Corso di Web Programming

7. JavaScript Parte II (Complementi)

Paolo Milazzo

Dipartimento di Informatica, Università di Pisa
http://www.di.unipi.it/∼milazzo
milazzo@di.unipi.it

Corso di Laurea in Informatica Applicata
A.A. 2010/2011

Sommario

Gli oggetti: creazione e manipolazione (1)

- Per il momento abbiamo usato JavaScript come un linguaggio fondamentalmente imperativo con qualche elemento di programmazione orientata agli oggetti e funzionale
- JavaScript è molto più object-oriented di quanto non sembri:
  - le variabili globali e le funzioni definite sono implicitamente proprietà e metodi dell'oggetto globale `window`
- Anche le funzioni in realtà sono internamente rappresentate come oggetti
  - Questo consente di simulare gli aspetti di programmazione funzionale che abbiamo visto (il passaggio di una funzione ad un'altra funzione)
- Vediamo ora come è possibile creare e manipolare oggetti in JavaScript

Gli oggetti: creazione e manipolazione (2)

- Differentemente da Java, in JavaScript non esiste un vero e proprio costrutto per la definizione di classi
- Per specificare le caratteristiche di un nuovo oggetto è sufficiente definirne il costruttore, che non è altro che una normalissima funzione che al suo interno fa uso di `this`

```
function Persona(nome, cognome, eta) {
  this.nome = nome;
  this.cognome = cognome;
  this.eta = eta;
}
```

- A questo punto si può usare `new` per creare un'istanza della "classe" che abbiamo definito

```
var p = new Persona("Paolo","Milazzo",32);
```

- E si può accedere alle proprietà dell'oggetto con la solita sintassi:

```
alert(p.nome+p.cognome);
```

Gli oggetti: creazione e manipolazione (3)

- Per definire i metodi di un oggetto è comodo usare le funzioni anonime:

```
function Persona(nome, cognome, eta) {
  this.nome = nome;
  this.cognome = cognome;
  this.eta = eta;
  this.hello = function() {
     return this.nome + " " + this.cognome + " ti saluta!";
  }
}
```

- E anche i metodi possono essere invocati con la solita sintassi

```
p.hello();
```

Gli oggetti: creazione e manipolazione (4)

- Esiste anche una notazione compatta per creare oggetti "anonimi" (senza usare un costruttore):

```
var p = {
  nome: "Paolo",
  cognome: "Milazzo",
  eta: 32,
  hello: function() {
    return this.nome + " " + this.cognome + " ti saluta!";
  }
}
```

Gli oggetti: creazione e manipolazione (5)

- Inoltre è anche possibile modificare aggiungere nuove proprietà o nuovi metodi a oggetti già costruiti semplicemente assegnandoli:

```
var p = {
  nome: "Paolo",
  cognome: "Milazzo",
}

p.eta = 32;
p.hello = function() {
    return this.nome + " " + this.cognome + " ti saluta!";
}
```

- Questo consente di creare oggetti dinamicamente partendo da un oggetto "vuoto" ottenuto tramite il costruttore `Object()`

```
var p = new Object();
p.nome: "Paolo";
p.cognome: "Milazzo";
p.eta = 32;
p.hello = function() {
    return this.nome + " " + this.cognome + " ti saluta!";
}
```

Gli oggetti: creazione e manipolazione (6)

- Per accedere alle proprietà di un oggetto si può usare la solita notazione "." , oppure la notazione degli array associativi
  - Gli array associativi sono array i cui indici possono non essere valori numerici, ma (solitamente) stringhe: si possono quindi accedere come segue `a["ciao"]`
- Questo consente di simulare gli array associativi tramite gli oggetti:

```
var elencotel = new Object();
elencotel["Paolo"] = "3391234567";
elencotel["Mario"] = "3387654321";
elencotel["Luca"] = "5556765654";
```

Gli oggetti: creazione e manipolazione (7)

- E' anche possibile ispezionare il contenuto di un oggetto usando la parola chiave `in`

```
p = new Object();
p.nome = "Paolo";
if ("nome" in p) alert ("il nome esiste ed e'" + p.nome);
```

Gli oggetti: creazione e manipolazione (8)

- Pur non essendoci un costrutto per la definizione di classi esiste comunque la possibilità di esprimere una forma di ereditarietà (estensione) tra gli oggetti
  - si usa la proprietà `prototype`, che è un oggetto visibile a tutti gli oggetti creati usando lo stesso costruttore (ad esempio `Array()`) e che può accedere alle proprietà della classe da estendere tramite `this`

```
var a = [3,5,4];

Array.prototype.sum = function() {
  var r=0;
  for (var i = 0; i < this.length; i++) {
     r += this[i];
  }
  return r;
}

var b = [1,2,3,4];

alert("la somma fa: " + a.sum()); // stampa 12
alert("la somma fa: " + b.sum()); // stampa 10
```

Le eccezioni

- Come molti altri linguaggi di programmazione, JavaScript consente di usare le eccezioni:
  - per intercettare e gestire gli errori nel codice
  - per interrompere l'esecuzione del proprio codice quando si verifichi una situazione che richiede l'intervento di un apposito gestore
- Si usa il solito costrutto `try/catch/throw`

```
try {
 alLert("ciao");
}
catch(e) {
 alert("Errore: " + e);
}
```

```
try { ....
 throw "Eccezione";
 ....
}
catch(e) {
 if (e == "Eccezione") alert("C'e' un problema...");
}
```

- L'eccezione sollevata può essere una stringa, un numero, un booleano o un oggetto

Metodi alternativi per la generazione di codice HTML

- Abbiamo visto che è possibile generare codice HTML dinamicamente accedendo al DOM del documento tramite i metodi `getElementById()`, `getElementsByTagName()`, `createElement()`, ...
- Questo metodo è molto generale e può essere usato con qualunque tag HTML
- Esistono altri approcci (di definizione meno recente) un po' più semplici che possono essere usati in casi particolari
  - La generazione del codice tramite `document.writeln()`
  - L'accesso ad alcuni elementi della pagina (forms, immagini, link e ancore) tramite appositi array messi a disposizione dal browser

Generare codice HTML tramite document.writeln() (1)

- Un modo semplice per generare codice HTML dinamicamente è tramite il metodo `document.writeln()` usando come parametro un'opportuna combinazione di stringhe e variabili
- Esempio: generazione di un form dinamico

```
var n = parseInt(prompt("quanti valori vuoi inserire?"));
document.writeln("<form>");
for (var i=0;i<n;i++) {
  document.writeln('<input type="text" name="v' +i+ '"><br>');
}
document.writeln("</form>");
```

- Attenzione:
  - Bisogna fare attenzione all'uso di virgolette e apici nelle stringhe
  - Il codice diventa difficile da leggere ed è facile fare errori
  - Questo approccio di solito NON SI PUO' USARE quando l'esecuzione del codice è legata a un evento (esempio: pressione di un bottone)
    - ★ Quando il browser finisce di caricare la pagina chiude lo stream di output che deve essere riaperto (cancellando il contenuto della finestra) per gestire chiamate a `document.writeln()` successive

Generare codice HTML tramite document.writeln() (2)

- Uso corretto di `document.writeln()`:

```
<html>
  <head> .... </head>
 <body>
  ....
  <script>
   ....
   document.writeln(stringa-html);
   ....
  </script>
  ....
 </body>
</html>
```

- Uso NON corretto di `document.writeln()`:

```
<html>
 <head> .... </head>
 <body>
  ....
  <input type="button" onclick="document.writeln(stringa-html)">
  ....
 </body>
</html>
```

Gli array images, forms, links e anchors (1)

- Al termine del caricamento del documento HTML il browser inizializza i seguenti array con oggetti Element del DOM
  - `document.images` – oggetti relativi alle immagini nel documento
  - `document.forms` – oggetti relativi ai form nel documento
  - `document.links` – oggetti relativi ai link nel documento
  - `document.anchors` – oggetti relativi alle ancore (destinazioni di link interni) nel documento
- Gli oggetti negli array sono disposti in ordine di apparizione dei corrispondenti elementi
  - Esempio: `document.images[3]` è l'oggetto Element della quarta immagine nel documento
- Gli attributi degli elementi possono essere acceduti (anche in scrittura) come proprietà dell'oggetto Element corrispondente
  - Esempio: `document.images[3].src="logo.png"` assegna un nuovo valore all'attributo `src` della quarta immagine nel documento

Gli array images, forms, links e anchors (2)

- Gli elementi degli array `document.images`, `document.forms`, `document.links` e `document.anchors` possono essere acceduti tramite la sintassi degli array associativi
- La chiave da usare è il valore dell'attributo `name` dell'elemento da accedere
  - Esempio: `document.images["logo"]` è l'oggetto Element dell'immagine nel documento che include tra gli attributi `name="logo"`
- Quando l'elemento specifica un `name` si possono usare anche le seguenti notazioni abbreviate:
  - Esempio: `document.images.logo` è equivalente a `document.images["logo"]`
  - Esempio: `document.logo` è equivalente a `document.images["logo"]`
- Il vantaggio della sintassi degli array associativi è che si può usare come indice una variabile o un'espressione
  - Esempio: `document.images["a" + i]`

Gli array images, forms, links e anchors (3)

- Tramite l'array `document.forms` è possibile accedere anche a tutti i campi (caselle di testo, ecc...) dei form tramite l'array `elements`
  - Esempio: `document.forms[0].elements[0]` è l'oggetto di tipo Element del primo campo contenuto in nel primo form
- Usando l'attributo `name` nei tag `<form>` e `<input>` è possibile usare le seguenti sintassi semplificate:
  - Esempio: `document.dati.elements["cognome"]` permette di accedere all'elemento di input in cui vale `nome="cognome"` del form in cui vale `nome="dati"`
  - Esempio: `document.dati.cognome` è equivalente a `document.dati.elements["cognome"]`

Gli array images, forms, links e anchors (4)

- Con le funzionalità fornite da `document.forms` è quindi possibile:
  - leggere o modificare il valore inserito da un utente in una casella di testo
    - Esempio: `document.dati.cognome.value`
  - verificare se una checkbox è stata selezionata (checked)
    - Esempio: `document.dati.automunito.checked` (restituisce `true/false`)
  - verificare il valore selezionato tramite una lista `radio`
    - Esempio: `document.dati.sesso.value` (restituisce il valore dell'elemento `radio` correntemente selezionato)
- E' possibile inoltre usare i metodi `submit()` e `reset()` dei form per simulare l'evento di pressione di bottoni di tipo `submit` e `reset`
  - Esempio: `document.dati.submit()` invia i dati del form `dati` a destinazione
  - Esempio: `document.dati.reset()` cancella il contenuto dei campi del form `dati`
- Questi metodi consentono di accedere e controllare il contenuto dei campi di un form prima di inviare le informazioni al destinatario

Le espressioni regolari (1)

- Per lavorare con le stringhe (ad esempio con i valori immessi da un utente in una casella di testo) è spesso utile usare le espressioni regolari
- Un'espressione regolare è un'espressione che descrive un pattern di caratteri
  - La definizione origninale si studia ad esempio nella teoria dei linguaggi regolari (riconosciuti da automi a stati finiti)
- Tutti i linguaggi di programmazione "moderni" forniscono metodi per lavorare sulle con le stringhe tramite una forma di espressioni regolari
- In JavaScript un'espressione regolare è rappresentata da un oggetto della classe `RegExp`

Le espressioni regolari (2)

- Le espressioni regolari consentono di realizzare
  - una forma avanzata di pattern matching: verificare se (e quante volte) un pattern è presente in una stringa
  - operazioni di ricerca e sostituzione di un pattern con un'altro all'interno di una stringa
- Un'espressione regolare si definisce in uno dei seguenti modi:
  - `var patt = new RegExp(pattern,modifiers);`
  - `var patt = /pattern/modifiers;`

  dove `pattern` descrive il pattern vero e proprio e `modifiers` descrive delle proprietà del pattern (ad esempio se deve essere case-sensitive oppure no)
- Ad esempio:
  - L'espressione `/ci/gi` fa "match" con:
  - `Cinquantacinque cincillà`

Le espressioni regolari (3)

Il pattern di una espressione regolare è definito dalla seguente sintassi (non completa):

- `abcd` – una sequenza di caratteri fa match esattamente con se stessa
  - `/ac/` fa match con `dddabacabacddd`
- `[aeiou]` – fa match con un singolo carattere tra quelli indicati (si possono usare notazioni abbreviate es. `[A-Z]`, `[0-9]`, ecc...)
  - `/[abc]/` fa match con `dddabacabacddd`
- `[^aeiou]` – fa match con un singolo carattere che non sia tra quelli indicati (eventualmente con notazioni abbreviate come sopra)
  - `/[^abc]/` fa match con `dddabacabacddd`
- `.` – fa match con un qualuque (singolo) carattere
  - `/a.a/` fa match con `dddabacabacddd`
- $\backslash s, \backslash d, \ldots$ – (metacaratteri) fanno match con caratteri speciali ($\backslash s$ con uno spazio bianco, $\backslash d$ con un numero, ...)

segue....

Le espressioni regolari (4)

- `pat1|pat2|...|patN` – fa match con stringhe che soddisfano uno tra i pattern indicati
- `pat+` – fa match con stringhe che soddisfino ripetutamente (almeno una volta) il pattern indicato
- `pat*` – fa match con stringhe che soddisfino ripetutamente (anche zero volte = stringa vuota) il pattern indicato
- `pat[N]` – fa match con stringhe che soddisfino N volte il pattern indicato
- `pat[N,M]` – fa match con stringhe che soddisfino da N a M volte il pattern indicato
- `pat?` – fa match con stringhe che soddisfino zero volte o una volta il pattern indicato
- `^pat` – fa match con una stringa che soddisfa il pattern indicato all'inizio della stringa considerata
- `pat$` – fa match con una stringa che soddisfa il pattern indicato al termine della stringa considerata

Le espressioni regolari (5)

Esempi:

```
var s = "dcd(c(dabacabacd)d)db";
p1 = /\(.*\)/;          // fa match con (c(dabacabacd)d)
p2 = /\([^\(\)]*\)/;    // fa match con (dabacabacd)
p3 = /a(b|c)/;          // fa match con ab
p4 = /(a(b|c))+/;       // fa match con abacabac
p5 = /(a(b|c)){3}/;     // fa match con abacab
p6 = /ad?b/;            // fa match con ab
p7 = /a.?a/;            // fa match con aba
p8 = /^d.d/;            // fa match con dcd
p9 = /d.d$/;            // fa match con dbd
```

Note:

- Quando si vogliono usare caratteri della sintassi dei pattern (es. `(`,`)`,`.`,`[`,`]`,...) come caratteri da matchare bisogna farli precedere da `\` (esempio `p1`)
- Di default il pattern matching è massimale: dalla posizione di partenza si fa match con la stringa più lunga possibile (esempio `p3`)

Le espressioni regolari (6)

- Il parametro `modifiers` di un pattern è una stringa che può contenere i seguenti caratteri:
  - `g` – indica che il pattern è globale: vanno cercate tutte le occorrenze con cui fa match in una data stringa
  - `i` – indica che il pattern non è case-sensitive
  - `m` – consente il pattern matching multilinea (ossia `^` e `$` fanno match ad ogni inizio e fine linea)
- Ad esempio:
  - L'espressione `/ci/gi` fa "match" con:
  - `Cinquantacinque cincillà`

Le espressioni regolari (7)

- Un oggetto `p` della classe `RegExp` offre un paio di metodi utili:
  - `p.test(s)` – verifica se l'espressone `p` fa match all'interno della stringa `s` e restituisce `true` o `false` di conseguenza
  - `p.exec(s)` – restituisce la sottostringa di `s` che fa match con `p` (attenzione, restituisce un array che ha il risultato atteso in posizione 0)
- Inoltre, anche il metodo `replace` delle stringhe può essere usato con le espressioni regolari
  - `s.replace(p,s1)` – restituisce la stringa ottenuta rimpiazzando i match di `p` nella stringa `s` con `s1`
  - Attenzione ai `modifiers`:

```
var s = var s = "dcd(c(dabacabacd)d)dbd";
document.writeln(s.replace(/a(b|c)/,"gg"));
document.writeln(s.replace(/a(b|c)/g,"gg"));
```

restituisce:

`dcd(c(dggacabacd)d)dbd`

`dcd(c(dgggggggggd)d)dbd`

Le espressioni regolari (8)

- Le espressioni regolari sono particolarmente comode per controllare la forma dei dati immessi da un utente in un form.
- Esempio:

```
function validateUsername(field) {
  if (field=="") return "Username non inserito";
  else if (filed.length<5)
    return "Lo username deve contenere almeno 5 caratteri";
  else if (/[^a-zA-Z0-9_-]/.test(field))
    return "I caratteri consentiti sono a-z, A-Z, 0-9, _ e -";
  return "";
}
```

- E' buona norma associare all'evento `onsubmit` di un form il codice JavaScript che verifica la buona formatezza dei valori inseriti
- Se i controlli di buona formatezza sono superati, il codice JavaScript invocherà il metodo `submit()` del form.