQL

- 1 Modifica il codice dello script in modo tale che venga mostrato a video l'elenco dei clienti e l'utente possa selezionare quelli a cui spedire la mail. Per fare questo colloca dei pulsanti checkbox accanto ai nomi letti dal database.
- 2 Modifica il codice dello script in modo da poter inviare email con mittente diverso da quello indicato nello script.
- 3 Infine scrivi un programma che permetta a un utente di accedere alla sezione riservata di un sito in caso di smarrimento password: il sistema genera casualmente una nuova password di 6 caratteri scelti ad esempio da un insieme limitato, in modo da non avere ambiguità (O oppure 0, I oppure 1 ecc.):

```
function passwordCasuale($lunghezza){
    $setCaratteri ="ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz23456789";
```

e la invia all'indirizzo del cliente. Confronta il tuo codice con quello presente nel file [mailRecuperoPassword.php](#).

# ESERCITAZIONI DI LABORATORIO 5

## CREARE FILE PDF CON PHP



I file utili alla corretta esecuzione di questa lezione sono reperibili nella cartella `htdocs/hoepli/php/pdf`, dove sono reperibili anche i file di esempio.

### ■ Il formato pdf

Il **Portable Document Format**, comunemente abbreviato **PDF**, è un formato di file basato su un linguaggio di descrizione di pagina sviluppato da **Adobe Systems** nel 1993 per rappresentare documenti in modo indipendente dall'hardware e dal software utilizzati per generarli o per visualizzarli. Il vantaggio di un documento in formato portatile sta principalmente nel fatto che può essere letto da qualunque versione di lettore **pdf**, anche se meno recente.

Un file in formato **Word**, ad esempio, non è sempre detto che possa essere letto, soprattutto se chi lo deve leggere possiede una versione anteriore rispetto a quella usata per crearlo. Oppure spesso ci sono problemi nella decodifica e nell'interpretazione di dati quando il mittente non utilizza quelli definiti dalle norme di codifica internazionali: ad esempio in alcuni messaggi di posta elettronica, in cui le “è”, gli accenti e altri simboli vengono trasformati in “?” o altri simboli strani, dovuto al fatto che entrambi i programma di posta elettronica non sono settati in **Unicode** (ISO10146).

Col formato **PDF** tutto ciò è stato risolto: se si trasforma un qualsiasi testo o immagine da un programma di scrittura o grafico in **PDF**, chi riceverà questo documento lo leggerà perfettamente come lo vedete voi sul vostro monitor.

### ■ La libreria FPDF

**FPDF** è una classe **PHP** che permette di generare file **PDF** con codice **PHP** puro. Esistono anche altre librerie, tuttavia questa è gratuita e assai diffusa: la lettera **F** che si antepone alla parola **PDF** indica l'origine **freeware** della libreria dalla quale proviene; inoltre questa classe può essere modificata in base alle proprie esigenze. Le sue caratteristiche principali sono le seguenti:

- ▶ gestione dell'intestazione e del piè di pagina;
- ▶ scelta del formato e delle caratteristiche della pagina (misura e margini);
- ▶ interruzione di pagina automatica;
- ▶ interruzione di riga automatica e giustificazione del testo;
- ▶ supporto immagini (JPEG, PNG e GIF);
- ▶ colori;
- ▶ links;
- ▶ compressione della pagina.

**AREA** *digitale*



Vantaggi offerti dal formato PDF

La libreria **fpdf** è scaricabile gratuitamente dal web. È stata tuttavia inclusa nella sottocartella **pdf** degli esempi proposti. Il file che viene scaricato da Internet è in formato compresso, è pertanto indispensabile esploderla per poterla utilizzare.

Riportiamo nella tabella seguente principali metodi e proprietà.

NOME	DESCRIZIONE
AddPage()	Aggiunge una pagina al file.
Cell(w,h,txt="",border,ln,align="",fill,link "")	Inserisce una cella di testo con le caratteristiche specificate nei parametri.
Footer()	Metodo per gestire il piè di pagina; viene richiamato ogni volta che c'è un "AddPage()".
Header	Metodo per gestire l'intestazione; viene richiamato ogni volta che c'è un "AddPage()".
Image (name,x,y,w=0,h=0,type "",link "")	Inserisce un'immagine con le caratteristiche specificate nei parametri.
Ln (h="")	Va a capo; si può specificare l'altezza "h".
Output (name="",dest "")	Invia il file alla destinazione definita; "name" è il nome del file e "dest" è la cartella di destinazione o il browser.
SetFillColor (r:int)	Definisce il colore di background (formato RGB).
SetFont(string family [, string style [, float size]])	Imposta il font utilizzato per stampare stringhe di caratteri. È obbligatorio chiamare questo metodo almeno una volta prima di stampare del testo oppure il documento risultante potrebbe non essere valido. Il font può essere sia uno standard oppure uno aggiunto tramite il metodo AddFont(). Il metodo può essere chiamato prima della creazione della prima pagina per mantenere il font di pagina in pagina. Se vogliamo cambiare soltanto la dimensione del font possiamo usare il metodo SetFontSize().
PageNo()	Restituisce il numero della pagina corrente.
Write(float h, string txt [, mixed link])	Questo metodo stampa del testo dalla posizione corrente. Quando viene raggiunto il margine destro (o viene incontrato il carattere \n) viene inserita una interruzione di linea e il testo continua dal margine sinistro. All'uscita del metodo, la posizione corrente è lasciata alla fine del testo.
Text(float x, float y, string txt)	Stampa una stringa di caratteri. L'origine è alla sinistra del primo carattere, sulla baseline. Questo metodo permette di posizionare una stringa con precisione sulla pagina, ma di norma è più semplice utilizzare Cell(), MultiCell() o Write() che sono i metodi standard per stampare del testo

Il codice che segue mostra come creare un semplice file **pdf** usando la libreria **fpdf**.

```
ciaoMondoPDF.php
1 <?php
2   require('fpdf.php');           // include libreria fpdf
3   $pdf = new FPDF();             // creo un oggetto
4   $pdf->AddPage();              // aggiungo una pagina
5   $pdf->SetFont('Arial', 'B', 18); // setto il tipo di font
6   $pdf->Cell(50, 10, 'Ciao Mondo!'); // creo una cella con testo
7   $pdf->Output();               // visualizzo la pagina
8 ?>
```

Prima di tutto richiamiamo il file **fpdf.php** che contiene la classe e includerlo con l'istruzione di riga 2.

Quindi creiamo l'istanza della classe **FPDF** (dando all'oggetto come nome proprio **\$pdf**) e a questo documento aggiungiamo una pagina col metodo **AddPage()**.

Possiamo ora definire il font: in questo caso carattere Arial, grassetto (Bold), corpo 18 punti.

Il metodo **Cell()** crea una cella di una tabella lunga 50 e alta 12 in cui scrive "Ciao Mondo!"

Infine il metodo **Output()** invia effettivamente il file PDF al client.

L'esecuzione crea un file PDF sul computer del client con le impostazioni di formattazione indicate sopra:



L'esempio che segue mostra quattro possibili metodi diversi per generare testo:

```
<?php
    // la cartella font si trova nella stessa directory dello script
    define('FPDF_FONTPATH','./font/');
    require('fpdf.php');
    $pdf = new FPDF();           // istanzio un oggetto
    $pdf->AddPage();           // apertura pagina
    $pdf->SetTextColor(0);       // Impostazione colore
    $pdf->SetFont('Times','', 18); // impostazione font
    $pdf->SetX(25);             // posizione dal margine sinistro in pixel
    // quattro funzioni diverse: Text(), Cell(), Write(), Multicell()
    $pdf->Text(5, 10,'Testo con Text()');
    $pdf->SetY(10);
    $pdf->Cell(0, 5, 'Testo con Cell()');
    $pdf->Write(5, 'Testo con Write()');
    $pdf->MultiCell(0, 5,"\n". 'Testo con '\n'. 'MultiCell()', 0,'center');
    $pdf->Output();
?>
```

L'esecuzione dello script ottiene il seguente risultato:



## ■ Leggere da un database per creare un documento PDF

In questo esempio leggeremo i dati da una tabella di un database in formato **MySQL** per collocarli in un documento **PDF** all'interno di una tabella formattata opportunamente. Dopo aver incluso la libreria **fpdf** effettuiamo la connessione al nostro database **provephp** (righe 4-9).

Viene quindi definita la query SQL di selezione dei record della tabella **Carrello** (riga 10) ed eseguita con l'istruzione presente nella riga successiva.

```

1 <?php
2 require('fpdf.php');
3 // connessione al database inserendo come parametri
4 $db = new mysqli("localhost", "root", "", "provephp");
5 // tentativo di connessione al database
6 if (mysqli_connect_errno()) {
7     printf("Connessione non effettuata: %s\n", mysqli_connect_error());
8     exit();
9 }
10 $query = "SELECT * FROM Carrello"; // predisposizione della query
11 if ($res = $db->query($query)) { // esecuzione query
12     printf(" - la select ha individuato %d righe.\n", $res->num_rows);
13 }
14 ob_end_clean(); // svuota il buffer di output

```

Inizializziamo un insieme di variabili che utilizzeremo nello script per la formattazione delle tabella per contenere le coordinate iniziali, la larghezza di ogni colonna:

```

16 // parametri configurazione pagina
17 $nrpagina = 0; // numero di pagina
18 $max = 25; // massimo numero di righe per pagina
19 $y = 0; // inizio pagina
20 $i = 0; // contatore righe
21 $y_inizio = 15; // definisco la posizione iniziale y della prima riga
22 $h = 6; // definisco l'altezza delle righe in pixel
23 $l1 = 10; // larghezza colonna 1
24 $l2 = 40; // larghezza colonna 2
25 $l3 = 60; // larghezza colonna 3
26 $l4 = 20; // larghezza colonna 4
27 $l5 = 20; // larghezza colonna 5

```

In particolare utilizzeremo:

- ▶ **\$y**: indica la distanza dal bordo superiore del foglio (come le **ordinate** del piano cartesiano);
- ▶ **\$y\_inizio**: indica il valore iniziale della **y**, per l'inizio di ogni nuova pagina (inizializzata a 15);
- ▶ **\$h**: indica l'altezza (in pixel) delle righe della tabella e pertanto indica di quanto incrementare la **y** per passare alla riga successiva (inizializzata a 6).

Nella **riga 30** istanziamo un oggetto **\$pdf** della classe **FPDF**, quindi viene definito il font dei caratteri usati nella tabella (**riga 31**) e col metodo **SetAutoPageBreak(false)** impostiamo la modalità “manuale” per la gestione del “salto pagina”, in modo che la gestione venga lasciata al programmatore. Dopo aver aggiunto la pagina (**riga 34**) e il colore di sfondo (**riga 36**), con i metodi **SetY()** e **SetX()** (**righe 38-39**) definiamo le coordinate iniziali (alto-sinistra) in cui verranno collocate successivamente le celle della tabella con il metodo **Cell()**.

```

30 $pdf = new FPDF(); // 
31 $pdf->SetFont('Arial','B',12); // settaggio del font
32 $pdf->SetAutoPageBreak(false); // disabilito fine pagina automatico
33 // -----
34 $pdf->AddPage(); // stampa dei titoli delle colonne
35 // colori riempimento titolo
36 $pdf->SetFillColor(232,232,232); // colore riempimento titolo
37 $pdf->Text(25, 10,'Articoli presenti nel tuo carrello:');
38 $pdf->SetY($y_inizio); // posizione dall'alto in pixel
39 $pdf->SetX(25); // posizione dal margine sinistro in pixel

```

Il codice seguente disegna le celle dell'intestazione della tabella (**righe 42-46**) e predisponde il colore bianco come sfondo della pagina sulla quale verranno inseriti i prodotti.

Dopo aver inizializzato la coordinata y ([riga 49](#)) viene incrementata la posizione di inizio riga ([riga 50](#)).

```

40 // inserisco titoli nelle celle della prima riga
41 // parametri (larghezza,altezza,titolo,direzione,bordo,allineamento,stondo)
42 $pdf->Cell($l1,$h,'ID',1,0,'L',1);
43 $pdf->Cell($l2,$h,'Nome',1,0,'L',1);
44 $pdf->Cell($l3,$h,'Descrizione',1,0,'L',1);
45 $pdf->Cell($l4,$h,'Quantita',1,0,'C',1);
46 $pdf->Cell($l5,$h,'Prezzo',1,0,'C',1);
47
48 $pdf->SetFillColor(255,255,255); // colore riempimento bianco
49 $y = $y_inizio; // inizio pagina
50 $y = $y + $h; // ci spostiamo in basso di una riga
51 $i = 0; // contatore righe

```

Scorriamo l'array associativo `$res` ottenuto come risposta dalla query mediante l'istruzione `foreach()` che legge i dati dal `recordset` ([riga 56](#)) e li trasferisce riga per riga in una variabile temporanea `$row`.

Innanzitutto controlloiamo se siamo giunti a "fine pagina" ([riga 57](#)): in questo caso si deve intestare la nuova pagina ([riga 18](#): lasciamo la scrittura di questo segmento di codice come esercitazione).

Se la riga non è l'ultima vengono salvati i campi letti dal database ([righe 61-65](#)), vengono settate le coordinate di inizio della nuova riga ([righe 67-68](#)) e vengono stampati i campi prima letti ([righe 71-75](#)).

```

56 foreach( $res as $row ) { // ciclo per estrarre i record
57     if ($i == $max){ // controllo per il salto pagina
58         // $nrpagina = intestaPagina($nrpagina);
59     }
60     //salvataggio campi letti dal database
61     $id = $row['ID'];
62     $nome = $row['Nome'];
63     $descr = $row['Descrizione'];
64     $quanti = $row['Quantita'];
65     $prezzo = $row['Prezzo'];
66
67     $pdf->SetY($y); // settaggio riga a nuova riga
68     $pdf->SetX(25); // settaggio colonna
69
70     //Stampa riga con i campi letti dal database
71     $pdf->Cell($l1, $h, $id, 1,0,'L',1);
72     $pdf->Cell($l2, $h, $nome, 1,0,'L',1);
73     $pdf->Cell($l3, $h, $descr, 1,0,'L',1);
74     $pdf->Cell($l4, $h, $quanti, 1,0,'C',1);
75     $pdf->Cell($l5, $h, $prezzo, 1,0,'R',1);
76
77     $y = $y + $h; // vado a riga successiva
78     $i = $i + 1; // incremento contatore di righe
79 }

```

Al termine viene chiusa la connessione al database ([riga 90](#)) e inviata la pagina pdf al client ([riga 92](#)).

```

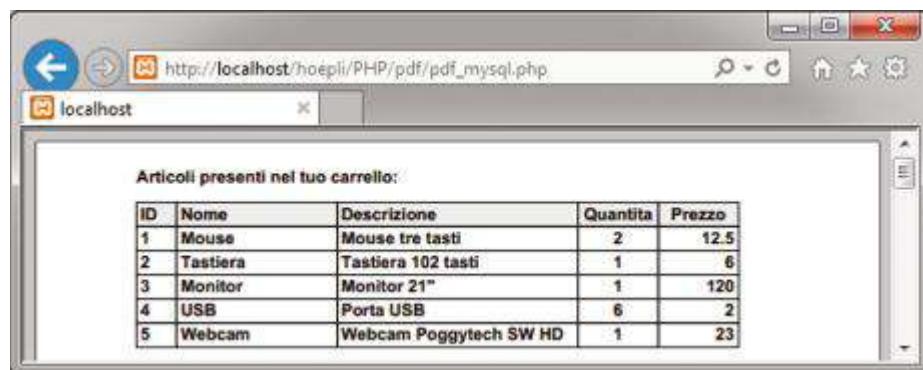
89 // chiudo connessione al database
90 $db->close();
91 // invio file-pdf al client
92 $pdf->Output();
93 ?>

```

Prima di mandare in esecuzione lo script verifichiamo il contenuto della tabella **carrello** del database **provephp** che, come possiamo vedere, contiene cinque articoli:

provephp.carrello: 5 righe totali (circa)					
	ID	Nome	Descrizione	Quantità	Prezzo
1	Mouse	Mouse tre tasti		2	12,5
2	Tastiera	Tastiera 102 tasti		1	6
3	Monitor	Monitor 21"		1	120
4	USB	Porta USB		6	2
5	Webcam	Webcam Poggytech SW HD		1	23

L'esecuzione dello script mostra i medesimi dati su file **pdf** salvabile sul computer del client:



## Prova adesso!

- Utilizzare la libreria fpdf
- Usare le conoscenze di PHP
- Usare la connessione a MySQL



[APRI IL FILE pdf\\_mysql.php](#)

- 1 Crea una tabella di nome **amici** con i seguenti campi (**ID, nome, cognome, indirizzo, telefono, email, foto**). Il campo **foto** è un link al file immagine (formato **.jpg, .gif** oppure **.png**).
- 2 Modifica il codice dello script in modo che venga mostrato in un file pdf il nome degli amici e la fotografia. Per fare questo utilizza il metodo **Image(imagine,x,y)**, dove **immagine** è una stringa che contiene l'URL del file dell'immagine, **x** è l'ascissa dell'angolo superiore-sinistro e **y** è l'ordinata dell'angolo superiore-sinistro.

Ad esempio:

```
$fpdf->Image('logo.jpg',10,75);
```

- 3 Aggiungi una funzione che gestisce l'intestazione della pagina ogni volta che si presenta la situazione di salto pagina, da richiamare ad esempio alla **riga 58** dello script prima descritto.

```

2 function intestaPagina($numero){
3     $numero = $numero + 1; // numero di pagina
4     // ... da scrivere
5     // ...
6     return $numero;
7 }
8 }
```

# ESERCITAZIONI DI LABORATORIO 7

## LE API DI GOOGLE: INTERAZIONE TRA JAVASCRIPT, AJAX E PHP

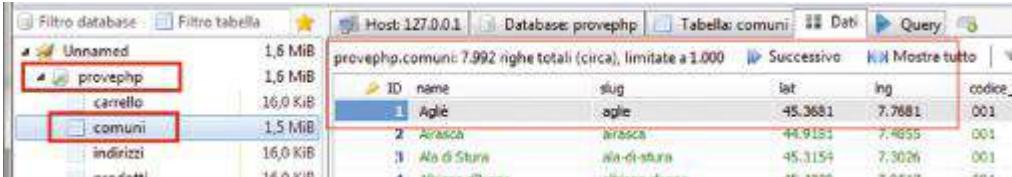
**AREA** *digitale*

 Lab. 6 Creare file di Excel e Word con PHP

### ■ Le API di Google: interazione tra JavaScript, AJAX e PHP

In questa esercitazione vedremo come far interagire le mappe di Google con PHP in modo da estrarre informazioni da un archivio per visualizzarle sulla cartina geografica.

In particolare realizziamo uno script che aiuta l'utente tramite la modalità **adsense** a ricercare una località all'interno della tabella **comuni** del database **provephp**.



ID	name	slug	lat	lon	codice_iz
1	Aglè	agle	45.3681	7.7681	001
2	Aracca	aracca	44.9181	7.4655	001
3	Alo di Stura	alo-di-stura	45.1151	7.3026	001
4	Albiano d'Ivrea	albiano-divrea	45.4339	7.9517	001

I valori restituiti dal database vengono collocati in un riquadro della pagina **HTML** e quindi vengono mostrati sulla mappa dell'Italia i **marker** indicanti le coordinate geografiche delle località mostrate.

Per fare questo la pagina utilizza i seguenti **linguaggi**:

- ▷ HTML, JavaScript, PHP, SQL

e le seguenti **librerie**:

- ▷ Ajax, API Google Maps

L'esempio è formato da due pagine, la prima **ajaxMysql.html** che ha il compito di:

- ▷ mostrare due **<DIV>**;
- ▷ inizializzare la mappa di **google**;
- ▷ leggere il nome del Comune da cercare con una casella di testo e trasferirlo alla pagina **elabora.php**;
- ▷ questa pagina riceve la stringa da cercare e effettua una interrogazione alla tabella **comuni** del database **provephp**;
- ▷ restituisce l'elenco di Comuni trovati all'interno di una stringa alla pagina **HTML** che mostra i marker sulla mappa nelle coordinate indicate dalla stringa ricevuta convertita in un array.

Iniziamo a vedere il codice della prima pagina **ajaxMysql.html**.

```

1 <HTML><HEAD>
2 <STYLE>
3 div.box {
4   background-color:azure;
5   width:350px;
6   height:400px;
7   overflow:auto;
8 }
9 div.mappa {
10   position:absolute;
11   left:370px;
12   top:116px;
13   width:550px;
14   height:400px;
15   background-color:yellow;
16 }
17 </STYLE>

```

Prima di tutto creiamo i fogli di stile necessari alla visualizzazione dei due `<DIV>` chiamati '`box`' e '`mappa`' che conterranno rispettivamente, la mappa e l'elenco dei Comuni e inseriamo la [Google Key](#).

```

18 <SCRIPT TYPE="text/javascript"
19 SRC="https://maps.googleapis.com/maps/api/
js?key=AIZaSyChIsh9mTgKrBBzRAdmRVFLBTDWG__5awK&sensor=true">
20 </SCRIPT>

```

La pagina [HTML](#) può essere divisa in 3 sezioni:

- 1 la sezione in cui vengono utilizzate le [API di Google Map](#);
- 2 la sezione in cui utilizziamo [AJAX](#) per collegarci con la pagina `elabora.php`;
- 3 la sezione di [HTML](#) che visualizza i risultati.

Nella prima parte vengono dichiarate alcune variabili globali: la matrice `posti[]`, che conterrà le posizioni dei marker, l'oggetto `map` di classe [Maps](#) e l'oggetto `marker` di classe [Marker](#).

```

25 // matrice globale di posti per memorizzare i luoghi con le relative coordinate
26 var posti = new Array();
27 var map;
28 var marker;

```

La funzione `init()` (riga 30) viene chiamata all'avvio della pagina e definisce le opzioni della mappa (righe 31-35), quindi dopo aver istanziato l'oggetto `map` (riga 37) personalizza i marker (righe 38-43).

```

29 //funzione eseguita all'apertura della pagina HTML
30 function init(){
31   var mapOptions = {                      // definizione delle opzioni della mappa
32     zoom: 7,
33     center: new google.maps.LatLng(42.1,12.4833),
34     mapTypeId: google.maps.MapTypeId.ROADMAP
35   }
36   // istanzio l'oggetti di classe Map
37   var map = new google.maps.Map(document.getElementById('mappa'), mapOptions);
38   var image = {                          // definizione icona del marker
39     url: 'mark.png',
40     size: new google.maps.Size(20, 32),
41     origin: new google.maps.Point(0, 0),
42     anchor: new google.maps.Point(0, 32)
43   };

```

Il ciclo successivo (righe 45-56) è assai importante perché è quello che pone i marker nella mappa secondo le coordinate presenti nell'array `posti` che viene riempito e, quindi, ricevuto dalla sezione di [AJAX](#). La matrice `posti` è formata da 3 colonne e da "n" righe contenenti il nome del Comune, la sua latitudine e longitudine. Ogni riga della matrice viene copiata in un array di comodo chiamato `posto[]` (riga 48).

```

44 // ciclo di scansione dell'array posti
45 for (var i = 0; i < posti.length; i++){
46     // ogni riga di posti[] è formata da 3 elementi: nome, latitudine e longitudine
47     // vengono assegnati all'array temporaneo posto[], formato così da 3 elementi
48     var posto = posti[i];
49     // creazione marker con coordinate lette da posto[] e posto[2] è titolo da posto[0]
50     var myLatlng = new google.maps.LatLng(posto[1], posto[2]);
51     var marker = new google.maps.Marker({
52         position: myLatlng,
53         map: map,
54         icon: image,
55         title: posto[0]
56     });
57     // inserimento del marker nella mappa (da ripetere per tutto l'array)
58     marker.setMap(map);
59 }
60 }
```

L'oggetto `myLatlng` di classe `LatLng` contiene le coordinate di latitudine e longitudine lette dagli elementi di posizione **1** e **2** dell'array `posto[]` e dopo aver istanziato un `marker` (**righe 51-56**) questo viene collocato nella mappa (**riga 58**).

Descriviamo la sezione di **AJAX** che è “racchiusa” nella funzione `mostra()`: questa funzione viene chiamata dalla sezione **HTML** (**riga 120**) ogni volta che l'utente digita un carattere nella casella di testo ricevendo come parametro `str` la stringa digitata fino a quel momento. L'istruzione di **riga 65** valuta il valore della stringa e se ha lunghezza nulla con l'istruzione di **riga 66** viene cancellato il contenuto del `<DIV> 'elenco'`.

In base al tipo di browser viene creato un oggetto `xmlhttp` con proprietà `onreadystatechange` che contiene la funzione anonima necessaria per elaborare i dati in provenienza dal server (**riga 77**):

- la proprietà `readyState` contiene lo stato della response: se è **4** significa che la richiesta è completa;
- la proprietà `status` invece contiene lo stato della risposta: se **200** significa che la risposta è corretta, mentre invece se vale **404** indica “Not found”.

Nella **riga 86** viene verificato il valore di questi attributi e in caso di esito positivo viene collocata la proprietà `responseText` nel riquadro riservato per il testo (**riga 89**), cioè viene inserito l'elenco dei nomi ricevuti dal server all'interno del `<DIV> 'elenco'`.

```

64 function mostra(str){
65     if (str.length==0){
66         document.getElementById("elenco").innerHTML="";
67         return;
68     }
69     if (window.XMLHttpRequest){
70         xmlhttp = new XMLHttpRequest(); // browser è IE7+, Firefox, Chrome, Opera, Safari
71     }
72     else{
73         xmlhttp = new ActiveXObject("Microsoft.XMLHTTP"); // browser è IE6, IE5
74     }
75     // dopo una richiesta al server, serve una funzione che possa ricevere i dati
76     // restituiti dal server. onreadystatechange contiene la funzione che lavora tali dati.
77     xmlhttp.onreadystatechange = function(){
78         // readyState contiene lo stato della response del server. Quando cambia readyState
79         // viene eseguita la funzione onreadystatechange
80         // 0 La richiesta non è inizializzata
81         // 1 È stabilita la connessione col server
82         // 2 La richiesta è ricevuta
83         // 3 La richiesta è processata
84         // 4 La richiesta è completa
85         // status: 200 OK, 404 Not found
86         if (xmlhttp.readyState==4 && xmlhttp.status==200){
87             // i dati trasmessi dal server sono ricavati da responseText di tipo di stringa
88             // e vengono inseriti nel div chiamato 'elenco'
89             document.getElementById("elenco").innerHTML=xmlhttp.responseText;
90 }
```

L'ultima parte della sezione di **AJAX** è assai interessante: viene dichiarato un array di comodo chiamato **tmp[]** che conterrà le singole righe dell'elenco dei Comuni (**riga 91**) e una variabile **stringa** che conterrà l'elenco così come ricevuto dal server (**riga 93**).

```

90     // utilizzo di un array temporaneo
91     var tmp = new Array()
92     // variabile stringa contiene tutto l'elenco di città e coord ricevute dalla pag.php
93     var stringa = xmlhttp.responseText;
94     // viene creato un array formato dalle righe dell'elenco di città
95     tmp=stringa.split("<BR>");
96     conta=0;
97     posti=[];

```

La **stringa** è formata da diverse righe, ciascuna delle quali termina con il tag **<BR>** necessario per andare a capo: il metodo **split(" <BR> ")** di **riga 95** crea un array (**tmp**) dove ogni elemento è una riga dell'elenco, come vediamo dall'esempio che segue:

**stringa:**

Millesimo,44.3641,8.2059<BR>Milano,45.4612,9.1878<BR>Milzano,45.2745,10.2003<BR>Milazzo,38.2194,15.2404<BR>Militello Rosmarino,38.0472,14.6778<BR>Milena,37.4708,13.7361...

**tmp:**

0	Millesimo,44.3641,8.2059
1	Milano,45.4612,9.1878
2	Milzano,45.2745,10.2003
3	Milazzo,38.2194,15.2404
4	Militello Rosmarino,38.0472,14.6778
...	

Nel ciclo for successivo (**righe 99-102**) viene effettuata una conversione dell'array **tmp**, dove ciascuna riga viene aggiunta alla matrice **posti**, così che l'array **posti** possiede 3 colonne e un numero di righe pari agli elementi letti dall'array **tmp**. Viene infine chiamata la funzione **init()** che aggiornerà la mappa con i nuovi marker contenuti nell'array **posti** (**riga 104**). Il metodo **open()** (**riga 111**) invia la richiesta alla pagina **elabora.php** del server e infine il metodo **send()** trasmette effettivamente la richiesta al server (**riga 114**).

```

98     // ciclo che crea la matrice posti con 3 campi: nome, latitudine, longitudine
99     for(i=0;i<tmp.length;i++){
100         posti[conta]=tmp[i].split(",");
101         conta++;
102     }
103     //Chiamata della funzione init() che aggiorna la mappa con i nuovi elementi
104     init();
105 }
106 }
107 // metodo open con 3 parametri:
108 // - il primo definisce quale metodo usare (GET/POST)
109 // - il secondo è l'url dove risiede lo script server-side
110 // - il terzo (booleano) specifica che la richiesta deve essere asincrona
111 xmlhttp.open("GET","elabora.php?stringa='"+str,true);
112 // trasmette effettivamente la richiesta al server. il parametro nullo indica POST
113 // per GET mettere i parametri tra parentesi
114 xmlhttp.send();
115 }

```

La parte finale del codice è rappresentata dal codice **HTML** che mostra la casella di testo in cui digitare il Comune da ricercare (**riga 120**) alla quale è associato l'evento **onkeyup** che richiama la funzione **mostra(this.value)** di cui abbiamo già parlato in precedenza.

Vengono poi mostrati due <DIV> per l'elenco e la mappa (righe 121-122):

```

116  </SCRIPT>
117  </HEAD>
118  <BODY>
119  <P><B>Località da cercare:</B></P>
120  <FORM>Nome: <INPUT TYPE="text" onkeyup="mostra(this.value)"></FORM>
121  <P>Elenco località:<div class="box" id="elenco"></div></P>
122  <DIV id="mappa" class="mappa"></DIV>
123  </BODY></HTML>
```

La pagina `elabora.php` riceve il campo POST `stringa` e lo salva nella variabile `$q` (riga 3), quindi viene inizializzata la stringa `$città` che verrà restituita con i nomi delle località (riga 4).

```

1 <?php
2 // ricevo il campo GET con il nome della città dalla pagina precedente
3 $q = $_GET["stringa"];
4 $città = ""; // stringa che verrà restituita con i nomi delle località
5 if (strlen($q) > 0){ // presenta parola
6     $db = new mysqli("localhost", "root", "", "provephp");
7     if (mysqli_connect_errno()) { // verifico l'avvenuta connessione
8         printf("Connessione non effettuata: %s\n", mysqli_connect_error());
9         exit();
10    }

```

Siccome la pagina può essere richiamata senza che venga inviata alcuna stringa, lo script effettua preliminarmente un controllo sulla lunghezza della stringa (riga 5), se maggiore di 0 viene effettuata la connessione al database `provephp` di MySQL (righe 6-9).

Quindi viene memorizzata una query di selezione (riga 11) ed eseguita (riga 12).

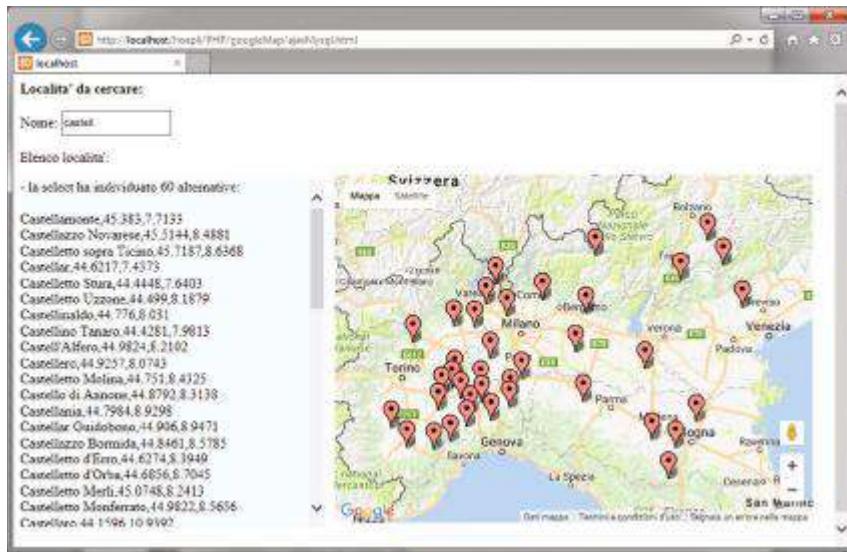
Come possiamo notare nella query di selezione viene usata la clausola `like` seguita dalla stringa di ricerca seguita dal simbolo percentuale (%). Questo consente di confrontare il nome della località con la sottostringa indicata prima del simbolo di percentuale. In tal modo possiamo ottenere le località che iniziano per alcuni caratteri.

```

11 $query = "SELECT Name,lat{lng FROM comuni WHERE Name like '".$q."%';";
12 if ($res = $db->query($query)) { // esecuzione query
13     printf("- la select ha individuato %d alternative:"."<BR>", $res->num_rows);
14 }
15 foreach( $res as $riga ) { // ciclo che estrae i record restituiti da SQL
16     $città.= $riga["Name"].",".$riga["lat"].",".$riga["lng"]."<BR>";
17 }
18 $db->close(); // chiusura connessione
19 }
20 if (strlen($città) == 0){ // controllo se esiste un risultato
21     echo "nessun nome trovato!";
22 }
23 else{
24     echo $città;
25 }
26 ?>
```

Il ciclo `foreach` che legge i risultati (righe 15-17) della query concatena i campi ricevuti nella variabile `$città` separandoli con il tag HTML `<BR>` e se al termine dell'iterazione la stringa `$città` ha lunghezza maggiore di zero viene inviata in risposta alla pagina precedente (righe 20-24).

L'esecuzione è mostrata di seguito dove possiamo notare che vengono mostrati i marker di tutti i Comuni il cui nome inizia per il testo digitato nella casella **Nome**:



A mano a mano che l'utente completa la parola nella casella di testo si riduce la “numerosità” dei marker in quanto sempre meno Comuni avranno la parte iniziale uguale alla stringa inserita.



## Prova adesso!



**APRI IL FILE** ajaxMysql.html e elabora.php

- Utilizzare le classi Map, Marker, event, InfoWindow
- Utilizzare gli ascoltatori di evento
- Utilizzare PHP
- Utilizzare il codice SQL di MySQL

Modifica il codice in modo tale che facendo click sulla mappa vengano lette le coordinate geografiche del click e appaia in una **infowindow** il nome del Comune più vicino al click effettuato dall'utente. Per fare questo utilizza **PHP** e la tabella **comuni** del database **localita** in cui ricercare le coordinate ricevute.

## AREA digitale



**Lab. 8** Inviare un file con i socket in PHP



VERSIONE  
SCARICABILE  
**EBOOK**

e-ISBN 978-88-203-7766-3

**[www.hoepliscuola.it](http://www.hoepliscuola.it)**

Ulrico Hoepli Editore S.p.A.  
via Hoepli, 5 - 20121 Milano  
e-mail [hoepliscuola@hoepli.it](mailto:hoepliscuola@hoepli.it)