Estratto con modifiche dall'articolo "Debugging avanzato con Javascript"

Se usati bene i debugger Javascript possono aiutarvi **a trovare e risolvere gli errori presenti nel codice**. Per diventare esperti in questo tipo di operazione dovrete conoscere gli strumenti per il debug che avete a disposizione, il flusso di lavoro tipico nell'attività di debugging e i requisiti di codice per un debug efficace. In questo articolo discuteremo alcune tecniche avanzate per diagnosticare e risolvere i bug usando un'applicazione web di esempio.

I debugger

Le interfacce dei debugger Javascript stanno diventando più pulite, più standardizzate tra i vari prodotti e più facili da usare: ciò rende più semplice per i novizi e per gli utenti più esperti imparare le tecniche di debugging.

Al momento ci sono strumenti di debug disponibili per i principali browser:

- Firefox può contare sulla nota estensione Firebug
- IE8 offre dei Developer Tools integrati nel browser
- Opera (dalla versione 9.5) supporta il debugger Opera Dragonfly
- Safari può sfruttare sia il debugger Drosera JS, sia un tool per l'esplorazione del DOM chiamato WebInspector (nelle versioni più recenti del browser il debugger è stato integrato proprio nel Web Inspector)

Familiarizzate con più strumenti di debug, perché non saprete mai in quale browser si verificherà il prossimo errore. Dal momento, poi, che i vari tool sono tutto sommato comparabili per le funzionalità che offrono, è facile passare da uno all'altro una volta che si sappia come usarne uno.

Il flusso di lavoro del debug

Quando si investiga su un problema specifico, seguirete in genere questo procedimento:

1. Trovate il codice che vi interessa nel pannello per la vista del codice del debugger
2. Impostate uno o più **breakpoint** dove pensate che possano accadere cose interessanti
3. Eseguite lo script ricaricando la pagina nel caso di script inline o cliccando su un pulsante in caso lo script sia attivato da un gestore di eventi
4. Aspettate fin quando il debugger mette in pausa l'esecuzione rendendo possibile procedere nel codice **passo a passo**
5. Investigate i **valori delle variabili**. Per esempio, cercate le variabili che non sono definite quando invece dovrebbero contenere un valore o quelle che restituiscono "false" quando vi aspettate invece che restituiscano "true"
6. Se necessario usate la **linea di comando** per valutare il codice o modificare la variabile per fare dei test
7. Trovate il problema imparando quale pezzo di codice o quale input hanno causato le condizioni di errore

Requisiti per l'uso dei debugger

La maggior parte dei debugger richiede **codice ben formattato**. Gli script scritti su una sola riga rendono difficile la scoperta di errori nei debugger basati sulla lettura riga per riga del codice. Il codice offuscato può anch'esso essere difficile da debuggare, specialmente il codice che è stato compresso (packed) e ha bisogno di essere espanso usando `eval()`. Molte librerie Javascript vi consentono di scegliere tra versioni compresse/offuscate e ben formattate: queste versioni non compresse, seppure più 'pesanti' sono quelle idealmente da usare per il debug perché hanno codice ben formattato.

Una dimostrazione di debug

Iniziamo con un piccolo esempio che contiene appositamente bug, per imparare a diagnosticare e trattare i problemi nel codice Javascript. Il nostro esempio è rappresentato da una pagina web errata.

Lanciare i debugger

In Firefox dovrete essere sicuri di aver installato l'estensione Firebug e per iniziare selezionare dal menu **Tools (Strumenti) > Firebug > Open Firebug (Apri Firebug)**.

Nelle versioni recenti, le funzionalità base di debugger sono integrate: basta selezionare **da Strumenti → Console degli errori**

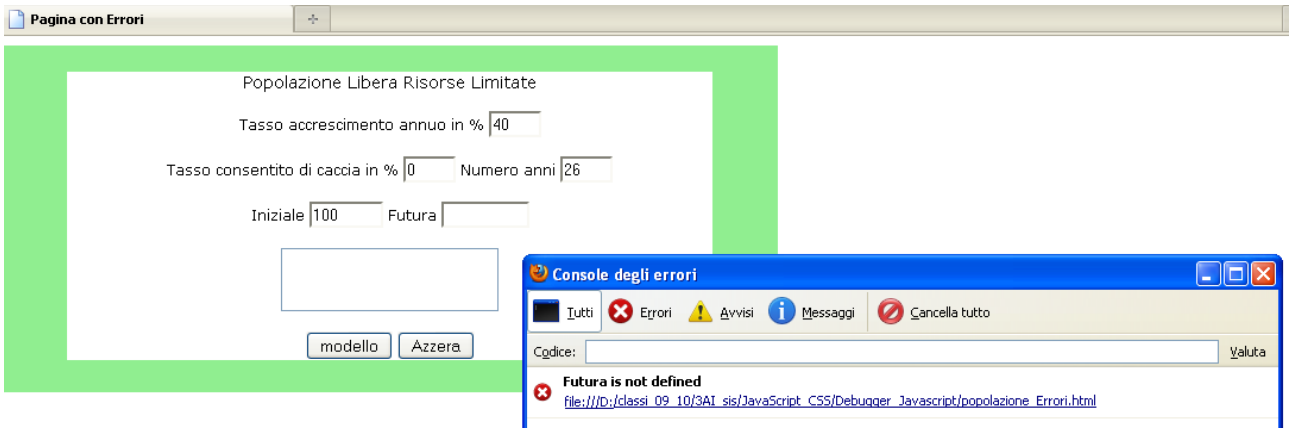
Su Opera 9.5 e successivi, scegliete dal menu **Tools (Strumenti) > Advanced (Avanzate) > Developer Tools (Strumenti per gli sviluppatori)**.

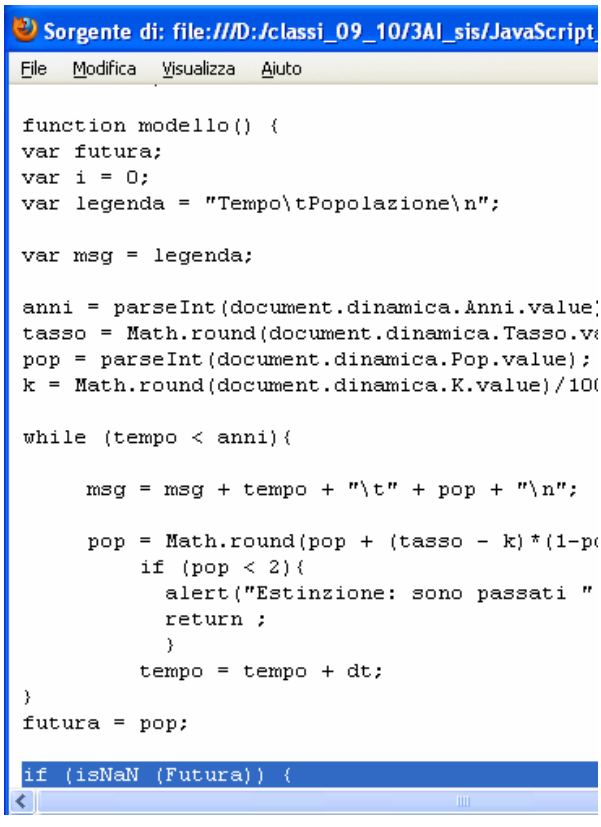
Su IE8 selezionate **Tools > Developer Tools**.

Su Safari o WebKit, prima abilitate il menu di debug se non è abilitato, poi selezionate **Debug (Sviluppo) > Show Web Inspector (Mostra...)** o nell'installazione in italiano **Sviluppo → Mostra console errori**

È il momento di attivare i debugger. Per seguire la demo, prestate attenzione alle istruzioni passo per passo che offriremo. Dal momento che alcune istruzioni richiedono modifiche sul codice, potete salvare la pagina della demo in locale e caricarla dal vostro disco prima di iniziare.

Bug **Futura is not defined**

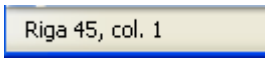




Cliccando sull'url che individual'errore
nella schermata precedente (funzionalità base del debugger integrato in Firefox)

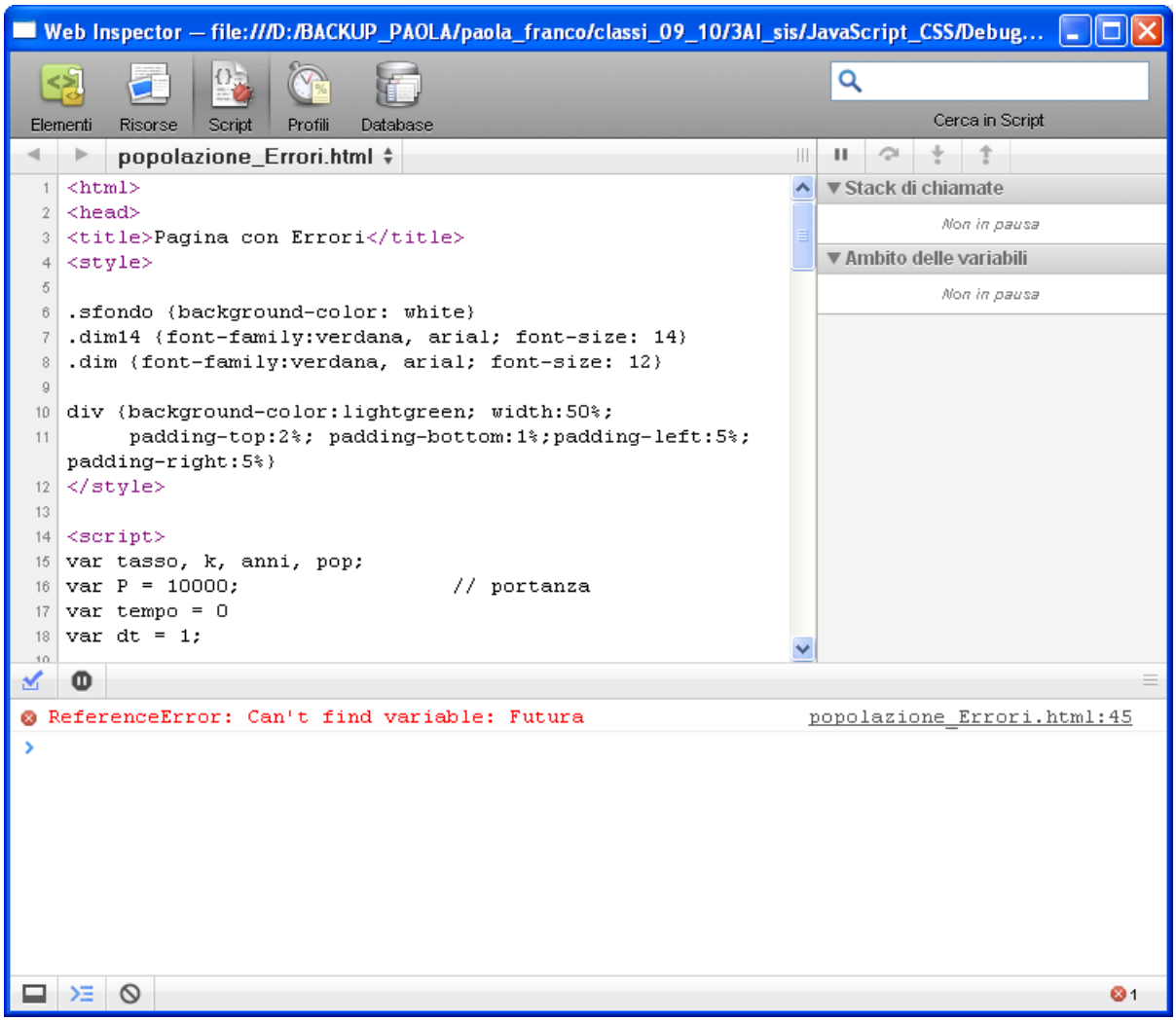
si apre il sorgente

con evidenza della riga di codice
che contiene
la variabile Futura non definita

ed è indicato il numero di riga e colonna 

Se date un'occhiata alle applicazioni di debug in WebInspector (integrato in Safari), Dragonfly (integrato in Opera), Firebug o in IE 8 , vi troverete inizialmente davanti a schermate (attivate spesso dalla tab 'Scripts') che consentono di scorrere il codice sorgente nel debugger.

Figura 1 – Schermata di WebInspector (in Safari)



Vogliamo inserire un **breakpoint** ad esempio all'interno di una funzione

Ecco come fare:

1. Cliccate sul numero di riga alla sinistra per impostare un breakpoint in corrispondenza della prima riga di codice all'interno della funzione
2. Ricaricate la pagina

Nota: il breakpoint deve essere impostato su una **riga con codice** che sarà eseguito quando la funzione viene attivata. La riga che contiene la funzione `nomeFunzione() {` è solo l'intestazione della funzione come pure le sole definizioni di variabili senza assegnamento di valore non sono da scegliere come breakpoint. Impostando un breakpoint in tali punti non si farà andare in pausa il debugger. Ricordate di eliminare i vecchi breakpoint: di solito cliccando di nuovo sul numero di riga

In Safari si può anche mettere in pausa con uso di opportuna icona 

Quando la pagina viene ricaricata, l'esecuzione dello script si blocca in corrispondenza del breakpoint

e vedrete qualcosa come quello mostrato in figura 2 nel caso di non aver impostato valori nei controlli del form :

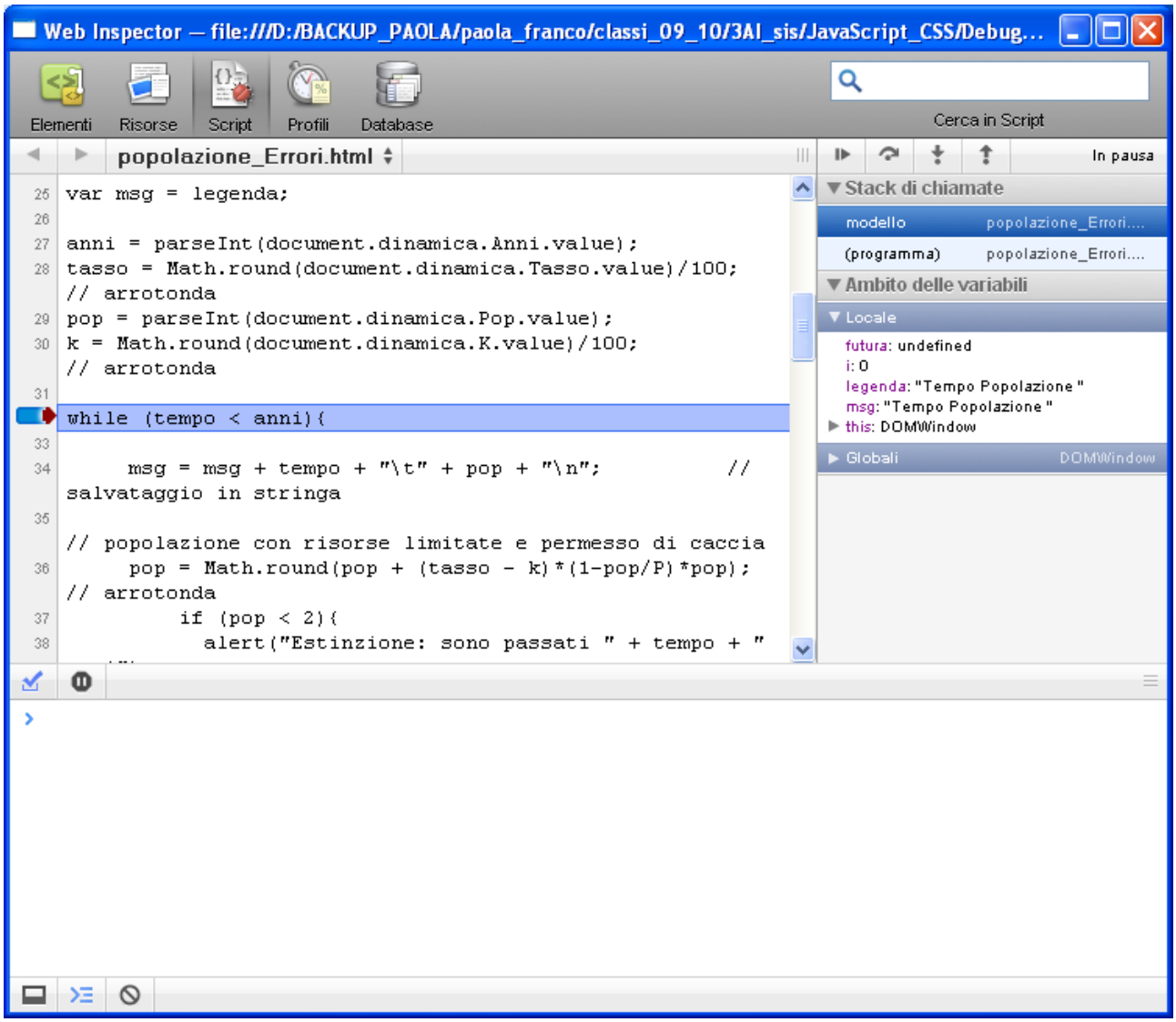


Figura 2 - Impostazione ed effetto del breakpoint in WebInspector

Procediamo **passo per passo** all'interno della funzione.

Incollate lo *snippet* di codice che vi interessa nella linea di comando per verifica.

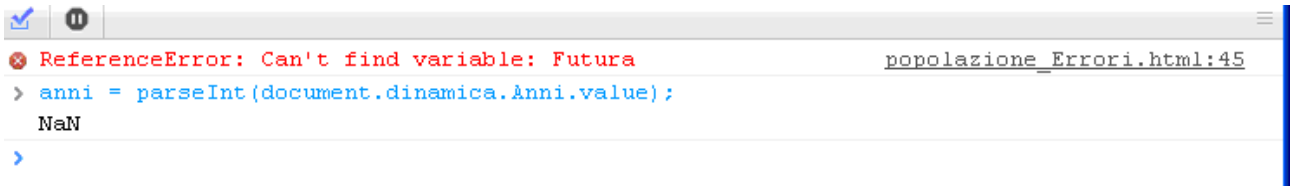


Figura 3 - in WebInspector output restituito dal comando

La figura mostra uno *screenshot* dopo la lettura di una variabile che legge valori da controlli del form (tali variabili sono dette `innerHTML` mentre con `outerHTML` si intendono variabili che scrivono valori nei controlli del form) dell'elemento restituito dal comando su cui state investigando:

Procediamo così nel test:

1. Trovate la linea di comando:
   Su Firebug selezionate la tab "Console"
   Nei Developer Tools di IE8 trovate la tab sulla destra con l'etichetta "Console"
   Su Dragonfly cercate sotto il pannello con il codice sorgente Javascript (potendo prevedere l'attivazione di arresti quando si incontra uno script gestendo con icona opportuna) 
2. Incollate l'istruzione da testare nel pannello della linea di comando
3. Premete Invio

Nel caso di aver impostato corretti valori nei controlli del form (in Safari):

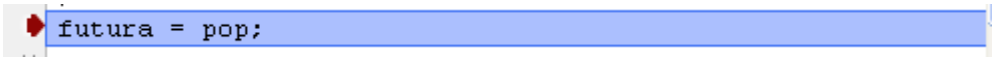
```
> anni = parseInt(document.dinamica.Anni.value);
  26
> pop = parseInt(document.dinamica.Pop.value);
  100
> k = Math.round(document.dinamica.K.value)/100;
  0
>
```

In Safari si possono sfruttare anche le opzioni di debug associate alle seguenti icone:

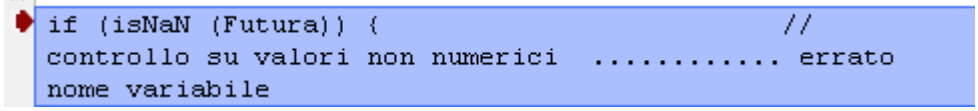
in realtà passa all'istruzione successiva : modalità **passo-passo**

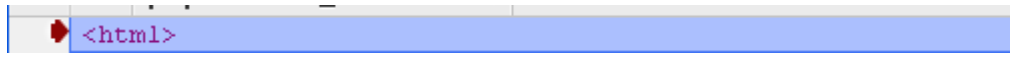
Ad esempio, attivandole in sequenza, si hanno i seguenti effetti – quindi il termine "chiamata di funzione" è da intendersi come corpo di un costrutto o addirittura singola istruzione :

Salta l'esecuzione del corpo
del costrutto while



passa all'istruzione successiva



con effetto 

Per capire cosa fa una funzione:

1. Usate il pulsante "step into" (*su Firebug è una freccia con la punta rivolta verso il basso, n.d.t*) per entrare nella funzione ad esempio `modello()`
2. Cliccate ancora ripetutamente sul pulsante "step into" per procedere nel codice una riga alla volta
3. Date un'occhiata all'overview delle variabili locali per vedere come cambiano mentre si procede passo a passo in questa funzione

Opzione di ricerca

Il WebInspector di Safari offre anche una potente funzionalità di ricerca. WebInspector vi consente di cercare in tutto il codice nello stesso tempo, compreso il markup, i CSS e Javascript. Il risultato viene mostrato in un pannello dedicato dovete fare doppio click sui risultati per saltare alla riga corrispondente, come mostrato nello screenshot.

Per trovare il punto che ha a che fare con la localizzazione dell'applicazione:

1. Scrivete Futura nel campo di ricerca
2. Impostate un breakpoint sulla riga in cui alla variabile è assegnato un valore
3. Ricaricate la pagina

Emulare le funzionalità *Watch*

Le applicazioni scritte bene mantengono al minimo il numero di variabili globali perché esse possono causare confusione quando differenti sezioni dell'applicazione provano a usare lo stesso nome di variabile.

La pratica nell'uso di variabili globali è sconsigliata, dunque, perchè può essere l'origine per conflitti e bug. Se dobbiamo verificare dove è impostata questa variabile e vedere se c'è un modo per renderla locale è molto utile la funzionalità "watch".

I debugger per molti altri linguaggi di programmazione adottano il concetto di "watch" che può interrompere il debugger quando una variabile cambia.

Né Dragonfly né Firebug supportano tale "watch" ma permettono di emularlo fornendo strumenti per ispezione con modifiche interattive

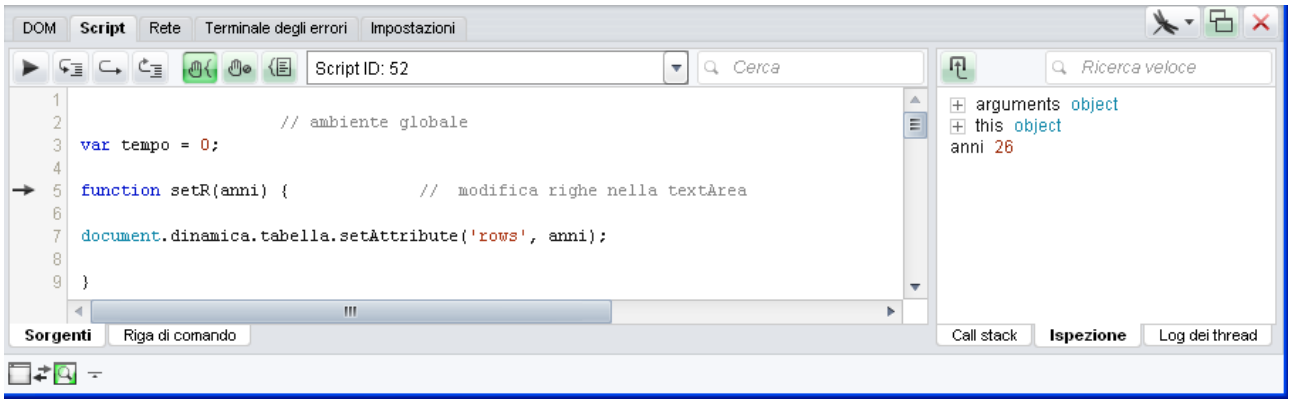


Figura 4 - in Dragonfly

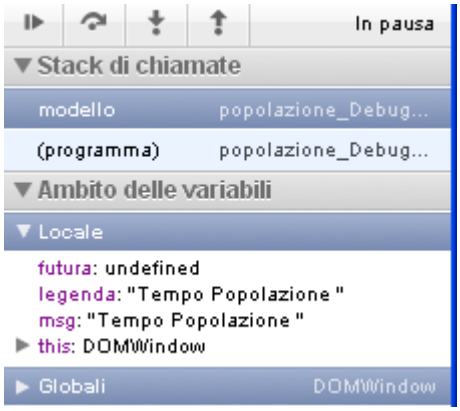
I Developer Tools di **IE8** hanno un pannello "watch", ma non è possibile interrompere l'esecuzione quando una variabile viene modificata. Considerato il supporto incompleto di IE8, non è possibile emulare la funzionalità nel modo in cui è possibile fare con Opera, Firefox e Safari.

La linea di comando

La linea di comando è uno strumento molto utile che consente di testare piccoli *snippet* di codice rapidamente.

L'integrazione della console di Tools di Debugger è anche molto utile dal canto suo: se il vostro comando restituisce come output un oggetto, avrete a disposizione una vista molto intelligente del tutto. Per esempio, otterrete una rappresentazione in forma simile a quella del markup se si tratta di un oggetto DOM.

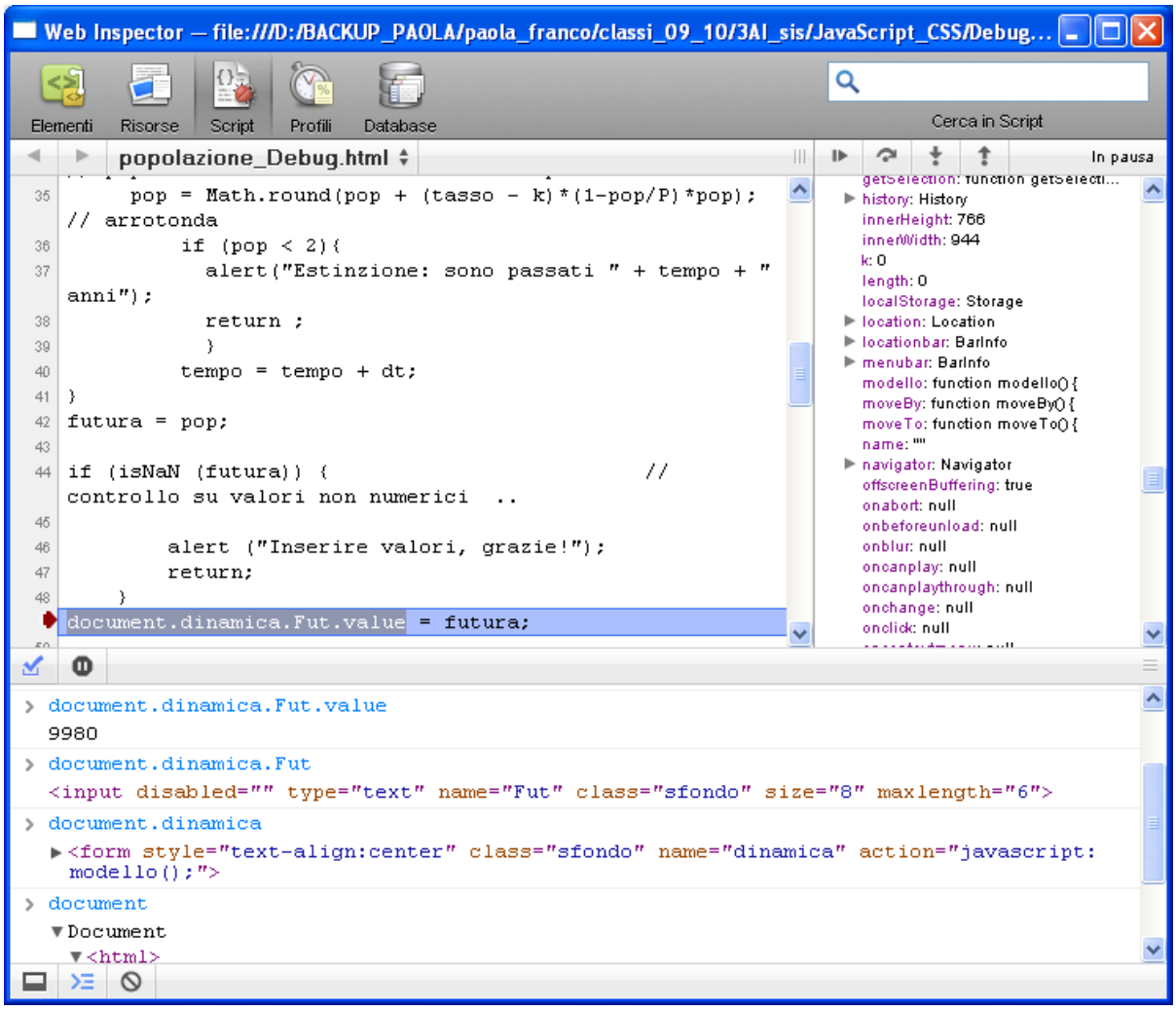
Potete usare la linea di comando per esplorare i problemi più in profondità.



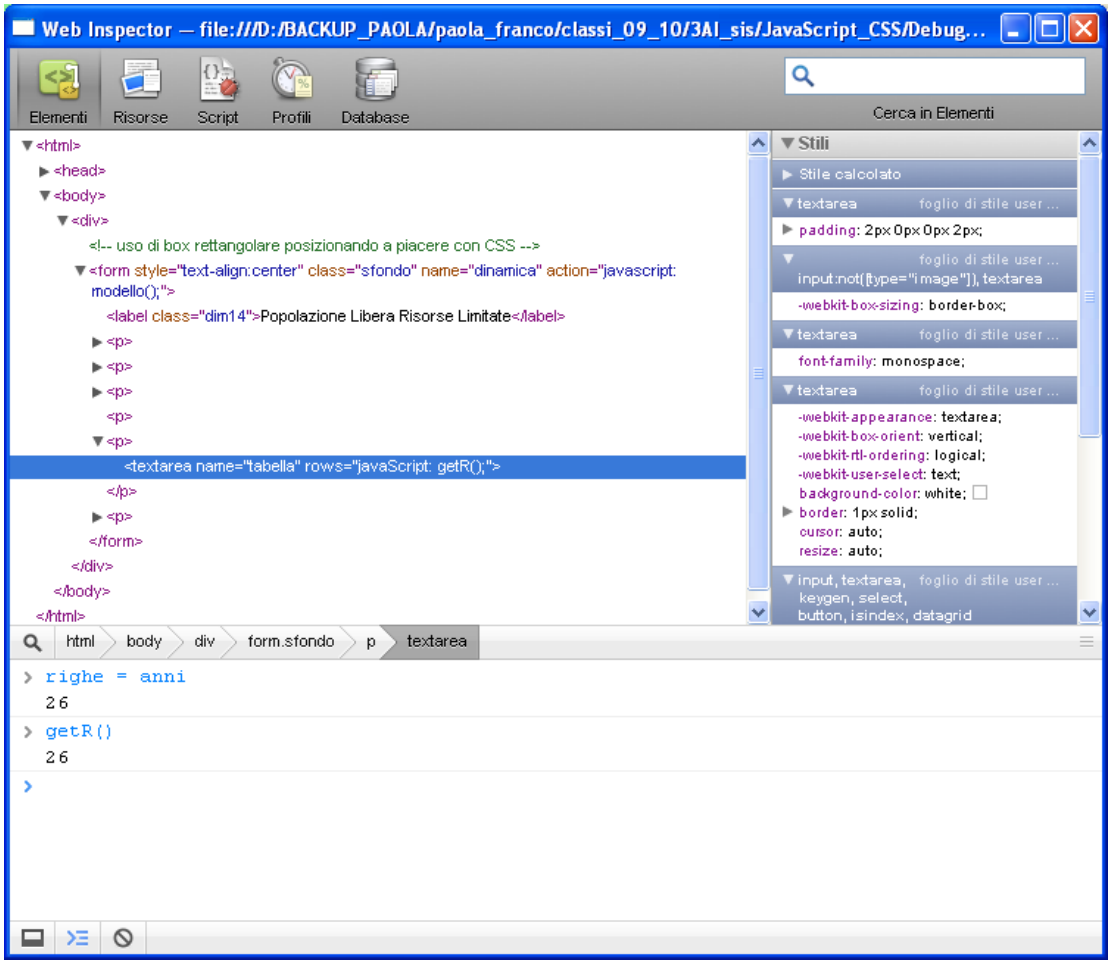
Oltre a conoscere il valore attuale delle variabili

```
> tasso
  0.4
```

Per esempio, potreste chiedervi qual è il valore corrente delle proprietà di elementi sia ispezionandoli con opzione Script



Sia con opzione Elementi, verificando la possibilità di modificare *dinamicamente* attributi dei tag:



Javascript è un linguaggio molto flessibile e uno dei suoi punti di forza (o di debolezza, dipende dai punti di vista) è che si possono sostituire funzioni core con quelle create da noi.

Funzionalità stack stack di chiamate

Quando l'esecuzione si interrompe al breakpoint, vorrete sapere dove si è verificata la chiamata. Ciò significa che dovete tornare indietro alla lista delle funzioni richiamate e scoprire come lo script è arrivato a quel punto. Per questo scopo usiamo la funzionalità stack.

Cliccando le funzioni precedenti nella lista degli stack potete tornare indietro a ritroso attraverso le chiamate e osservare come siete arrivati a quel punto.

È importante che proviate da voi per comprendere pienamente quanto è potente questa funzionalità. Notate che quando passate ad un'altra voce di stack il pannello delle **variabili locali** è aggiornato per mostrarvi lo stato corrente delle variabili nella funzione che effettua la chiamata.

Come usare lo stack delle chiamate per trovare una funzione problematica:

1. Cliccate la funzione che volete vedere nello stack delle chiamate
2. Notate che il pannello delle variabili locali viene aggiornato per mostrare le variabili locali per quella funzione
3. Ricordate che se usate i pulsanti di step essi vi porteranno avanti dall'ultima chiamata anche se state ispezionando altre parti dello stack

Riferimenti

“Debugging avanzato con Javascript”

di: Chris Mills e Hallvord R. M. Steen

http://javascript.html.it/articoli/leggi/3010/debugging-avanzato-con-javascript/

Ttraduzione dell'articolo Advanced Debugging with JavaScript di Chris Mills e Hallvord R. M. Steen pubblicato originariamente su A List Apart il 03 Febbraio 2009. La traduzione viene qui presentata con il consenso dell'editore e dell'autore

http://www.alistapart.com/d/advanceddebuggingwithjavascript/debugmeapp-ETM.htm

Conclusioni

Questo articolo dimostra le basi dell'uso dei debugger e alcune tecniche avanzate di debugging Javascript. Avete imparato come impostare breakpoint sia dai debugger sia con gli script, come procedere passo a passo nel codice, come usare l'interfaccia utente dei debugger, come impostare breakpoint avanzati e come integrare bookmarklet nelle tecniche di debugging.

Se avete avuto problemi a seguire la parte più avanzata di questo articolo, non preoccupatevi. Anche padroneggiare inizialmente le basi vi migliorerà come sviluppatori.