
CORSO DI LAUREA IN INFORMATICA

# Tecnologie Software per il Web

AJAX & JSON

Docente: prof. Romano Simone
a.a. 2024-2025

# AJAX: **A**synchronous **J**avaScript **A**nd **X**ml


AJAX is a developer's dream, because you can:

- Read data from a web server - after a web page has loaded
- Update a web page without reloading the page
- Send data to a web server - in the background

https://www.w3schools.com/whatis/whatis_ajax.asp

# Un nuovo modello

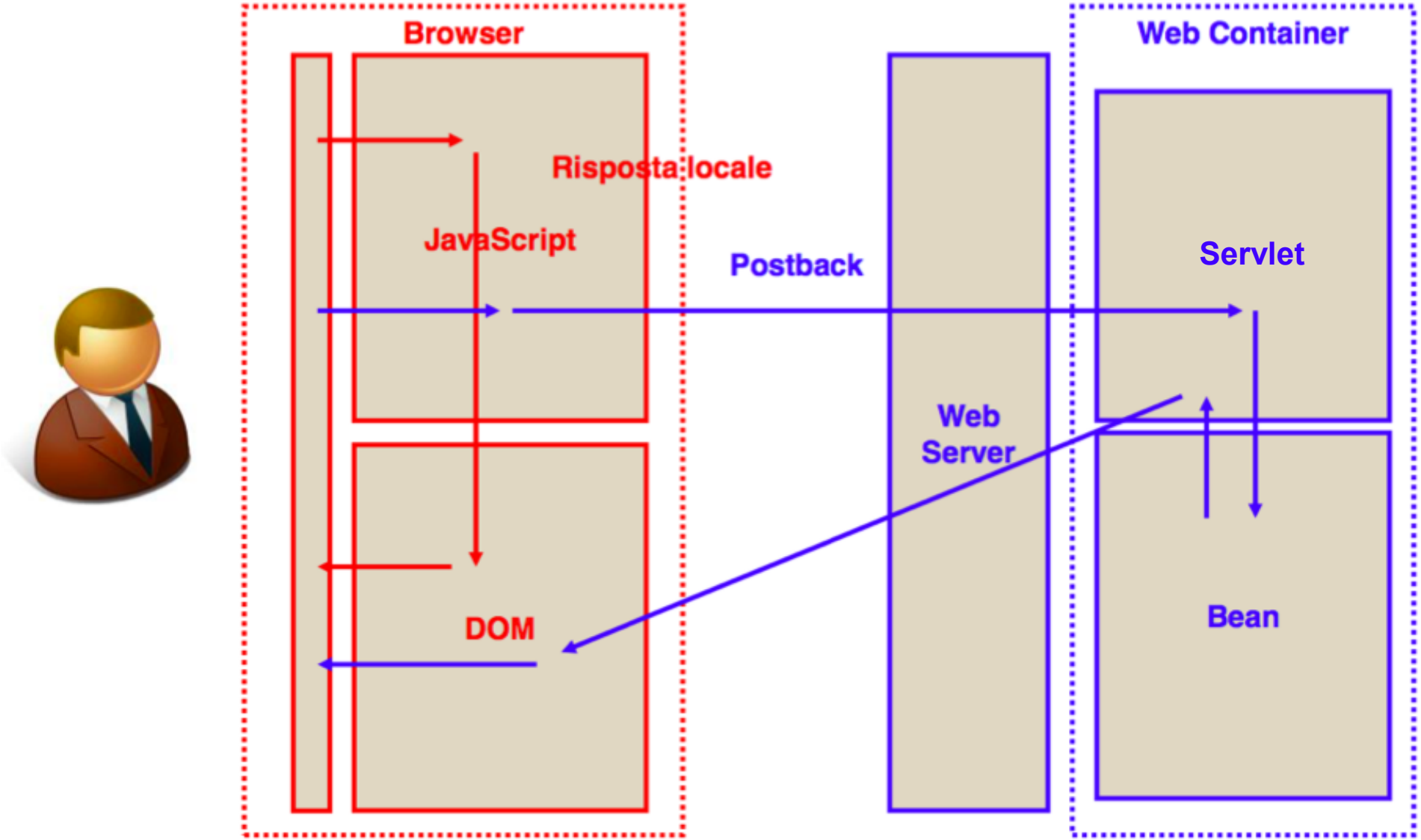
- L'utilizzo di **DHTML** (client-side script + HTML + CSS) delinea un nuovo modello per applicazioni Web

**=>**

*Modello a eventi simile a quello delle applicazioni tradizionali*

- A livello concettuale abbiamo però due livelli di eventi:
  - **Eventi locali** che portano ad una modifica diretta DOM da parte di JavaScript e quindi a cambiamenti locali della pagina
  - **Eventi remoti** ottenuti tramite ricaricamento della pagina che viene modificata lato server in base ai parametri passati in GET o POST
- Il ricaricamento di pagina per rispondere a un'interazione con l'utente prende il nome di **postback**

# Modello a eventi a due livelli



# Esempio di postback

- Consideriamo un form in cui compaiono due tendine che servono a selezionare il comune di nascita di una persona
  - Una con le province
  - Una con i comuni

- Si vuole fare in modo che scegliendo la provincia nella prima tendina, nella seconda appaiano solo i comuni di quella provincia

# Esempio di postback (2)

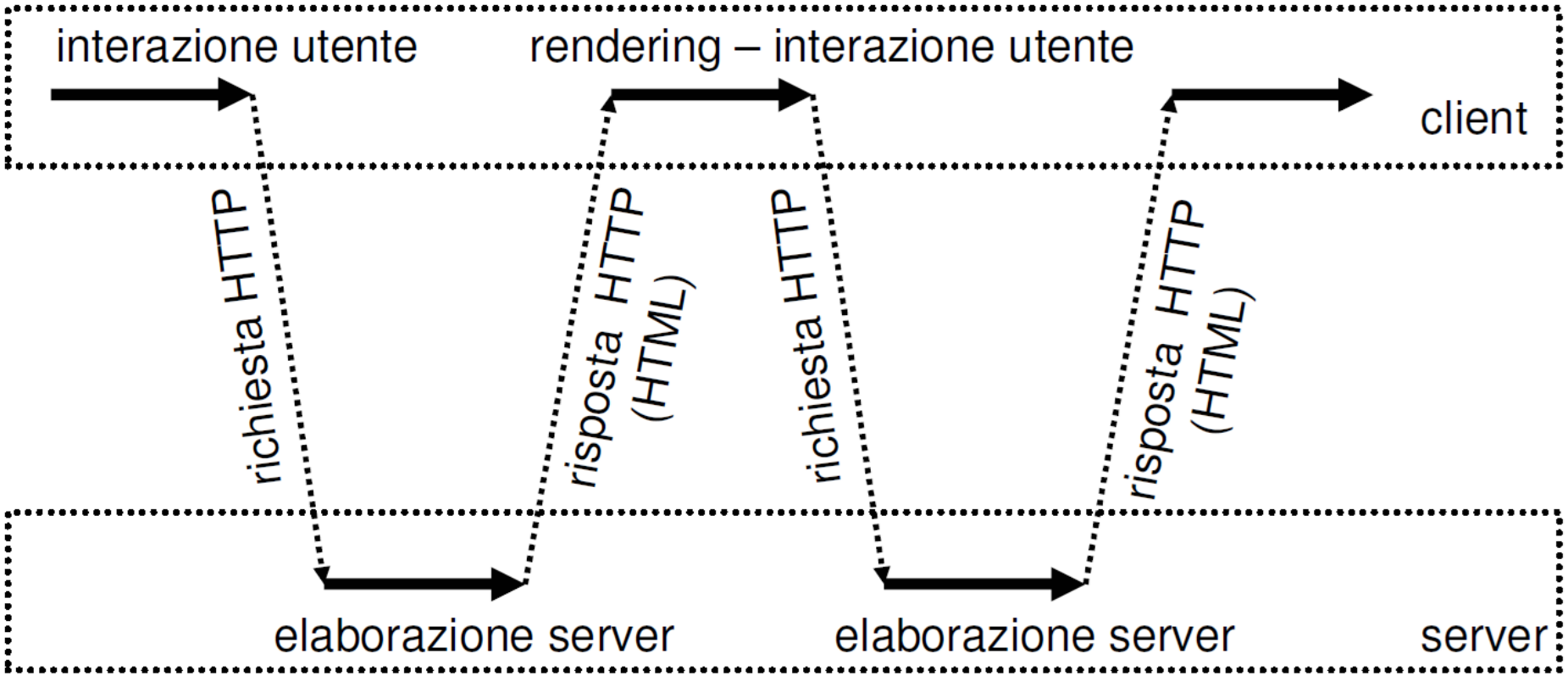
- Per realizzare questa interazione si può procedere in questo modo:
  1. Si crea una JSP che inserisce nella tendina dei comuni l'elenco di quelli che appartengono alla provincia passata come parametro
  2. Si definisce un evento **onChange** collegato all'elemento **select** delle province
  3. Lo script collegato ad onChange forza il ricaricamento della pagina con una richiesta get/post (**postback**)

- Quindi:
  - L'utente sceglie una provincia
  - Viene invocata la JSP con parametro provincia impostato al valore scelto dall'utente
  - La pagina restituita contiene nella tendina dei comuni l'elenco di quelli che appartengono alla provincia scelta

# Limiti del modello a ricaricamento di pagina

- Quando lavoriamo con applicazioni desktop siamo abituati a un elevato livello di interattività:
  - applicazioni reagiscono in modo rapido e intuitivo ai comandi
- Applicazioni Web tradizionali espongono invece un modello di interazione rigido
  - Modello ***"Click, wait, and refresh"***
  - È necessario refresh della pagina da parte del server per la gestione di qualunque evento (sottomissione di dati tramite form, visita di link per ottenere informazioni di interesse, ...)
- È ancora un **modello sincrono**: l'utente effettua una richiesta e deve attendere la risposta da parte del server

# Modello di interazione classico



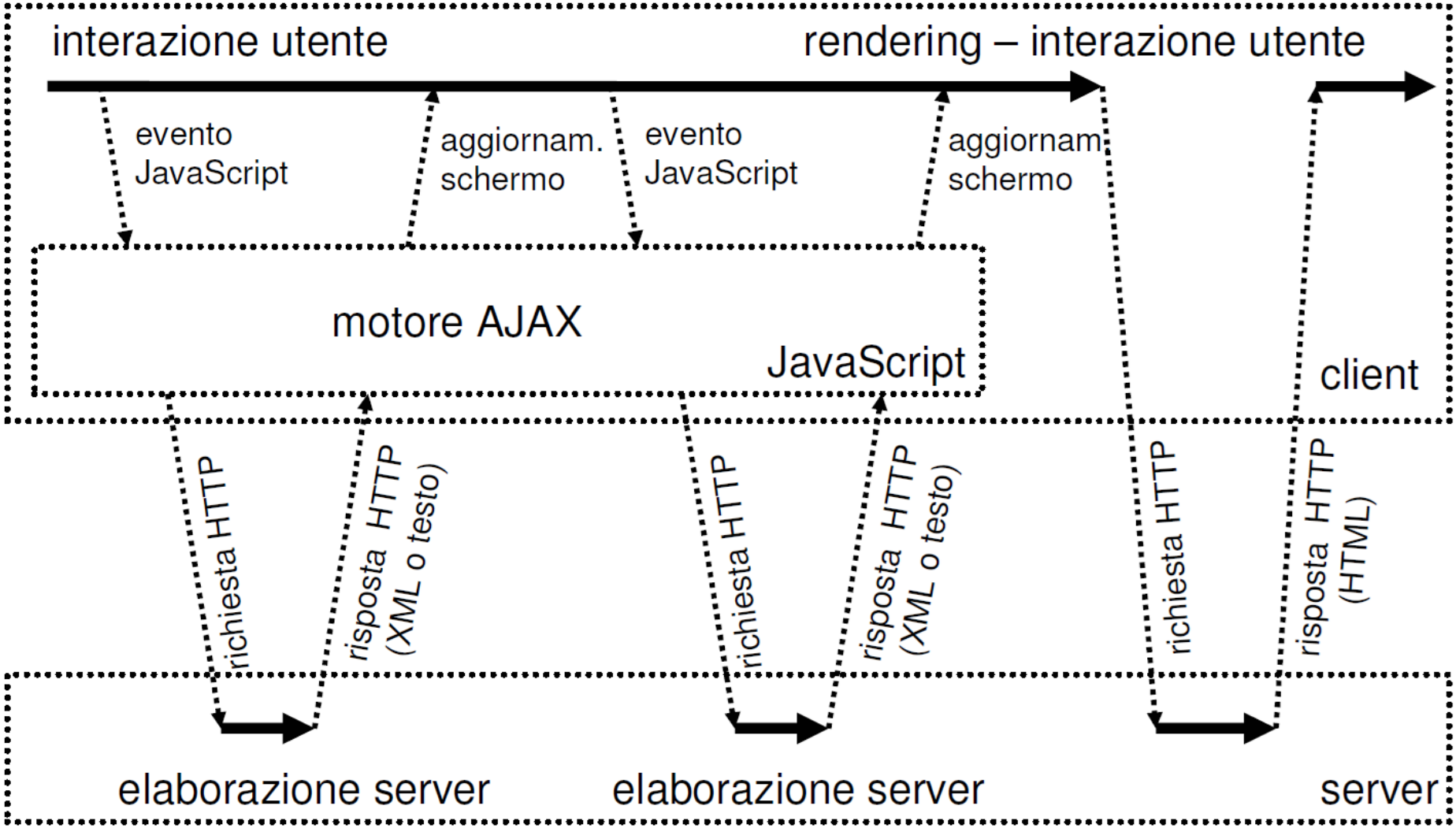
# AJAX e asincronicità

- Il modello di interazione (*tecnologia?*) **AJAX** è nato per superare queste limitazioni
- **AJAX** sta per **A**synchronous **J**avascript **A**nd **X**ml
- È basato su tecnologie standard e combinate insieme per realizzare un modello di interazione più ricco:
  - JavaScript/Eventi
  - DOM
  - XML (o JSON)
  - HTML
  - CSS

# AJAX e asincronicità

- AJAX punta a supportare applicazioni user friendly con elevata interattività (si usa spesso il termine RIA: **Rich Interface Application**)
- L'idea alla base di AJAX è quella di consentire agli script JavaScript di interagire direttamente con il server
- L'elemento centrale è l'utilizzo dell'oggetto JavaScript **XMLHttpRequest**
  - Consente di ottenere dati dal server senza necessità di ricaricare l'intera pagina
  - Realizza una comunicazione **asincrona** tra client e server: *il client non interrompe interazione con utente anche quando è in attesa di risposte dal server*

# Modello di interazione con AJAX



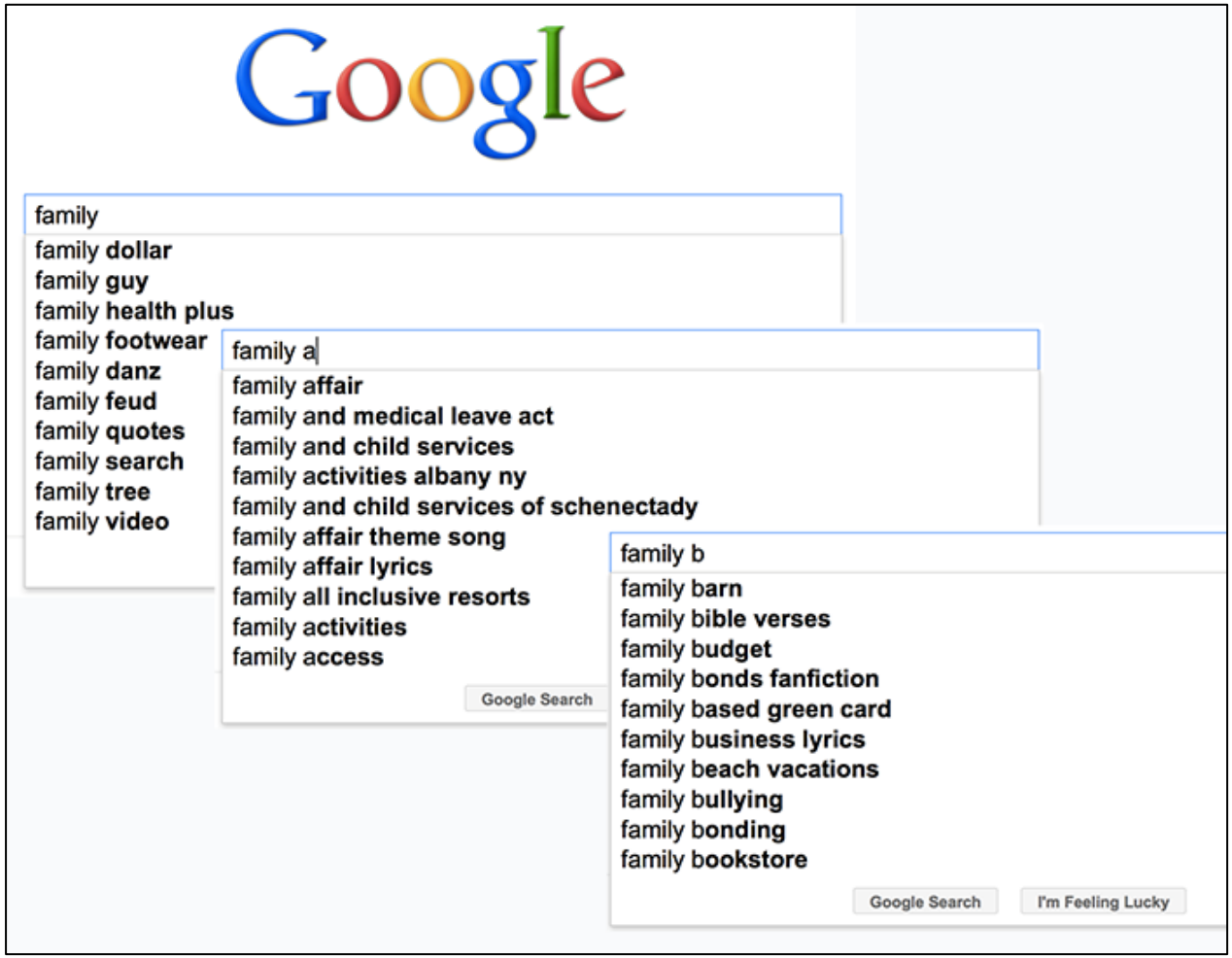
# Tipica sequenza AJAX

1. Si verifica un evento determinato dall'interazione fra utente e pagina Web
2. L'evento comporta l'esecuzione di una funzione JavaScript in cui:
   - Si istanzia un oggetto di tipo **XMLHttpRequest**
   - Si configura l'oggetto XMLHttpRequest: si associa una funzione di **callback**, si effettua una configurazione, …
   - Si effettua chiamata asincrona al server
3. Il server elabora la richiesta e risponde al client
4. Il browser invoca la funzione di **callback** che:
   - elabora il risultato
   - aggiorna il DOM della pagina per mostrare i risultati dell'elaborazione

# Google suggest

- *AJAX was made popular in 2005 by Google, with Google Suggest…*

- Google Suggest is using AJAX to create a very dynamic web interface: When you start typing in Google's search box, a JavaScript sends the letters off to a server and the server returns a list of suggestions



# XMLHttpRequest

- È l'oggetto **XMLHttpRequest** che si occupa di effettuare la richiesta di una risorsa via HTTP a server Web
  - Può inviare eventuali informazioni sotto forma di parametri (come avviene con una form)
- Può effettuare sia richieste GET che POST
- Le richieste possono essere di tipo
  - **Sincrono**: blocca flusso di esecuzione del codice JavaScript (*NON ci interessa*)
  - **Asincrono**: **NON** interrompe il flusso di esecuzione del codice JavaScript, **NÉ** le operazioni dell'utente sulla pagina (**thread dedicato**)

# Creazione di un’istanza: dettagli browser-specific

- I browser recenti supportano **XMLHttpRequest** come oggetto nativo
- Per creare un oggetto **XMLHttpRequest**, basta invocarne il costruttore:

**var xhr = new XMLHttpRequest();**

- *La gestione della compatibilità con browser "molto" vecchi complica un pò le cose, la esamineremo in seguito per non rendere difficile la comprensione del modello di interazione AJAX*

# Metodi di XMLHttpRequest

- La lista dei metodi disponibili è diversa da browser a browser
- In genere si usano solo quelli presenti in Safari (sottoinsieme più limitato, ma comune a tutti i browser che supportano AJAX):
  - **open()**
  - **setRequestHeader()**
  - **send()**
  - **getResponseHeader()**
  - **getAllResponseHeaders()**
  - **abort()**

# Metodo open()

- **open()** ha lo scopo di inizializzare la richiesta da formulare al server
- Lo standard W3C prevede 5 parametri, di cui 3 opzionali:

  **open (method, uri [,async][,user][,password])**

- L'uso più comune per AJAX ne prevede 3, di cui uno comunemente fissato:

  **open (method, uri, true)**

- Dove:
  - **method**: stringa e assume il valore "get" o "post"
  - **uri**: stringa che identifica la risorsa da ottenere (URL assoluto o relativo)
  - **async**: valore booleano che deve essere impostato come **true** per indicare al metodo che la richiesta da effettuare è di tipo asincrono

# Metodi setRequestHeader() e send()

- **setRequestHeader(nomeheader, valore)** consente di impostare gli header HTTP della richiesta da inviare
  - Viene invocata più volte, una per ogni header da impostare
  - Per una richiesta GET gli header sono opzionali
  - Sono invece necessari per impostare la codifica utilizzata nelle richieste POST
  - È comunque importante impostare l'header **connection** al valore **close** (*"close" connection option for the sender to signal that the connection will be closed after completion of the response*)
- **send(body)** consente di inviare la richiesta al server
  - Non è bloccante se il parametro async di open è stato impostato a true. *Che cosa succederebbe altrimenti?*
  - Prende come parametro una stringa che costituisce il body della richiesta HTTP di tipo get, o null se la richiesta è di tipo get

# Esempi

**GET**

```
var xhr = new XMLHttpRequest();
xhr.open("get","pagina.html?p1=v1&p2=v2", true );
xhr.setRequestHeader("connection", "close");
xhr.send(null);
```

**POST**

```
var xhr = new XMLHttpRequest();
xhr.open("post","pagina.html", true );
xhr.setRequestHeader("content-type",
      "x-www-form-urlencoded");
xhr.setRequestHeader("connection","close");
xhr.send("p1=v1&p2=v2");
```

# Create a new XMLHttpRequest (gestione della compatibilità)

```
function createXMLHttpRequest() {
    var request;
    try {
        // Firefox 1+, Chrome 1+, Opera 8+, Safari 1.2+, Edge 12+, Internet Explorer 7+
        request = new XMLHttpRequest();
    } catch (e) {
        // past versions of Internet Explorer
        try {
            request = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (e) {
            try {
                request = new ActiveXObject("Microsoft.XMLHTTP");
            } catch (e) {
                alert("Il browser non supporta AJAX");
                return null;
            }
        }
    }
    return request;
}
```

# Proprietà di XMLHttpRequest

- Stato e risultati della richiesta vengono memorizzati dall'interprete JavaScript all'interno dell'oggetto durante la sua esecuzione
- Le proprietà **XmlHttpRequest** comunemente supportate dai vari browser sono:
  - **readyState**
  - **onreadystatechange**
  - **status**
  - **statusText**
  - **responseText**
  - **responseXML** (non disponibile in versioni di IE precedenti alla 7)

# Proprietà readyState

- Proprietà in sola lettura di tipo intero che consente di leggere in ogni momento lo stato della richiesta
- Ammette 5 valori:
  - 0: **uninitialized** - l'oggetto esiste, ma non è stato ancora richiamato open()
  - 1: **open** - è stato invocato il metodo open(), ma send() non ha ancora effettuato l'invio dati
  - 2: **sent** - metodo send() è stato eseguito e ha effettuato la richiesta
  - 3: **receiving** - la risposta ha cominciato ad arrivare
  - 4: **loaded** - l'operazione è stata completata
- Attenzione:
  - *Questo ordine non è sempre identico e non è sfruttabile allo stesso modo su tutti i browser*
  - **L'unico stato supportato da tutti i browser è il 4**

# Proprietà onreadystatechange

- Come si è detto l'esecuzione del codice non si blocca sulla ***send()*** in attesa dei risultati
- Per gestire la risposta si deve quindi adottare un approccio a eventi
- Occorre registrare una funzione di ***callback*** che viene richiamata in modo asincrono ad ogni cambio di stato della proprietà readyState
- La sintassi è:

  **xhr.onreadystatechange = *nomefunzione***

  **xhr.onreadystatechange = function() {*istruzioni*}**

- *Attenzione: per evitare comportamenti imprevedibili l'assegnamento va fatto prima del* ***send()***

# Proprietà status e statusText

- **status** contiene un valore intero corrispondente al codice HTTP dell'esito della richiesta:
  - **200** in caso di successo (l'unico in base al quale i dati ricevuti in risposta possono essere ritenuti corretti e significativi)
  - Possibili altri valori (in particolare, errore: 403, 404, 500, ...)
- **statusText** contiene invece una descrizione testuale del codice HTTP restituito dal server...

Esempio:

```
if (xhr.status != 200 ) alert( xhr.statusText );
```

# Proprietà responseText e responseXML

- Contengono i dati restituiti dal server
  - **responseText** stringa che contiene il body della risposta HTTP
    - disponibile solo a interazione ultimata, cioè:

      (**readyState==4**)

  - **responseXML** body della risposta convertito in documento XML (*se possibile*)
    - anch'esso disponibile solo a interazione ultimata
    - consente la navigazione del documento XML via JavaScript
    - può essere **null** se i dati restituiti non sono un documento XML ben formato

# Metodi getResponseHeader() e getAllResponseHeaders()

- Consentono di leggere gli header HTTP che descrivono la risposta del server
- Sono utilizzabili solo nella funzione di ***callback***
- Possono essere invocati sicuramente in modo safe solo a richiesta conclusa (**readyState==4**)
- In alcuni browser possono essere invocati anche in fase di ricezione della risposta (**readyState==3**)

Sintassi:

- **getAllResponseHeaders()**
- **getResponseHeader(*header_name*)**

# Ruolo della funzione di callback associata ad onreadystatechange

- Viene invocata ad ogni variazione di **readyState**
- Usa **readyState** per leggere lo stato di avanzamento della richiesta
- Usa **status** per verificare l'esito della richiesta
- Ha accesso agli header di risposta rilasciati dal server con

  **getAllResponseHeaders()** e **getResponseHeader()**
- Se **readystate==4** può leggere il contenuto della risposta con **responseText** e **responseXML**

# Esempio

https://www.w3schools.com/xml/tryit.asp?filename=tryajax_first

```
<!DOCTYPE html>
<html>
<body>

<div id="demo">
<h1>The XMLHttpRequest Object</h1>
<button type="button" onclick="loadDoc()">Change Content</button>
</div>

<script>
function loadDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("demo").innerHTML =
      this.responseText;
    }
  };
  xhttp.open("GET", "ajax_info.txt", true);
  xhttp.send();
}
</script>

</body>
</html>
```

# Gestire la risposta XML

- AJAX è in grado di ricevere **documenti XML** accessibili tramite la proprietà **responseXML** dell'oggetto **XMLHttpRequest**
- É possibile elaborare i documenti XML ricevuti utilizzando l'API W3C DOM
- Il modo con cui operiamo su dati in formato XML è analogo a quello che abbiamo visto per ambienti Java
- Usiamo un parser e accediamo agli elementi del DOM XML di nostro interesse
  - L'oggetto **XMLHttpRequest** ha un in-built XML parser

```
var response = request.responseXML.documentElement;
alert("Risposta: \n" + request.responseText);
var result = response.getElementsByTagName("result")[0].firstChild.nodeValue;
document.getElementById("datiCapoluogo").innerHTML = result;
```

Ottengo il nodo testuale figlio del primo nodo "result"

- Per visualizzare i contenuti ricevuti modifichiamo il DOM della pagina HTML

# Esempio

- Scegliamo un nome da una lista e mostriamo i suoi dati tramite Ajax

```
<html>
  <head>
    <script src="selectmanager_xml.js"></script>
  </head>
  <body>
    <form action=""> Scegli un contatto:
    <select name="manager"
       onchange="showManager(this.value)">
    <option value="Carlo11">Carlo Rossi</option>
    <option value="Anna23">Anna Bianchi</option>
    <option value="Giovanni75">Giovanni Verdi</option>
  </select></form>
  <b><span id="companyname"></span></b><br/>
     <span id="contactname"></span><br/>
     <span id="address"></span>
     <span id="city"></span><br/>
     <span id="country"></span>
  </body>
</html>
```

Lista di selezione

Area in cui mostrare i risultati

# Esempio (2)

- Ipotizziamo che i dati sui contatti siano contenuti in un database. Il server:
  - riceve una request con l'identificativo della persona
  - interroga il database
  - restituisce un file XML con i dati richiesti

```
<?xml version='1.0' encoding='UTF-16'?>
<company>
<compname>Microsoft</compname>
<contname>Anna Bianchi</contname>
<address>Viale Risorgimento 2</address>
<city>Bologna</city>
<country>Italy</country>
</company>
```

# Esempio: selectmanager_xml.js

```
var xmlHttp;
function showManager(str)
{ xmlHttp=new XMLHttpRequest();
  var url="getmanager_xml.jsp?q="+str;
  xmlHttp.onreadystatechange=stateChanged;
  xmlHttp.open("GET",url,true);
  xmlHttp.send(null);
}
function stateChanged()
{ if (xmlHttp.readyState==4) && (xmlHttp.state == 200)
  {
    var xmlDoc=xmlHttp.responseXML.documentElement;
    var compEl=xmlDoc.getElementsByTagName("compname")[0];
    var comName = compEl.childNodes[0].nodeValue;
    document.getElementById("companyname").innerHTML=
      compName;
    ...
  }
}
```

# Funzioni di Callback Multiple

- Il codice mostrato in precedenza non è in grado di svolgere più di un AJAX task (l'oggetto XMLHttpRequest è referenziato tramite una variabile globale)
- Se si devono eseguire più AJAX task in una web application, occorre creare una funzione per l'esecuzione dell'oggetto XMLHttpRequest e una funzione di callback per ciascun AJAX task
  - La chiamata alla funzione dovrebbe contenere (almeno) l'URL e la funzione di callback da chiamare quando la risposta è pronta

```
loadDoc("url-1", myFunction1);
loadDoc("url-2", myFunction2);

function loadDoc(url, cFunction) {
  const request = new XMLHttpRequest();
  request.onreadystatechange = function() {cFunction(this);}
  request.open("GET", url, true);
  request.send();
}

function myFunction1(request) {
  // action goes here
}
function myFunction2(request) {
  // action goes here
}
```

# Example: ajax call (index.jsp in ajax.zip)

```
<div>
    <!-- "javascript:void(0)" is used here to avoid submitting the form -->
    <form id="form" action="javascript:void(0)" method="get">
        <p>
            CAP: <input type="text" id="CAP" name="CAP" onchange="cercaCapoluogo()"/>
            <input type="submit" value="Cerca Capoluogo" onclick="cercaCapoluogo()" />
        </p>
    </form>
</div>

<p>
    Capoluogo: <strong><span id="datiCapoluogo"></span></strong>
</p>
```

**cercaCapoluogo** (in questo caso) gestisce sia l'evento change che l'evento cerca

# Example: ajax call (2)

```
function cercaCapoluogo() {
    var input = document.getElementById('CAP').value;
    var params = 'CAP=' + input;
    loadAjaxDoc('cercaCapoluogo', "GET", params, handleCAP);
}

function createXMLHttpRequest() {
    var request;
    try {
        // Firefox 1+, Chrome 1+, Opera 8+, Safari 1.2+, Edge 12+, Internet Explorer 7+
        request = new XMLHttpRequest();
    } catch (e) {
        // past versions of Internet Explorer
        try {
            request = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (e) {
            try {
                request = new ActiveXObject("Microsoft.XMLHTTP");
            } catch (e) {
                alert("Il browser non supporta AJAX");
                return null;
            }
        }
    }
    return request;
}
```

**createXMLHttpRequest** lo abbiamo già visto qualche slide fa…

# Example: ajax call (3)

```java
@WebServlet("/cercaCapoluogo")
public class ServletCercaCapoluogo extends HttpServlet {

    /**
    private static final long serialVersionUID = 1L;

    private IDAOCapoluogo daoCapoluogo = new DAOCapoluogoMock();

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

        response.setContentType("text/xml");
        PrintWriter out = response.getWriter();
        String CAP = request.getParameter("CAP");
        Capoluogo capoluogo = null;
        if (CAP != null && !CAP.equals("")) {
            capoluogo = daoCapoluogo.findCapolougoByCAP(CAP);
        }
        String risultato = null;
        if (capoluogo != null) {
            risultato = capoluogo.getNome() + " (" + capoluogo.getRegione() + ")";
        } else {
            risultato = "non trovato";
        }
        out.println("<?xml version=\"1.0\" encoding=\"iso-8859-1\" ?>");
        out.println("<response>");
        out.println("<functionName>aggiornaDatiCapoluogo</functionName>");
        out.println("<result>" + risultato + "</result>");
        out.println("</response>");
    }

    /** Handles the HTTP <code>GET</code> method.
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }
```

# Example: ajax call (4)

```
function loadAjaxDoc(url, method, params, cFuction) {
    var request = createXMLHttpRequest();
    if(request){

        request.onreadystatechange = function() {
            if (this.readyState == 4) {
                if (this.status == 200) {
                    cFuction(this);
                } else {
                    if(this.status == 0){ // When aborting the request
                        alert("Problemi nell'esecuzione della richiesta: nessuna risposta ricevuta nel tempo limite");
                    } else { // Any other situation
                        alert("Problemi nell'esecuzione della richiesta:\n" + this.statusText);
                    }
                    return null;
                }
            }
        };

        setTimeout(function () {      // to abort after 15 sec
            if (request.readyState < 4) {
                request.abort();
            }
        }, 15000);
        …
```

di **setTimeout** ne parliamo tra qualche slide..ignoriamolo per ora…

**loadAjaxDoc** continua nella prossima slide

# Example: ajax call (5)

```
        ...
        if(method.toLowerCase() == "get"){
            if(params){
                request.open("GET", url + "?" + params, true);
            } else {
                request.open("GET", url, true);
            }
            request.setRequestHeader("Connection", "close");
            request.send(null);
        } else {
            if(params){
                request.open("POST", url, true);
                request.setRequestHeader("Connection", "close");
                request.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
                request.send(params);
            } else {
                console.log("Usa GET se non ci sono parametri!");
                return null;
            }
        }
    }
}
```

il codice riportato può gestire sia chiamate GET che POST

# Example: ajax call (6)

```
function handleCAP(request){
    var response = request.responseXML.documentElement;
    alert("Risposta: \n" + request.responseText);
    var result = response.getElementsByTagName("result")[0].firstChild.nodeValue;
    document.getElementById("datiCapoluogo").innerHTML = result;
}
```

**HandleCAP** è la funzione di callback usata quando **readystate=4** e **status=200**

# Vantaggi e svantaggi di Ajax

- Si guadagna in **interattività**, ma si perde la **linearità dell'interazione**
- Mentre l'utente è all'interno della stessa pagina le richieste sul server possono essere numerose e indipendenti
- Il tempo di attesa passa in secondo piano o non è avvertito affatto
- Possibili criticità sia per l’utente che per lo sviluppatore

# Criticità nell'interazione con l'utente

- Le richieste AJAX permettono all'utente di continuare a interagire con la pagina
- Ma non necessariamente lo informano di che cosa stia succedendo e possono durare troppo!

***=> L'effetto è un possibile disorientamento dell'utente***

- Di conseguenza, di solito si agisce su due fronti per limitare i comportamenti impropri a livello utente:
  1. Rendere visibile in qualche modo l'andamento della chiamata (barre di scorrimento, info utente, ...)
  2. Interrompere le richieste che non terminano in tempo utile per sovraccarichi del server o momentanei problemi di rete (timeout)

# Il metodo abort()

- **abort()** consente l'interruzione delle operazioni di invio o ricezione
  - non ha bisogno di parametri
  - termina immediatamente la trasmissione dati
- *Attenzione: non ha senso invocarlo dentro la funzione di **callback***
  - Se readyState non cambia, il metodo non viene richiamato; readyState non cambia quando la risposta si fa attendere
- Si crea un'altra funzione da far richiamare in modo asincrono al sistema mediante il metodo

  **setTimeOut(*funzioneAsincronaPerAbortire*, *timeOutInMillis*)**
- All'interno della ***funzioneAsincronaPerAbortire*** si valuta se continuare l'attesa o abortire l'operazione

```
setTimeout(function () {      // to abort after 15 sec
    if (request.readyState < 4) {
        request.abort();
    }
}, 15000);
```

# Aspetti critici per il programmatore

- *È accresciuta la complessità delle Web Application*
- Le applicazioni AJAX pongono problemi di debug, test e mantenimento
  - Il test di codice JavaScript è complesso
- Mancanza di standardizzazione di XMLHttpRequest e assenza di supporto nei vecchi browser


# XML VS. JSON

# XML è la scelta giusta?

- *Abbiamo però visto nell'esempio precedente che l'utilizzo di XML come formato di scambio fra client e server porta alla generazione e all'utilizzo di quantità maggiore di byte rispetto al testo piano*
  - Oneroso in termini di risorse di elaborazione (*non dimentichiamo che JavaScript è interpretato*)

- *Esiste un formato più efficiente e semplice da manipolare per scambiare informazioni tramite AJAX?*
  - La risposta è SI; questo formato è nella pratica industriale quello più utilizzato oggi

# JSON

- JSON è l'acronimo di JavaScript Object Notation
  - Formato per lo scambio di dati, considerato molto più comodo di XML
  - Leggero in termini di quantità di dati scambiati
  - Molto semplice ed efficiente da elaborare da parte del supporto runtime al linguaggio di programmazione (*in particolare per JavaScript*)
  - Ragionevolmente semplice da leggere per un operatore umano
  - È largamente supportato dai maggiori linguaggi di programmazione
  - Si basa sulla notazione usata per gli object literal e gli array literal in JavaScript

# Object Literal e Array Literal

- In JavaScript è possibile creare un oggetto mediante un object literal:

```
var Beatles = {
    Paese : "Inghilterra",
    AnnoFormazione : 1959,
    TipoMusica : "Rock"
}
```

- In modo analogo è possibile creare un array utilizzando un array literal

```
var Membri = ["Paul","John","George","Ringo"];
```

# Object Literal e Array Literal (2)

- Possiamo anche avere oggetti che contengono array:

```
var Beatles =
{
 "Paese" : "Inghilterra",
 "AnnoFormazione" : 1959,
 "TipoMusica" : "Rock",
 "Membri" : ["Paul","John","George","Ringo"]
}
```

# Object Literal e Array Literal (3)

- È infine possibile definire array di oggetti:

```
var Rockbands = [
 {
  "Nome" : "Beatles",
  "Paese" : "Inghilterra",
  "AnnoFormazione" : 1959,
  "TipoMusica" : "Rock",
  "Membri" : ["Paul","John","George","Ringo"]
 },
 {
  "Nome" : "Rolling Stones",
  "Paese" : "Inghilterra",
  "AnnoFormazione" : 1962,
  "TipoMusica" : "Rock",
  "Membri" : ["Mick","Keith","Charlie","Bill"]
 }
]
```

# La sintassi JSON

- La sintassi JSON si basa su quella degli object e array literal di JavaScript
- Una **stringa JSON** altro non è che la stringa equivalente ad un object o array literal JavaScript

**Object literal JavaScipt**

```
{
 "Paese" : "Inghilterra",
 "AnnoFormazione" : 1959,
 "TipoMusica" : "Rock'n'Roll",
 "Membri" : ["Paul","John","George","Ringo"]
}
```

**Stringa JSON**

```
'{"Paese" : "Inghilterra", "AnnoFormazione" :
1959, "TipoMusica" : "Rock'n'Roll", "Membri" :
["Paul","John","George","Ringo"] }'
```

# Parser JSON

- In passato, per ottenere un oggetto JavaScript a partire da una stringa JSON, si usava la funzione **eval()**
  - Uso di **eval**() presenta rischi di sicurezza: *stringa passata come parametro potrebbe contenere codice malevolo*
- Oggi, si utilizzano parser appositi che traducono solo oggetti JSON e non espressioni JavaScript di qualunque tipo
  - Per convertire una stringa JSON in un oggetto JavaScript, si usa quindi la funzione built-in di JavaScript **JSON.parse(JSONstring)**
  - Se si vuole convertire un oggetto JavaScript in una stringa JSON, si usa **JSON.stringify(object)**

# Esempio di parsing JSON

```
<!DOCTYPE html>
<html>
<body>

<h2>Create Object from JSON String</h2>

<p id="demo"></p>

<script>
let text = '{"employees":[' +
'{"firstName":"John","lastName":"Doe" },' +
'{"firstName":"Anna","lastName":"Smith" },' +
'{"firstName":"Peter","lastName":"Jones" }]}';

const obj = JSON.parse(text);
document.getElementById("demo").innerHTML =
obj.employees[1].firstName + " " + obj.employees[1].lastName;
</script>

</body>
</html>
```

**Create Object from JSON String**

Anna Smith

https://www.w3schools.com/js/tryit.asp?filename=tryjs_json_parse

# AJAX e ricezione file JSON

- Simile a quanto avviene con la ricezione di file XML, con due differenze, una lato server e una lato client
- Lato Server:
  - La servlet deve trasmettere la stringa JSON (non XML) nel body della risposta HTTP al client
    1. Occorre impostare il content type "application/json"
    2. Occorre un parser JSON (scarica da https://github.com/stleary/JSON-java la libreria **JSON-Java** e importala nel progetto Eclipse) per creare la stringa JSON
- Lato client:
  - Si converte la stringa JSON in un oggetto JavaScript usando

    **JSON.parse()**

* Un'altra libreria JSON molto usata in Java è GSON: https://mvnrepository.com/artifact/com.google.code.gson/gson

# Esempio (index-json.jsp in ajax-example.zip)

```
<body>
    <h1>Esempio AJAX con JSON</h1>

    <p>
        <a href="esci.jsp">Esci</a>
    </p>

    <div>
        <!-- "javascript:void(0)" is used here to avoid submitting the form -->
        <form id="form" action="javascript:void(0)" method="get">
            <p>
                CAP: <input type="text" id="CAP" name="CAP" onchange="cercaCapoluogo()"/>
                <input type="submit" value="Cerca Capoluogo" onclick="cercaCapoluogo()" />
            </p>
        </form>
    </div>

    <p>
        Capoluogo: <strong><span id="datiCapoluogo"></span></strong>
    </p>

</body>
```

# Esempio (2)

Lato Server:

```
@WebServlet("/cercaCapoluogoJson")
public class ServletCercaCapoluogoJson extends HttpServlet {

    /**
    private static final long serialVersionUID = 1L;

    private IDAOCapoluogo daoCapoluogo = new DAOCapoluogoMock();

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

        response.setContentType("application/json");
        PrintWriter out = response.getWriter();
        String CAP = request.getParameter("CAP");
        Capoluogo capoluogo = null;
        if (CAP != null && !CAP.equals("")) {
            capoluogo = daoCapoluogo.findCapolougoByCAP(CAP);
        }
        String risultato = null;
        if (capoluogo != null) {
            risultato = capoluogo.getNome() + " (" + capoluogo.getRegione() + ")";
        } else {
            risultato = "non trovato";
        }
        JSONObject json = new JSONObject();
        json.put("functionName", "aggiornaDatiCapoluogoJSON");
        json.put("result", risultato);
        out.print(json.toString());
    }

    /** Handles the HTTP <code>GET</code> method.
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }
```

# Esempio (3)

- Lato Client:

```
function handleCAP(request){
    var response = JSON.parse(request.responseText);
    alert("Risposta: \n" + request.responseText);
    document.getElementById("datiCapoluogo").innerHTML = response.result;
}
```

# Riferimenti

- AJAX introduction
  - **https://www.w3schools.com/xml/ajax_intro.asp**