Alma Mater Studiorum
Università degli studi di Bologna

seconda facoltà di ingegneria

Corso di Laurea in Ingegneria Informatica

I LINGUAGGI PER LA PROGRAMMAZIONE WEB-BASED: L'ALTERNATIVA DART

Elaborata nel corso di
Sistemi Distribuiti

Relatore:
ANDREA OMICINI

Presentata da:
JENNIFER TESSARINI

I sessione
Anno Accademico 2011-2012

Alla mia famiglia
per aver creduto in me.
A Riccardo
che mi è sempre accanto.

Indice

Introduzione

Il Web ha subito numerose trasformazioni rispetto al passato. Si è passati da un Web statico, in cui l'unica possibilità era quella di leggere i contenuti della pagina, ad un Web dinamico e interattivo come quello dei social network.

Il Web moderno è, ancora oggi, un universo in espansione. La possibilità di arricchire le pagine con contenuti interattivi, video, foto e molto altro, rende l'esperienza web sempre più coinvolgente. Inoltre la diffusione sempre più ampia di mobile device ha reso necessario l'introduzione di nuovi strumenti per sfruttare al meglio le funzionalità di tali dispositivi. Tutto ciò grazie allo sviluppo delle tecnologie più svariate, frutto di un lavoro costante da parte di una web community aperta che ha permesso di definire tecnologie come HTML5 e CSS3.

Esistono al momento tantissimi linguaggi di scripting e di programmazione, ma anche CMS che offrono a chiunque la possibilità di scrivere e amministrare siti web. Nonostante le grandi potenzialità che offrono, spesso queste tecnologie si occupano di ambiti specifici e non permettono di creare sistemi omogenei che comprendano sia client che server.

Dart si inserisce proprio in questo contesto. Tale linguaggio dà a i programmatori la possibilità di poter sviluppare sia lato client sia lato server. L'obiettivo principale di questo linguaggio è infatti la risoluzione di alcune problematiche comuni a molti programmatori web. Importante in questo senso è il fatto di rendere strutturata la costruzione di programmi web attraverso l'uso di interfacce e classi. Fornisce inoltre un supporto per l'integra-

zione di svariate funzionalità che allo stato attuale sono gestite da differenti tecnologie.

L'obiettivo della presente tesi è quello di mettere a confronto Dart con alcune delle tecnologie più utilizzate al giorno d'oggi per la programmazione web-based.

Nel primo capitolo verrà descritta brevemente l'evoluzione del Web, dalla creazione di pagine statiche con il Web 1.0 a pagine dinamiche grazie al Web 2.0. Inoltre si parlerà di HTML, con particolare riferimento a XHTML e HTML5.

Nel secondo capitolo si introdurrà il nuovo linguaggio sviluppato da Google, Dart, descrivendone le principali caratteristiche e soffermandoci su quelle utili al confronto con altri linguaggi.

Il terzo capitolo sarà incentrato sul confronto più importante: Dart e JavaScript. Quest'ultimo infatti è il linguaggio di scripting più utilizzato per scrivere applicazioni web lato-client. Questo confronto verterà su tematiche importanti come il concetto di classe, la tipizzazione e l'interazione con gli elementi della pagina.

Il quarto capitolo vedrà il confronto fra Dart e due librerie JavaScript di largo impiego: jQuery e node.js. La prima parte verterà sul lato client di un'applicazione e quindi l'interazione con la pagina e la comunicazione tra client e server tramite chiamate JSON. La seconda parte invece sarà incentrata sul lato server, focalizzando l'attenzione sulla gestione delle chiamate di I/O e sulla costruzione di un primo Web Server.

Infine nel quinto ed ultimo capitolo si prenderà in considerazione un nuovo linguaggio derivato da JavaScript chiamato CoffeeScript. In questo capitolo saranno messi a confronto quest'ultimo e Dart in relazione soprattutto a funzioni ed oggetti.

Capitolo 1

Web e HTML

1.1 Nascita del Web

La data di nascita del *World Wide Web* [1] viene comunemente indicata nel 6 agosto 1991, giorno in cui l'informatico inglese Tim Berners-Lee pubblicò il primo sito web dando così vita al fenomeno "WWW" .

L'idea del *World Wide Web* era nata due anni prima, nel 1989, presso il CERN di Ginevra, il più importante laboratorio europeo di fisica. Alla sua base vi era il progetto dello stesso Berners-Lee di elaborare un software per la condivisione di documentazione scientifica con il fine di migliorare la comunicazione e quindi la cooperazione tra i ricercatori dell'istituto.

Parallelamente alla creazione del software, iniziò anche la definizione di standard e protocolli per scambiare documenti su reti di calcolatori, tra cui il linguaggio HTML e il protocollo di rete http.

Questi standard e protocolli supportavano inizialmente la sola gestione di pagine HTML statiche, costituite da file ipertestuali visualizzabili e navigabili utilizzando opportune applicazioni, chiamati browser.

Dopo i primi anni in cui era stato usato solo dalla comunità scientifica, il 30 aprile 1993 il CERN decise di mettere il WWW a disposizione del pubblico. La semplicità della tecnologia decretò un immediato successo e in poco tempo il WWW divenne la modalità più diffusa al mondo per inviare e

ricevere dati su Internet, facendo nascere quella che oggi è nota come l'"era del web".

1.2 Dal Web 1.0 al Web 2.0

Web 1.0 e Web 2.0 sono due termini nati nello stesso momento e utilizzati per descrivere l'evoluzione concettuale del *World Wide Web*. Essi sono concepiti per distinguere due modi diversi di vedere il web. La caratteristica principale del Web 1.0 è quella di permettere agli utenti di visualizzare solamente le pagine web, senza poter interagire con esse o modificarne il contenuto. Infatti le informazioni al loro interno non sono dinamiche e sono aggiornabili di tanto in tanto solo dal webmaster. Di fatto il Web 1.0 ha come scopo la condivisione e non la creazione.

Un passo in avanti è stato fatto grazie all'integrazione nelle pagine web di database e all'utilizzo di sistemi di gestione dei contenuti (CMS). Esso è stato da alcuni definito Web 1.5.

Con Web 2.0 si indica invece un insieme di applicazioni che permettono interattività fra i siti web e gli utenti. In altre parole è definito social web. Infatti sebbene dal punto di vista tecnologico molti strumenti della rete possano apparire invariati (come forum, chat e blog, che preesistevano già nel Web 1.0) è proprio la modalità di utilizzo della rete ad aprire prospettive in cui l'utente diventa parte centrale del web e ha la possibilità sia di creare che di modificare contenuti multimediali.

È stata perciò data la possibilità di accedere a servizi in grado di costruire siti web anche per l'utente meno esperto. Infatti se prima la costruzione di un sito web personale richiedeva la padronanza di elementi di HTML e di programmazione, oggi con i blog chiunque è in grado di pubblicare i propri contenuti, personalizzandoli a proprio gusto, senza possedere alcuna particolare preparazione tecnica.

Rivoluzionaria è anche la tecnologia Wiki che consiste in una pagina che viene aggiornata dai suoi stessi utilizzatori e i cui contenuti sono sviluppati

in collaborazione da tutti coloro che vi hanno accesso. Lo scopo anche qui è quello di condividere, ma anche di scambiare, immagazzinare e ottimizzare la conoscenza in modo collaborativo.

Tutto ciò è reso possibile dall'introduzione di nuovi strumenti: l'utilizzo di linguaggi di scripting come JavaScript, di elementi dinamici e dei fogli di stile (CSS) per gli aspetti grafici e dei CMS (sistemi di gestione dei contenuti). In particolare un CMS è uno strumento software installato su un server web studiato per facilitare la gestione dei contenuti di siti web. L'amministratore del CMS gestisce dalla propria postazione i contenuti da inserire o modificare tramite un pannello di interfaccia e controllo. Si può anche personalizzare l'aspetto esteriore delle pagine scegliendo un foglio di stile CSS appositamente progettato.

Grazie a questi strumenti e a tanti altri si possono creare delle vere e proprie applicazioni web che si discostano dal vecchio concetto di semplice ipertesto e che puntano a somigliare sempre di più ad applicazioni tradizionali per computer.

1.3 HTML

L'*HyperText Markup Language* (HTML) è un semplice linguaggio usato per i documenti ipertestuali disponibili nel *World Wide Web*. L'HTML è un linguaggio di pubblico dominio la cui sintassi è stabilita dal *World Wide Web Consortium* (W3C), e che è basato su un altro linguaggio avente scopi più generici, l'SGML [1] .

L'HTML non è un linguaggio di programmazione, in quanto non prevede meccanismi che consentono di prendere delle decisioni, non è in grado di compiere delle iterazioni e non ha altri costrutti propri della programmazione, ma è solamente un linguaggio di markup. Ciò significa che descrive le modalità di impaginazione, formattazione o visualizzazione grafica del contenuto

[1] Lo Standard Generalized Markup Language è un metalinguaggio avente come principale funzione la stesura di testi chiamati *Document Type Definition*

di una pagina web senza garantire però la garanzia che uno stesso documento venga visualizzato nello stesso modo su due dispositivi diversi. In generale un linguaggio di markup è un insieme di regole che descrivono i meccanismi di rappresentazione di un testo utilizzando convenzioni standardizzate, chiamati tag.

Al giorno d'oggi i documenti HTML sono in grado di incorporare molte tecnologie che permettono di aggiungere al documento controlli più sofisticati sulla grafica, interazioni dinamiche con l'utente, animazioni interattive e contenuti multimediali. Si tratta di linguaggi come CSS, JavaScript e jQuery, XML, JSON, o di altre applicazioni multimediali di animazione vettoriale o di streaming audio o video.

1.3.1 XHTML

XHTML è la riformulazione di HTML come applicazione XML. Ciò significa che un documento XHTML deve essere valido e ben formato. Ma cos'è XML? XML è una sorta di "super-linguaggio" che consente la creazione di nuovi linguaggi di marcatura. Esso è potente, flessibile e rigoroso ed è alla base di tutte le nuove specifiche tecnologiche rilasciate dal W3C. I principali obiettivi di XML, dichiarati nella prima specifica ufficiale (ottobre 1998), sono pochi ed espliciti:

- utilizzo del linguaggio su Internet;
- facilità di creazione dei documenti;
- supporto di più applicazioni;
- chiarezza e comprensibilità.

Nella visione di Tim Berners Lee XML è destinato ad essere il fondamento di un web finalmente universale.

Il motivo dell'introduzione di questo nuovo linguaggio è il fatto che si è preferito ridefinire le regole dell'HTML4 piuttosto che creare nuovi stan-

dard. In questo modo il vocabolario rimane uguale, ma cambiano le regole sintattiche. Gli obiettivi principali di XHTML sono:

- portare HTML nella famiglia XML con i benefici che ciò comporta in termini di estensibilità e rigore sintattico;
- mantenere la compatibilità con i software che supportano HTML4.

1.3.2 HTML5

HTML5 rappresenta un'evoluzione del modello di markup, che non solo si amplia per accogliere nuovi elementi, ma modifica in modo sensibile anche le basi della propria sintassi e le regole per la disposizione dei contenuti sulla pagina. A questo segue un potenziamento delle API JavaScript che vengono estese per supportare tutte le funzionalità di cui una applicazione moderna potrebbe aver bisogno:

- salvare informazioni sul device dell'utente;
- accedere all'applicazione anche in assenza di una connessione Web;
- comunicare in modo bidirezionale sia con il server sia con altre applicazioni;
- eseguire operazioni in background;
- pilotare flussi multimediali (video, audio);
- editare contenuti anche complessi, come documenti di testo;
- pilotare lo storico della navigazione;
- usare metafore di interazione tipiche di applicazioni desktop, come il drag and drop;
- generare grafica 2D o 3D in tempo reale;

- accedere e manipolare informazioni generate in tempo reale dall'utente attraverso sensori multimediali quali microfono e webcam.

Le novità introdotte dall'HTML5 rispetto all'HTML4 sono finalizzate soprattutto a migliorare il disaccoppiamento tra struttura, caratteristiche di resa e contenuti di una pagina web.

Semantica

Alcune delle nuove caratteristiche in HTML5 sono funzioni per inserire audio, video, elementi grafici, storage di dati lato client e documenti interattivi. HTLM5 introduce nuovi tag per identificare questi elementi, dando un valore semantico a queste parti della pagina. Alcuni dei nuovi tag sono: *<nav>*, *<header>*, *<footer>* , *<aside>*, *<section>* e *<figure>*. Un'altra importante modifica riguarda la dichiarazione di *Document Definition Type* che dovrebbe essere posta nella prima riga di una pagina Web ad indicare la grammatica, HTML per l'appunto, usata nel documento. Nell'HTML4 era questa:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/ strict.dtd">
```

Ma con l'arrivo dell'HTML5 si introduce una semplificazione e basta semplicemente scrivere l'istruzione:

```
<!DOCTYPE html>
```

Che si affianca a quella da utilizzare in caso si intenda scrivere una pagina XHTML5:

```
<xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
```

Offline application e LocalStorage

Una delle tante novità di HTML5 è la cosiddetta *application cache* o *offline web applications*. Attraverso questo strumento è possibile dire al browser

quali file salvare (non nella cache standard ma in un cache apposita) così da consentire una navigazione il più possibile completa di un sito anche quando si è offline Il vantaggio principale della *application cache* rispetto a quella tradizionale è il controllo. Infatti, mentre nella cache standard è il browser a decidere quali sono i file da salvare, con lo strumento offerto da HTML5 possiamo decidere con precisione quali risorse tenere in memoria. Oltre a questo, con l'*application cache* è possibile salvare anche script così da consentire un utilizzo totale delle funzionalità HTML5 anche offline. Un ulteriore vantaggio rispetto alla cache tradizionale, infine, è che l'*application cache* consente di salvare anche file che non si sono visitati mentre nella normale cache del browser una pagina per essere memorizzata deve essere stata visitata almeno una volta. Le *offline web application* sono uno strumento molto potente che, combinato con il *local storage*, consente di creare delle vere e proprie applicazioni indipendenti dalla effettiva connessione ad Internet o meno.

Il *local storage* è lo strumento di HTML5 attraverso il quale le pagine web possono salvare i dati localmente attraverso il browser dell'utente. Rispetto ai cookies il *localStorage* è la soluzione ideale per progetti in cui abbiamo bisogno di salvare lo stato di un'applicazione e i dati trasmessi sono piuttosto pesanti (il limite è di 5 MB). Per i cookies, invece, il limite è molto più basso e, soprattutto, essendo parte del protocollo HTTP i dati contenuti nei cookies vengono inviati ad ogni richiesta, creando notevole traffico. È possibile in tal modo salvare un'enorme quantità di dati, senza compromettere le performance del sito. Lo svantaggio principale del *localStorage* è che non esiste un modo di specificare la scadenza dei dati, che dunque deve essere gestita autonomamente. Oltre a questo, un altro aspetto da tenere in considerazione è che tutti i dati vengono salvati come stringhe, quindi quando ci serve ad esempio un numero è necessario prima trasformarlo nel tipo utile al nostro scopo.

API specifiche per mobile device

Per sfruttare al massimo le funzionalità di tutti i device mobili, HTML5 prevede l'utilizzo di una serie di API per accedere a caratteristiche e dati specifici dei vari terminali. Importante è la funzione di geolocalizzazione, attualmente è già implementata e funzionante sulla maggior parte dei browser per sistemi mobile. Attraverso questo meccanismo il device comunica al browser la posizione dell'utente. La funzione per accedere a questi dati è:

```
navigator.geolocation.getCurrentPosition(
   gotPosition,
   errorGettingPosition,
   {'enableHighAccuracy':true,'timeout':10000,'maximumAge':0}
);
```

Qui *gotPosition* è la funzione di callback da chiamare quando i dati di geolocalizzazione vengono reperiti, mentre *errorGettingPosition* è la funzione che viene eseguita quando si verifica un errore.

Sono comunque in fase di progettazione API avanzate anche per l'utilizzo, ad esempio, della fotocamera, della lista contatti o dei dispositivi audio.

WebSockets

Uno dei problemi più ardui che ogni sviluppare affronta è far comunicare il browser con il server senza dover ogni volta ricaricare la pagina, aprendo cioè una connessione diretta e aggiornabile a prescindere dal resto della pagina. *WebSockets* risolve questo problema. Questa tecnologia è costituita da un set di API nato apposta per permettere ai browser e server di comunicare in maniera asincrona e senza bisogno dell'interazione dell'utente. Infatti consente di creare un canale di comunicazione *full-duplex* tra il browser e il server, dando così la possibilità di comunicare in maniera semplice e user-friendly. In tal modo è possibile realizzare ad esempio una chat in real time.

WebWorkers

HTML5 cerca di risolvere anche i problemi relativi ad applicazioni sempre più complesse, che spesso rallentano il browser, rendendo la navigazione nel resto della pagina o l'utilizzo di altre finestre quasi impossibile. Per fare ciò è stata inventata la tecnologia *Web Workers*, ovvero un sistema che permette di eseguire programmi in background, mantenendo quindi la pagina attiva e utilizzabile da parte dell'utente.

Multimedia

HTML5 ha introdotto i due tag <*audio*> e <*video*> per inserire in maniera facile, veloce e senza l'utilizzo di plugin esterni dei file audio o dei file video. Ciò che è veramente innovativo è il fatto che questi elementi sono parte del DOM e dunque possono essere liberamente modificati e gestiti da JavaScript, ad esempio in combinazione con altri oggetti HTML5 tipo il canvas per manipolare in tempo reale un video.

Capitolo 2

Dart

2.1 Introduzione

Dart è un linguaggio di programmazione Web open source sviluppato da Google. È stato presentato alla conferenza "Goto Aarhus 2011".

L'obiettivo principale di Dart è quello di sostituire JavaScript come lingua franca dello sviluppo web su una piattaforma web. Dart è intenzionato a risolvere i problemi di JavaScript offrendo al tempo stesso migliori prestazioni, la possibilità di sviluppare più facilmente strumenti utili alla gestione di progetti di grandi dimensioni e migliori funzionalità legate alla sicurezza. Gli obiettivi principali sono:

- creare un linguaggio di programmazione strutturato flessibile per il web.
- rendere Dart familiare e semplice ai programmatori e quindi facile da imparare.
- assicurarsi che tutti i costrutti permettano alte performance e un veloce startup dell'applicazione.
- rendere Dart appropriato per tutti i tipi di device sul web.
- fornire tools che rendano Dart veloce su tutti i maggiori browser moderni.

Dart si propone di risolvere diversi problemi relativi alle attuali applicazioni web. Questi sono:

- Piccoli script evolvono in grandi applicazioni senza una struttura ben definita, cosa che rende difficile il debug e la manutenzione dell'applicazione stessa. Inoltre queste enormi applicazioni non possono essere suddivise per poter essere modificate da team diversi nello stesso momento.

- I linguaggi di scripting sono molto leggeri e permettono di scrivere codice molto velocemente. I collegamenti fra le diverse parti dell'applicazione non sono incluse nella struttura stessa ma nei commenti. Per qualcuno diverso dall'autore, tutto ciò rende più difficile la lettura e la manutenzione di una parte di codice.

- Al momento lo sviluppatore deve scegliere fra linguaggi statici o dinamici. I linguaggi statici tradizionali richiedono una serie di tools molto pesanti e di uno stile di programmazione inflessibile e troppo limitato.

- Gli sviluppatori non possono creare sistemi omogenei che comprendano entrambi client e server ad eccezione di alcuni casi.

- Diversi linguaggi implicano cambi di contesto scomodi e aggiungono complessità al processo di codifica.

Dart nasce quindi con l'intento di aiutare gli sviluppatori a scrivere applicazioni complesse ma allo stesso tempo affidabili per il web moderno. Gli obiettivi sono quelli di semplicità, efficienza e scalabilità. Per ottenere ciò, Dart combina la potenza di nuove caratteristiche con costrutti familiari in una sintassi chiara e leggibile.

Linguaggio Class-Based

Dart è un linguaggio Object-Oriented e quindi supporta il concetto di oggetto. Proprio come nei linguaggi OO come Java, Dart supporta anche il

concetto di ereditarietà. Questo può essere realizzato tramite l'uso di classi e interfacce. Infatti Dart è un linguaggio di tipo Class-Based. L'ereditarietà in Dart presenta però un limite, ovvero che non è possibile ereditare da più classi e quindi si dice che non supporta l'ereditarietà multipla. Importanti sono anche i concetti di interfaccia e classe astratta che contribuiscono a creare applicazioni riutilizzabili e ben strutturate [1].

Optional-Typed

Dart permette al programmatore un insieme di checking statico e dinamico. Il programmatore può scrive codice non tipizzato per creare semplici prototipi. Mentre l'applicazione cresce e diventa più stabile, i tipi possono essere aggiunti per permettere un debug e imporre la struttura che si desidera [2].

Modalità di esecuzione

I programmi Dart possono girare sia in modalità production sia in modalità checked. La modalità production è la predefinita di un programma Dart, ottimizzata per la rapidità d'esecuzione. In tale modalità i tipi statici vengono ignorati.

La modalità checked è adatta agli sviluppatori e aiuta a trovare gli errori di tipo a runtime. In generale è consigliato lo sviluppo ed il debug in modalità checked e il rilascio definitivo in modalità production. [3].

Function

Come la maggior parte dei linguaggi moderni, Dart utilizza le funzioni come *first-class* con chiusura completa e una sintassi semplice.

Dart inoltre supporta le funzioni con parametri opzionali, parametri named e valori di default per tali parametri. Le funzioni possono essere assegna-

[1] Per maggiori dettagli vedi [2]
[2] Per maggiori dettagli vedi [3]
[3] Per maggiori dettagli vedi [4]

te a variabili e passate come parametri di altre funzioni e possono supportare la chiusura lessicale, che permette un accesso alle variabili al di fuori del loro contesto.

In particolare il team Dart usa frequentemente le funzioni come handler degli eventi [4].

Isolates

Tutto il codice in Dart viene eseguito nel contesto di un *isolate*. Possono essere usati altri *isolate* per la programmazione concorrente e per eseguire codice di terze parti in modo più sicuro. Gli isolate all'interno di un'applicazione non condividono nessuno stato o memoria, ma hanno una propria memoria heap. Tuttavia comunicano attraverso scambio di messaggi che vengono mandati attraverso porte.

Nella VM standalone, la funzione *main()* viene eseguita nel primo isolate (conosciuto anche come *main isolate*). Quando tale isolate termina la propria esecuzione, termina l'intera VM, a prescindere dal fatto che altri isolate siano ancora in esecuzione [5].

2.2 Classi

Uno degli aspetti fondamentali della programmazione ad oggetti è l'utilizzo di costruttori. Questi permettono di creare le istanze delle classi e di inizializzarle durante il processo di creazione. La creazione di una nuova istanza avviene esattamente come in Java, ovvero in questo modo:

```
Person pers = new Person();
```

Se all'interno della classe non è stato dichiarato nessun costruttore, verrà richiamato un costruttore di default, senza argomenti. Per cui se una sottoclasse non ha un costruttore, verrà chiamato il costruttore di default della superclasse. Per esempio:

[4] Per maggiori dettagli vedi [2]

[5] Per maggiori dettagli vedi dart:isolate library [4]

```
class Person {
  Person.fromJson(Map data) {
    print('in Person');
  }
}

class Employee extends Person {
  // Person does not have a default constructor;
  // you must call super.fromJson(data).
  Employee.fromJson(Map data) : super.fromJson(data) {
    print('in Employee');
  }
}

main() {
  var emp = new Employee.fromJson({});

  // Prints:
  // in Person
  // in Employee
}
```

Vi sono in Dart due tipi speciali di costruttori che permettono di risolvere alcuni problemi:

- *named constructor*
- *factory constructor*

Named Constructors

Come la maggior parte dei linguaggi tipizzati dinamicamente, Dart non supporta l'overloading. Con i metodi, questa non è una grande limitazione

poiché si possono sempre utilizzare nomi diversi, ma con i costruttori questo non è possibile. Per ovviare a questo problema Dart definisce i named constructors. In tal modo quindi è possibile dichiarare più costruttori per una stessa classe.

```
class Point {
  num x, y;

  // Named constructor
  Point.fromJson(Map json) : x = json['x'], y = json['y'];

  Point(this.x, this.y);
}
```

Per creare nuove istanze di un *named constructor* si può fare in questo modo:

```
var jsonData = JSON.parse('{"x":1, "y":2}');
var point = new Point.fromJson(jsonData);
```

Ricordando che i costruttori non sono ereditati, ossia che un *named constructor* di una superclasse non è ereditato dalla sottoclasse, se si vuole che una sottoclasse sia creata con un *named constructor* definito nella superclasse, è necessario implementare tale costruttore nella sottoclasse.

Factory constructors

Quando si necessita di creare un'istanza di una classe, ma si vuole più flessibilità rispetto al semplice programmare una chiamata ad un costruttore per un tipo concreto di oggetto si usa la parola chiave factory. In tale modo si può implementare un costruttore che non sempre crea una nuova istanza di quella classe. Per esempio, un costruttore factory può restituire un'istanza da una cache o può ritornare un'istanza di una sottoclasse. Dart

supporta tutto ciò senza il bisogno di cambiare il modo di creare un oggetto. Infatti basta definire un costruttore*factory*. Quando viene richiamato sembra un normalissimo costruttore, ma l'implementazione è libera e si può fare qualsiasi cosa si voglia. Per esempio:

```
class Symbol {
  final String name;
  static Map<String, Symbol> _cache;

  factory Symbol(String name) {
    if (_cache == null) {
      _cache = {};
    }

    if (_cache.containsKey(name)) {
      return _cache[name];
    } else {
      final symbol = new Symbol._internal(name);
      _cache[name] = symbol;
      return symbol;
    }
  }

  Symbol._internal(this.name);
}
```

Questa classe definisce simboli. Un simbolo è come una stringa ma ci garantisce che esista un solo simbolo con quel dato nome in qualsiasi momento. Questo permette di comparare tranquillamente due simboli testando semplicemente se rappresentano lo stesso oggetto.

Il costruttore di default ha il prefisso factory. Di conseguenza, quando sarà invocato, non creerà nessun nuovo oggetto. Nell'esempio infatti si cerca un

simbolo precedentemente salvato nella cache con il nome dato e viene riusato se trovato. Il *caller* non si accorge di questo ed esegue semplicemente:

```
var a = new Symbol('something');
var b = new Symbol('something');
```

La seconda chiamata a *new* ritorna il precedente simbolo salvato. Questo è utile poiché significa che non è necessario un costruttore *factory* all'inizio, ma una volta creato può essere usato per evitare di modificare tutto il codice che chiama *new* invece di chiamare qualche metodo statico.

2.3 Interfacce

Una caratteristica interessante di Dart è che non si utilizza la sintassi esplicita per le interfacce. In Dart infatti si usano le interfacce implicite. Ciò significa che ogni volta che si definisce una classe si definisce automaticamente anche un'interfaccia, nella quale si descrivono i metodi pubblici della classe stessa. Ad esempio:

```
class Person {
  final _name;
  Person(this._name);
  void greet(who) => 'Hello $who, I am $_name.';
}
```

Data questa classe è possibile richiamare il metodo sottostante:

```
greetBob(Person person) => person.greet('bob');
```

Se pensiamo di creare una nuova classe, *Imposter*, che vogliamo passare a tale metodo senza che questa erediti da *Person*, possiamo usare le interfacce implicite:

```
class Imposter implements Person {
  void greet(who) => 'Hello $who, it is a pleasure to meet you.';
}
```

Infatti si può vedere come si usa il termina *implements* e non *extends*.

2.4 Tipi opzionali

Come detto in precedenza, in Dart è possibile scrivere programmi che non hanno nessuna annotazione di tipo e possono essere eseguiti senza problemi. Si può però decidere di aggiungere la notazione dei tipi al proprio programma con queste conseguenze:

- Aggiungere i tipi non impedisce al programma la compilazione e l'esecuzione, anche se l' annotazione è incompleta o sbagliata.

- Il programma avrà la stessa semantica indipendentemente dall' annotazione di tipo che si aggiunge.

Ci sono in ogni caso dei benefici derivanti dall'utilizzo dei tipi:

- È più facile per le persone leggere il codice se le annotazioni di tipo sono state disposte con giudizio;

- Gli errori sono facilmente individuabili. Dart mette a disposizione un checker statico che informa il programmatore su potenziali problemi, senza interferire con il lavoro.

- A volte i tipi possono aiutare a migliorare le performance quando si compila in JavaScript.

Quindi è una scelta del programmatore l'utilizzo o meno dei tipi. Se infatti non si è abituati ad utilizzare i tipi, possono essere tranquillamente non considerati. Non si riceveranno quindi warning e sarà possibile sviluppare nello stile di qualsiasi altro linguaggio con tipizzazione dinamica. In ogni caso le funzioni di libreria di Dart definiscono parametri di input e di output che sono soggetti a controlli da parte del *checker*. Eseguendo quindi in modalità *check*, se viene passato un parametro errato, sarà generato un errore.

Se invece si predilige l'uso dei tipi, possono essere utilizzati ovunque, proprio come in un linguaggio tipizzato dinamicamente anche se non si avrà lo stesso livello di controllo. Infatti le regole di Dart sono molto meno rigide. Approfondiremo ora ciò che avviene in modalità *checked*.

Checker Statico

Il *checker* statico di Dart lavora molto similmente a C. Questo avvisa di potenziali problemi in fase di compilazione. La maggior parte di questi warning sono associati ai tipi. Il *checker* però non produce errori, in quanto non interferisce sulla compilazione ed è possibile quindi eseguire il codice. Inoltre non riferisce ogni possibile violazione di tipo, in quanto non è un *typechecker*. Di seguito vediamo due esempi.

```
String s1 = '9';
String s2 = '1';
...
int n = s1 + s2;
print(n);
```

In questo caso il *checker* informa di un possibile errore, ma il codice continua ad essere eseguito, settando n a '91' e stampando appunto 91. Nel caso seguente:

```
Object lookup(String key) {...} // a lookup method
//in a heterogenous table
String s = lookup('Frankenstein');
```

Non viene generato alcun errore da parte del *checker*, in quanto ci sono buone possibilità che il codice sia corretto, nonostante la mancanza di informazioni. Il programmatore spesso sa quello che il *typechecker* non sa e quindi capisce che il valore salvato nella tabella sotto 'Frankenstein' è una stringa, nonostante il metodo dichiara di ritornare un *Object*.

I programmi Dart possono essere eseguiti in modalità *check* durante lo sviluppo. Se si esegue un programma in tale modalità, il sistema automaticamente esegue il controllo dei tipi quando vengono passati come parametri, quando sono variabili di ritorno e quando vengono eseguiti assegnamenti. Se fallisce il controllo, l'esecuzione termina in quel punto con un messaggio di errore. Per cui la seguente riga di codice blocca il controllo in quanto *Object* non è un sottotipo di *String*.

```
String s = new Object();
```

In ogni caso il seguente codice funziona correttamente, poichè l'oggetto attuale ritornato da *foo* in fase di esecuzione è di tipo *String*. Questo perché quando un oggetto è assegnato ad una variabile, Dart controlla a runtime che il tipo di oggetto sia un sottotipo del tipo di variabile dichiarato.

```
Object foo(){return "x";}
String s = foo();
```

Tutti questi controlli impongono una perdita di performance, in modo tale che non sia generalmente possibile eseguirli in fase di produzione. Il beneficio di questi controlli è il fatto che catturano errori di tipo dinamico nel punto in cui essi si trovano, rendendo più semplice il debug. Anche se la maggior parte di questi errori viene trovata durante la fase di testing, tuttavia la modalità *check* aiuta a localizzarli.

2.5 Funzioni

Dart possiede tre notazioni per creare le funzioni: una per le *named functions*, una per le *anonymous functions* ed una per le *arrow functions*. La forma *named* è la seguente:

```
void sayGreeting(String salutation, String name) {
  final greeting = '$salutation $name';
  print(greeting);
}
```

Tale forma assomiglia molto ad una normale definizione di funzione in C o ad un metodo in Java e Javascript. A differenza del C e del C++, queste possono essere innestate all'interno di un'altra funzione.

Se non si vuole dare un nome ad una funzione esiste una forma detta anonima. È simile alla dichiarazione precedente ma senza un nome esplicito o un tipo di ritorno:

```
window.on.click.add((event) {
  print('You clicked the window.');
})
```

Si può dire che le *anonymous functions* vengono frequentemente utilizzate per gli handler degli eventi e per i callbacks, mentre le named functions vengono utilizzate molto raramente.

Infine, se vi è la necessità di una funzione veramente semplice che deve solamente valutare un'espressione, esiste l'operatore =>:

```
  var items = [1, 2, 3, 4, 5];
  var odd = items.filter((i) => i % 2 == 1);
  print(odd); // [1, 3, 5]
```

In pratica, è preferibile utilizzare le *arrow functions* quando è possibile poiché sono molto concise ma anche facili da individuare grazie al simbolo =>.

Dart possiede un ulteriore asso nella manica: l'operatore =>può essere utilizzato anche per definire alcuni elementi. Infatti per definire un campo si può scrivere così:

```
class Rectangle {
  num width, height;
  bool contains(num x, num y) => (x < width) && (y < height);
  num area() => width * height;
}
```

Le *arrow functions* inoltre sono perfette per definire semplici *getters* o *setters*, per qualsiasi altro metodo composto da una singola riga di codice che restituisce un valore o per accedere alle proprietà di un oggetto.

In Dart è possibile utilizzare i parametri opzionali. Mettere un parametro di una funzione fra '[ ]' significa che quello è un parametro opzionale.

```
String say(String from, String msg, [String device]) {
```

```
  var result = "$from says $msg";
  if (device != null) {
    result = "$result with a $device";
  }
  return result;
}
```

In questo modo, richiamando la funzione, sarà possibile omettere quel parametro.

In ogni caso si possono avere valori di default per questi parametri. I valori di default devono essere delle costanti. Se non è fornito nessun valore, il valore sarà settato automaticamente a *null*.

```
String say(String from, String msg,
    [String device='carrier pigeon', String mood]) {
  var result = "$from says $msg";
  if (device != null) {
   result = "$result with a $device";
  }
  if (mood != null) {
    result = "$result (in a $mood mood)";
  }
  return result;
}
```

I parametri opzionali sono anche parametri *named*. Si può scrivere:

```
assert(say("Bob", "Howdy", device: "tin can and string") ==
    "Bob says Howdy with a tin can and string");


assert(say("Bob", "Howdy", mood: "fresh") ==
    "Bob says Howdy with a carrier pigeon (in a fresh mood)");
```

È possibile inoltre passare una funzione come parametro ad un'altra funzione. Per esempio:

```
var ages = [1,4,5,7,10,14,21];
var oddAges = ages.filter((i) => i % 2 == 1);
```

Si può anche assegnare una funzione ad una variabile, nel modo seguente:

```
var loudify = (msg) => '!!! ${msg.toUpperCase()} !!!';
assert(loudify('hello') ==
    '!!! HELLO !!!');
```

Infine le funzioni supportano la chiusura lessicale. Il seguente esempio mostra come la funzione *makeAdder* catturi la variabile n e la renda disponibile alla funzione che questa ritorna. Ovunque vada la funzione che ritorna, si ricorderà n.

```
Function makeAdder(num n) {
  return (num i) => n + i;
}

main() {
  var add2 = makeAdder(2);
  assert(add2(3) == 5);
}
```

2.6 Dart e HTML

Come tutti i linguaggi orientati al web, Dart ha la possibilità di essere integrato all'interno delle pagine HTML.

L'Html prevede un tag apposito per gli script ($<script>$). I tag script di HTML forniscono un attributo type per definire il linguaggio di scripting che si vuole adottare. Per Dart questo attributo ha il valore "application/dart". Come per altri script, il contenuto può essere inserito come corpo del tag script o specificato con un URL utilizzando l'attributo *src*.

Lo script Dart dovrà avere una funzione di primo livello *main()* o dichiarata direttamente nello script o in un file importato. Il browser invoca il

main() al caricamento. All'interno di uno script Dart vi è la possibilità di importare risorse dall'esterno. Questo è permesso utilizzando *#source* oppure *#import*. Gli script Dart permettono di includere script esterni utilizzando la direttiva *#source*, invece per utilizzare librerie esterne occorre usare la direttiva *#import* [6].

Un esempio dell'uso della direttiva *#source* è il seguente:

```
<html>
  <body>
    <script type='application/dart'>
      #source(Hello.dart)
      void main() {
        hello('Hello from Dart');
      }
    </script>
    <div id="message"></div>
  </body>
</html>
```

[6]Per maggiori dettagli vedi [5]

Capitolo 3

Dart e JavaScript

3.1 Introduzione a JavaScript

3.1.1 Caratteristiche generali di JavaScript

JavaScript [6] è un linguaggio di scripting Object-oriented. È il più utilizzato per quanto riguarda la scrittura di programmi web-based ed è, per questo motivo, il maggiore avversario di Dart.

La caratteristica principale di JavaScript è il fatto che il codice non debba essere compilato, ma interpretato da un programma ospite (ad esempio un browser) che fornisce API utili allo script per richiedere l'esecuzione di operazioni specifiche non incluse nei costrutti del linguaggio JavaScript in sè.

Il successo di JavaScript è dovuto al fatto che permette di scrivere funzioni integrate all'interno di pagine HTML che possano interagire quindi con il DOM del browser per compiere azioni dinamiche. Ad esempio con JavaScript è possibile:

- caricare nuovi contenuti sulle pagine o inviare dati al server tramite Ajax senza dover ricaricare la pagina;

- animare gli elementi di una pagina, farli apparire o scomparire, ridimensionarli, spostarli, ecc...;

- inserire contenuti interattivi, ad esempio giochi, oppure riprodurre audio o video;

- verificare la correttezza dei valori di input in un form web prima di inviarli al server;

- inviare informazione a diversi siti web riguardo le abitudini dell'utente e le attività del browser.

Inoltre, poiché JavaScript è l'unico linguaggio che i principali browser supportano, è diventato un linguaggio target per la compilazione in molti frameworks scritti in altri linguaggi.

3.1.2 Com'è nato JavaScript

Nel 1995 Netscape decise di dotare il proprio browser di un linguaggio di scripting che permettesse ai web designer di interagire con i diversi oggetti della pagina, ma soprattutto con le applet Java. Infatti in quello stesso anno Netscape era particolarmente vicina alla Sun Microsystems, con cui aveva stretto una partnership.

Brendan Eich venne incaricato del progetto e inventò LiveScript, chiamato così per indicarne la vivacità e dinamicità. Così le due aziende il 4 dicembre 1995 annunciarono la nascita di questo nuovo linguaggio, descrivendolo come "complementare all'HTML e a Java". La versione beta di Netscape Navigator 2.0 incorporava quindi LiveScript, ma Netscape decise di ribattezzare il nuovo linguaggio di scripting JavaScript.

La versione 2.0 di Netscape Navigator fu un grande successo, ma i web designer non utilizzarono JavaScript per interagire con le applet Java, ma piuttosto per rendere più vive le pagine.

Dato il successo di JavaScript, Microsoft sviluppò un linguaggio compatibile, chiamato JScript, supportato da Internet Explorer.

A causa di alcune differenze presenti in Internet Explorer 3, Netscape e Sun decisero di standardizzare JavaScript e si affidarono all'ECMA.

ECMAScript è dunque figlio di JavaScript e oggi quando si parla di JavaScript, JScript ed ECMAscript sostanzialmente si indicano tre varietà dello stesso linguaggio.

3.2 Il concetto di Classe

Una delle principali differenze strutturali tra JavaScript e Dart consiste nel concetto di classe.

3.2.1 JavaScript Prototype-Based

JavaScript, come detto in precedenza, è un linguaggio Object-based, ovvero basato sul paradigma ad oggetti. Ciò, però, non implica l'utilizzo di classi. Infatti JavaScript è un linguaggio cosiddetto Prototype-based, ovvero utilizza un meccanismo di ereditarietà basato sulla clonazione di oggetti già esistenti, detti prototipi. In questa tipologia, gli oggetti sono visti come pure strutture dati, contenenti proprietà e metodi utili alla manipolazione di questi stessi.

Per quanto riguarda la creazione di un oggetto è importante sottolineare il fatto che si debba definire una funzione. Infatti il compito che nei linguaggi Class-based è svolto dalle classi, nei linguaggi Object-based è svolto dalle funzioni.

Per creare un oggetto:

```
// costruttore
function Person(name){
   this.name = name;
};

Person.prototype.greet = function(){
   return 'Hello, ' + this.name;
}
```

```
// crea un Oggetto
var person = new Person(name);
```

Ogni funzione JavaScript possiede un attributo *prototype* che si riferisce ad un oggetto *prototype*. All'oggetto *prototype* è possibile aggiungere attributi e metodi, ma occorre ricordare che non è come tutti gli altri oggetti, in quanto la sua funzione è fare da modello per altri oggetti. Ogni attributo e metodo aggiunto ad un oggetto *prototype* viene reso disponibile a tutti gli oggetti della funzione costruttore a cui l'oggetto *prototype* è attribuito. Anche gli oggetti di sistema (come gli array) sono creati da funzioni costruttore che presentano un attributo *prototype*. Vediamo un esempio:

```
function Person(name){
   this.name = name;
}

Person.prototype.greet = function(){
   return 'Hello, ' + this.name;
}

function Employee(name, salary){
   Person.call(this, name);
   this.salary = salary;
}

Employee.prototype = new Person();
Employee.prototype.constructor = Employee;

Employee.prototype.grantRaise = function(percent) {
   this.salary = (this.salary * percent).toInt();
}
```

3.2.2 Dart Class-Based

Come detto in precedenza Dart, a differenza di JavaScript, è un linguaggio Class-based e prevede i concetti di classe, di interfaccia e ovviamente di oggetto.

Esempio di classe:

```
class Person{
   var name;
   Person(this.name);
}
```

Per creare un oggetto la sintassi è identica a quella di Javascript. L'ereditarietà è invece gestita in modo molto più semplice:

```
class Person(name){
   var name;
   Person(this.name);
   greet() => 'Hello, $name';
}

class Employee extends Person{
   var salary;
   Employee(name, this.salary) : super(name);
   grantRaise(percent) {
      salary = (salary * percent).toInt();
   }
}
```

3.3 I Tipi

3.3.1 JavaScript e la tipizzazione dinamica

Come nella maggioranza dei linguaggi di scripting, i tipi sono associati ai valori, non alle variabili. Per esempio una variabile 'x' può essere assegnata ad un numero, per poi essere riassegnata come stringa. JavaScript supporta diversi modi di testare il tipo di un oggetto. Per capire meglio il significato di tipizzazione dinamica prendiamo ad esempio il *Duck Typing*, utilizzato da Javascript. Questo termine si riferisce ad uno stile di tipizzazione dinamica dove la semantica di un oggetto è determinata dall'insieme corrente dei suoi metodi e delle sue proprietà anziché dal fatto di estendere una particolare classe o implementare una specifica interfaccia. Il concetto può essere sintetizzato in questa affermazione di James Whitcomb Riley: "Quando io vedo un uccello che cammina come un'anatra, nuota come un'anatra e starnazza come un'anatra, io chiamo quell'uccello 'anatra' ".

Nel *duck typing* si è interessati solo a quegli aspetti di un oggetto che vengono utilizzati, piuttosto che al tipo di oggetto stesso. Per esempio, in un linguaggio tipizzato staticamente, si può creare una funzione che prende un oggetto di un tipo *Duck* e chiamare su quell'oggetto due metodi, come ad esempio *walk* e *quake*. In un linguaggio tipizzato duck, la stessa funzione può prendere un oggetto di qualsiasi tipo e chiamare su questo oggetto i metodi *walk* e *quake*. Se l'oggetto non ha tali metodi allora verrà segnalato un errore di run-time. Se l'oggetto ha invece quei metodi, allora saranno eseguiti senza problemi.

Segue un esempio, scritto in JavaScript.

```
var Duck = function(){
   this.quack = function(){alert('Quaaaaaack!');};
   this.feathers = function(){
       alert('The duck has white and gray feathers.');
    };
   return this;
```

```
};

var Person = function(){
   this.quack = function(){alert('The person imitates a duck.');};
   this.feathers = function(){alert('The person takes a feather
                  from the ground and shows it.');};
   this.name = function(){alert('John Smith');};
   return this;
};

var in_the_forest = function(duck){
   duck.quack();
   duck.feathers();
};

var game = function(){
   var donald = new Duck;
   var john = new Person;
   in_the_forest(donald);
   in_the_forest(john);
};

game();
```

Il risultato in output sarà il seguente:

```
Quaaaaaak
The duck has white and gray feathers.
The person imitates a duck.
The person takes a feather from the ground and shows it.
```

3.3.2 Tipi primitivi in JavaScript

A questo punto possiamo definire i tipi primitivi presenti in JavaScript e le proprietà che hanno. Questi possono essere utilizzati per costruire altre strutture dati. Lo standard ECMAScript definisce sei tipi di dato:

- Numeri
- Stringhe
- Booleani
- *Null*
- *Undefined*
- *Object*
- Liste

Tutti questi tipi definiscono valori non modificabili ad eccezione degli *Object* e vengono detti tipi primitivi. In particolare, le stringhe sono immodificabili, come nella maggior parte dei linguaggi (basti pensare a Java).

Booleani

I booleani in Javascript possono assumere due valori: *true* e *false*. Ogni valore può essere convertito in booleano seguendo le seguenti regole:

1. False, 0, stringa vuota (“ ”), NaN, null e undefined diventano valori false.
2. In tutti gli altri casi il valore diventa true.

Per eseguire tale conversione esplicitamente si può usare la funzione *Boolean()*:

- Boolean(“ ”) = false

- Boolean(234) = true

In ogni caso è raramente necessario, in quanto Javascript esegue la conversione automaticamente quando si aspetta un booleano, come ad esempio nello statement di un if. Per quanto riguarda gli operatori logici sono supportati: && (AND), ||(OR), e ! (NOT) .

Numeri

In accordo con lo standard ECMAScript, in JavaScript esiste un solo tipo di numero che è "double-precision 64-bit binary format IEEE 754 value". Non esiste quindi il tipo intero. Infatti osserviamo questo esempio:

$$0.1 + 0.2 == 0.30000000000000004$$

In pratica, i valori interi sono considerati come interi a 32-bit, aspetto importante per le operazioni di bitwise. Sono supportate le classiche operazioni matematiche, incluse l'addizione, la sottrazione e il modulo. Esiste inoltre un oggetto chiamato *Math* che permette di sfruttare operazioni matematiche più complesse.

Si possono anche convertire stringhe in interi utilizzando la funzione *parseInt()*, che accetta in ingresso, oltre al valore, una base per la conversione. Se questa non è specificata si deve prestare attenzione alle conversioni poiché potrebbero esserci effetti non desiderati.

```
parseInt("010")
\\risulta 8
```

Allo stesso modo è possibile convertire numeri in virgola mobile usando la funzione *parseFloat()* che usa sempre la base dieci. Infine è possibile usare l'operatore '+' per convertire i valori in numeri. Per tutte queste funzioni e ogni qual volta che la stringa non è numerica, viene restituito un valore speciale chiamato NaN. Inoltre per rappresentare i numeri in virgola mobile esistono alcuni valori simbolici: '+Infinity', '-Infinity' e NaN (*not-a-number*).

Nonostante un numero sia spesso rappresentato solo dal suo valore, JavaScript offre alcuni operatori binari. Questi possono essere usati per rappresentare diversi valori booleani tramite un singolo numero usando il *bit masking*. In ogni caso è considerata una pratica troppo complessa e difficile da leggere e capire.

Stringhe

A differenza di C, le stringhe in JavaScript non sono modificabili, esattamente come in Java. Questo significa quindi che una volta create non possono essere più modificate ed eventualmente è necessario creare una nuova sfruttando operazioni sulla stringa originale. Le stringhe in JavaScript sono una sequenza di caratteri, in particolare Unicode, dove ogni carattere è rappresentato da un numero a 16-bit. Anche le stringhe, proprio per il fatto che hanno moltissime proprietà, sono oggetti.

Null e Undefined

Null è un oggetto di tipo object che indica espressamente un non valore, mentre *undefined* è un oggetto di tipo indefinito che indica un valore non inizializzato. È possibile in JavaScript dichiarare una variabile senza assegnarne il valore ed in questo caso il tipo è *undefined*.

Array: un caso particolare di Object

Gli array in JavaScript funzionano molto similmente a normalissimi oggetti, ma hanno una proprietà molto utile chiamata *lenght*. Questa proprietà rappresenta la lunghezza dell'array. Il vecchio modo di creare un array è il seguente:

```
var a = new Array();
a[0] = "dog";
a[1] = "cat";
a[2] = "hen";
```

```
a.length
\\risultato
3
```

Altrimenti può essere creato direttamente così:

```
var a = ["dog", "cat", "hen"];
a.length
\\risultato
3
```

Se si cerca di accedere ad un indice di un array non esistente si otterrà *undefined*. Gli array possiedono numerosi metodi, tra cui:

- concat: ritorna un nuovo array con gli elementi aggiunti.
- slice: ritorna una parte dell'array
- splice: permette di modificare un array eliminando una sezione e sostituendola con più elementi

3.3.3 Tipi primitivi in Dart

Il linguaggio Dart supporta i seguenti tipi:

- Stringhe
- Numeri
- Booleani
- Liste o meglio array
- *Literal*
- *Dynamic*

Stringhe

Una stringa in Dart, come in JavaScript, è una sequenza di caratteri Unicode. Per creare una stringa è possibile sia usare la singola (') che le doppie virgolette (''). Esempi di come creare una stringa:

```
var s1 = 'Single quotes work well for string literals.';
var s2 = "Double quotes work just as well.";
var s3 = 'It\'s easy to escape the string delimiter.';
var s4 = "It's even easier to just use the other string delimiter.";
```

É possibile mettere il valore di un'espressione dentro ad una stringa utilizzando il carattere $ {expression}. Se l'espressione è una variabile si possono ommettere le parentesi. Esempi:

```
var s = 'string interpolation';

assert('Dart has $s, which is very handy.' ==
       'Dart has string interpolation, which is very handy.');
assert('That deserves all caps. ${s.toUpperCase()} is very handy!' ==
       'That deserves all caps. STRING INTERPOLATION is very handy!');
```

Si possono semplicemente concatenare le stringhe usando stringhe di caratteri adiacenti, come nell'esempio.

```
var s = 'String ''concatenation'
        " works even over line breaks.";
```

Anche per Dart le stringhe sono oggetti immutabili. Infatti non esistono nelle API metodi che permettano di cambiare lo stato di un oggetto *String*. Per cui, come per JavaScript, è necessario creare una nuova stringa eseguendo operazioni sulla stringa originale. Vediamo un esempio:

```
var greetingTemplate = 'Hello, NAME!';
var greeting = greetingTemplate.replaceAll(new RegExp("NAME"), 'Bob');
assert(greeting != greetingTemplate); // greetingTemplate non e' cambiato
```

Numeri

A differenza di JavaScript esistono due tipi di numeri. Oltre ai *double* a 64-bit vi sono anche gli *Int* ovvero valori integer di lunghezza arbitraria. Entrambi questi tipi sono sottotipi di *num*. L'interfaccia *num* definisce le operazioni base come la somma, la sottrazione, la divisione e la moltiplicazione, così come le operazioni di bitwise. Grazie a questa interfaccia sono specificate anche le operazioni di bit shift e le operazioni logiche come AND e OR.

Sebbene questa interfaccia contenga moltissimi altri metodi per calcoli più complessi (come il valore assoluto o la radice quadrata), si possono utilizzare anche le funzioni della classe *Math* che risulta molto più completa. Per convertire una stringa in un numero esistono ulteriori funzioni, similmente a JavaScript. Di seguito alcuni esempi.

```
// String -> int
var one = Math.parseInt("1");
assert(one == 1);

// String -> double
var onePointOne = Math.parseDouble("1.1");
assert(onePointOne == 1.1);

// int -> String
var oneAsString = 1.toString();
assert(oneAsString == "1");

// double -> String
var piAsString = 3.14159.toStringAsFixed(2);
assert(piAsString == "3.14");
```

Booleani

Dart definisce un tipo formale di booleano, chiamato *bool*. Come in JavaScript questo oggetto può assumere solo due valori: *true* e *false*. Quando Dart si aspetta un valore booleano e questo valore non è *true*, allora sarà *false*. Tuttavia, diversamente da JavaScript, il valore '1' o l'oggetto non *null* non sono trattati come valori *true*. Facciamo un esempio:

```
var name = 'Bob';
if (name) {
   print("You have a name!"); // Stampa in JavaScript, non in Dart.
}
```

In JavaScript questo codice stampa "You have a name!" poiché *name* non è un oggetto vuoto. Tuttavia in Dart lo stesso codice non stampa nulla poiché *name* è convertito a *false* in quanto diverso da *true*.

Ecco un altro esempio di codice che permette di vedere la differenza tra JavaScript e Dart.

```
if (1) {
  print("JavaScript prints this line because it thinks 1 is true.");
} else {
  print("Dart prints this line because it thinks 1 is NOT true.");
}
```

Il modo di utilizzare i booleani da parte di Dart è quindi stato progettato per evitare strani comportamenti che possono verificarsi quando molti valori possono essere trattati come *true*. Ciò significa che, invece di usare gli *if-statement*, è possibile esplicitamente controllare i valori. Vediamo sotto degli esempi.

```
// Check for an empty string.
var fullName = ' ';
assert(fullName.isEmpty());
```

```
// Check for zero.
var hitPoints = 0;
assert(hitPoints <= 0);

// Check for null.
var unicorn;
assert(unicorn == null);

// Check for NaN.
var iMeantToDoThis = 0/0;
assert(iMeantToDoThis.isNaN());
```

Liste o array

In Dart gli array sono oggetti Liste, per cui vengono chiamate semplicemente *lists*. Quando si compila Dart in JavaScript, una lista viene compilata in un array. In ogni caso una lista in Dart è scritta esattamente come un array in JavaScript. Anche i campi per accedere alla lunghezza della lista e il modo in cui ci si riferisce ad un elemento della lista è uguale a JavaScript.

```
var list = [1,2,3];
assert(list.length == 3);
assert(list[1] == 2);
```

Per aggiungere un elemento invece si usa semplicemente il metodo *add()*, a differenza di JavaScript dove è necessario assegnare esplicitamente un valore alla fine dell'array. Per rimuovere un elemento, riducendo la lunghezza della lista, si usa il metodo *removeRange()*. Oltre a questi esistono moltissimi altri metodi tra cui:

- filter(): metodo che ritorna una nuova collezione con i soli elementi che soddisfano la condizione.

- every() e some(): metodi che controllano se una collezione soddisfa ogni condizione o solo una, rispettivamente.
- sort(): metodo che permette di dividere una lista secondo qualunque criterio.

Dynamic

In Dart esiste un tipo che viene assegnato di default quando non ne viene esplicitato uno dal programmatore. Questo si chiama *dynamic*. In questo modo si evita che il *checker* segnali errori di tipo.

Literal

Si può inizializzare un oggetto di qualsiasi di questi tipi speciali utilizzando *literal*. Per esempio, 'this is string' è una stringa *literal* e *true* è un *literal* booleano.

3.4 Interazione con il DOM

3.4.1 Cos'è il DOM

Il DOM (acronimo di *Document Object Model*) ,è un modello che descrive come i diversi oggetti di una pagina sono collegati tra loro. Il DOM quindi descrive la struttura di un documento HTML (e XML) e permette ai costruttori di pagine per il Web di poter accedere e manipolare tutti gli elementi della pagina stessa. È importante sottolineare che il DOM è indipendente dalla piattaforma, ovvero è un'interfaccia definita dal W3C per essere lo strumento universale per tutti i creatori di pagine Web. Questo significa che il DOM è indipendente sia dal tipo di browser, sia dalla versione, sia dal sistema operativo.

Il DOM non è un linguaggio di programmazione, ma senza di esso JavaScript, come altri linguaggi, non avrebbe nessun modello o nozione di pagina Web, pagina XML o di elementi che hanno a che fare con esse.

Ogni elemento in un documento è una parte del modello a oggetti, così che tali elementi possano essere acceduti e manipolati usando il DOM e un linguaggio di scripting come JavaScript.

Inizialmente JavaScript e il DOM erano strettamente interconnessi, ma successivamente si sono evoluti in due entità separate.

Browser differenti hanno diverse implementazioni del DOM e queste implementazioni presentano vari livelli di conformità con l'attuale standard, ma ogni browser usa un proprio modello a oggetti del documento per rendere le pagine accessibili allo script. A parte l'elemento <*script*> che viene definito da JavaScript, questo script JavaScript crea una funzione da eseguire quando il documento è caricato e quando l'intero DOM è disponibile per l'uso. Questa funzione crea un nuovo elemento H1, aggiunge testo a tale elemento e aggiunge H1 all'albero del documento.

```
<html>
  <head>
    <script>
     // esegue questa funzione quando il documento e' stato caricato
      window.onload = function() {
      // crea una coppia di elementi in una pagina HTML vuota
      heading = document.createElement("h1");
      heading_text = document.createTextNode("Big Head!");
      heading.appendChild(heading_text);
      document.body.appendChild(heading);
      }
    </script>
  </head>
  <body>
  </body>
</html>
```

Esistono moltissimi tipi differenti di dato che vengono passati alle funzioni. La tabella seguente descrive brevemente questi tipi di dato.

document	Quando un elemento ritorna un oggetto di tipo document, questo oggetto è l'oggetto riferito al documento principale.
element	element si riferisce ad un element o nodo di tipo element ritornato da un element delle API DOM. Piuttosto che dire che, ad esempio, il metodo *document.createElement()* ritorna un riferimento dell'oggetto ad un node, possiamo semplicemente dire che il metodo ritorna l'elemento che è stato appena creato nel DOM. Gli oggetti element implementano l'interfaccia *DOM Element* e soprattutto la più basilare interfaccia *Node*.
nodeList	Un nodeList è un array di elementi come quello ritornato dal metodo *document.getElementsByTagName()*. Gli oggetti in un nodeList sono acceduti tramite un indice in due modi equivalenti: • list.item(1); • list[1];
attribute	Quando un *attribute* è restituito da un *element*, esso è il riferimento ad un oggetto che mostra una particolare interfaccia per gli attributi. Gli attributi sono nodi nel DOM proprio come gli elementi, anche se sono poco usati.
namedNodeMap	Un *namedNodeMap* è simile ad un array, ma gli *item* sono acceduti tramite il nome o un indice, anche se quest'ultimo caso è puramente un vantaggio per l'enumerazione, dato che non hanno un particolare ordine nella lista. Un *namedNodeMap* ha un metodo *item()* adatto a questo scopo ed è possibile anche aggiungere o rimuovere un element dal *namedNodeMap*.

3.4.2 Eventi

Per rendere interattiva una pagina web occorre utilizzare il meccanismo degli eventi. Sarà quindi necessario scrivere script con un linguaggio *event-driven*. *DOM events* permette quindi una programmazione di tipo *event-driven* per registrare diversi event handler o listeners sui nodi degli elementi della pagina.

Nel caso della programmazione web un evento è un qualcosa che succede ad un elemento del DOM della pagina, come un click del mouse o un errore quando non si carica correttamenteun'immagine. Quando viene generato un evento, il browser invia un *event* al relativo elemento. Se è stato settato un *handler* a quest'ultimo, allora verrà chiamata una funzione. La combinazione fra l'uso del DOM e degli eventi permette di realizzare applicazioni dinamiche. Storicamente il modello ad eventi è stato utilizzato in modi diversi nei diversi web browser. Questo ha causato problemi di compatibilità e, per far fronte a ciò, il modello ad eventi è stato standardizzato dal W3C nel Level 2 del DOM. Esistono per questo motivi moltissimi tipi di eventi che possono essere generati. Gli HTML event racchiudono tre categorie di eventi:

- Common/ W3C events
- Microsoft-specific events
- Touch events

Common/W3C events

In questa categoria sono raggruppati gli eventi più sfruttati per la creazione di pagine web. Questi sono:

- Mouse events
- Keyboard events
- HTML frame/object events

- HTML form events
- User interface events
- Mutation events (notifica di ogni modifica nella struttura del documento)

Microsoft-specific events

Due importanti tipi di eventi sono stati aggiunti da Microsoft e in alcuni casi possono essere utilizzati solo su Internet Explorer (IE):

- *Clipboard events*
- *Data binding events*

Touch events

I browser web che vengono eseguiti sui moderni touch device generano eventi come *touchstart*, *touchend*, *touchenter*, *touchleave*, *touchmove* e *touchcancel*. Apple iOS e Google Android sono due esempi di sistemi operativi mobile che supportano questi eventi nei loro browser. Quando trascini un dito sullo schermo del device touch, si dà il via ad una *lifecycle* di *touch event* e sono innescati i seguenti eventi.

- *Touchstart*: quando si appoggia un dito sullo schermo.
- *Touchend*: quando si toglie il dito dallo schermo.
- *Touchmove*: quando si muove un dito sullo schermo.
- *Touchenter*: quando un punto touch entra nell'area interattiva definita da un elemento del DOM.
- *Touchleave*: quando un punto touch esce dall'area interattiva definita da un elemento del DOM.

3.4.3 Event Handler in JavaScript

Esistono diverse modalità per intercettare un evento (*event handling*). Esaminiamo le due modalità più utilizzate. La prima modalità consiste nell'associare ad un evento di uno specifico oggetto l'esecuzione di un particolare codice tramite JavaScript:

```
<input type="button" value="Hello" id="btnHello1" />
<script type="text/javascript">
document.getElementById("btnHello1").onclick = function(){
   alert("Hello World!");
}
</script>
```

In modo analogo possiamo associare ad un evento l'esecuzione di una specifica funzione:

```
<input type="button" value="Hello" id="btnHello2" />
<script type="text/javascript">

document.getElementById("btnHello2").onclick = HelloWorld;

function HelloWorld(){
    alert("Hello World!");
}

</script>
```

L'altra modalità è decisamente la migliore e si tratta di gestire gli eventi tramite l'*Object Model*. Presenta i seguenti vantaggi:

- può essere utilizzato con qualsiasi elemento del DOM e non solo con gli elementi HTML;
- fornisce una gestione migliore del codice eseguito in corrispondenza dello scatenarsi di un evento;

- consente di gestire più handler per lo stesso evento.

Lo standard W3C prevede l'uso della funzione:

```
target.addEventListener(type, listener, useCapture);
```

Qui *target* è l'oggetto di cui si desidera intercettare l'evento, *type* è la stringa che rappresenta il tipo di tale evento, *listener* è l'oggetto che riceve la notifica (deve implementare l'interfaccia *EventListener* oppure essere un puntatore ad una funzione JavaScript) e *useCapture* è un campo booleano utile per definire se tutti gli eventi del tipo specificato saranno gestiti dal listener indicato. In Microsoft Internet Explorer anziché *addEventListener()* si utilizza:

```
bSuccess = object.attachEvent(sEvent, fpNotify);
```

Qui *object* è l'oggetto di cui si desidera intercettare l'evento, *sEvent* è la stringa che rappresenta il tipo di tale evento, *fpNotify* è il puntatore alla funzione JavaScript che si desidera invocare e *bSuccess* è un valore di ritorno booleano che indica se l'evento è stato correttamente associato alla funzione specificata. In modo analogo è possibile rimuovere un *event handler* utilizzando *removeEventListener* (W3C Standard) oppure *detachEvent* (Internet Explorer).

3.4.4 Rivoluzione del DOM in Dart

Il cambiamento più semplice, a livello di DOM, è stato quello di eliminare nomi troppo laboriosi. In tal modo si permette al programmatore di usare ciò che si usa con più frequenza più facilmente.

Il primo vero cambiamento è stato quello di semplificare la ricerca degli elementi nel DOM. Infatti questo presenta una miriade di metodi per cercare oggetti, ma grazie all'introduzione di *query()* e *queryAll()* è possibile utilizzare queste funzioni per trovare qualsiasi cosa.

```
// Old:
  elem.getElementById('foo');
  elem.getElementsByTagName('div');
  elem.getElementsByName('foo');
  elem.getElementsByClassName('foo');

// New:
  elem.query('#foo');
  elem.queryAll('div');
  elem.queryAll('[name="foo"]');
  elem.queryAll('.foo');
```

In JavaScript i tipi di collezioni del DOM erano differenti dal tipo Array, cosa che rendeva tutto più difficile dato che vi erano discordanze tra i metodi a disposizione. Con Dart tutto ciò è stato risolto. I metodi come *elements*, *nodes* e *query()* che ritornano collezioni, lo fanno ritornando oggetti che implementano l'interfaccia *collection* di Dart stesso. Questi sono oggetti *lists*, *maps* e *sets*. Questi tipi implementati da Dart permettono di eliminare una serie di metodi specializzati. Infatti :

```
// Old:
  elem.hasAttribute('name');
  elem.getAttribute('name')
  elem.setAttribute('name', 'value');
  elem.removeAttribute('name');

// New:
  elem.attributes.contains('name');
  elem.attributes['name'];
  elem.attributes['name'] = 'value';
  elem.attributes.remove('name');
```

Per creare una nuova istanza di un tipo DOM era necessario trovare il metodo *factory* nel documento. Con Dart invece basta utilizzare i costruttori, come in qualsiasi linguaggio OO:

```
// Old:
  document.createElement('div');

// New:
  new Element.tag('div');
```

Il più grande cambiamento sta ovviamente nella gestione degli eventi. È stato reso più semplice il collegamento con gli *event handler*.

Come spiegato in precedenza, vi sono due modi per lavorare con gli eventi nel DOM. Dart si è occupato di eliminare la modalità che permetteva di settare le proprietà *on_* su un *Element* e ha creato una classe *ElementEvents*. Per ogni tipo di evento esiste una proprietà (*click*, *mouseDown*, ecc.) che è un oggetto *Event* che può aggiungere o rimuovere *listener* e gestire eventi. In questo modo tutti gli eventi relativi ad un elemento sono nascosti in una singola proprietà. Esempi:

```
// Old:
elem.addEventListener('click',
   (event) => print('click!'), false);

elem.removeEventListener( 'click', listener);

// New:
elem.on.click.add(
   (event) => print('click!'));

elem.on.click.remove(listener);
```

3.5 Integrazione HTML e DART: differenze rispetto a JS

L'integrazione del codice Dart differisce sotto certi aspetti da quella JavaScript. Ogni pagina HTML può avere più tag script Dart, ma ognuno di questi in una pagina viene eseguito isolatamente. Questo fondamentalmente differisce dal modo in cui JavaScript è integrato in HTML. In JavaScript la dichiarazione di più tag si trova nello stesso *namespace*. In Dart il codice all'interno di un tag script non può direttamente accedere al codice definito in un altro tag. Se uno script vuole caricare codice da un'URL differente, si devono usare *#source* o *#import*. Ogni tag script deve definire il proprio *main()* per essere eseguito.

Diversamente da JavaScript, i costrutti di Dart come interfacce, classi e funzioni sono dichiarative. Ogni applicazione Dart prevede un *main()* esplicito che viene invocato dal browser al momento dell'esecuzione. L'ordine di esecuzione non è garantito. Ci sono alcune conseguenze:

1. Tutti i tag script con il MIME *type* della pagina settato a "application/dart" possono essere caricati asincronicamente.

2. Il codice Dart è eseguito solo dopo che la pagina è stata analizzata. I programmatori Dart possono assumere che il DOM sia totalmente caricato.

Infine non è permesso l'utilizzo di *event listener* in questa forma:

```
<div onclick="foo()">Click this text.</div>
```

Con JavaScript i programmatori potevano inserire il codice degli *event listener* direttamente nei nodi HTML, anche se ciò è tipicamente sconsigliato nelle moderne applicazioni JavaScript.

Capitolo 4

Dart e jQuery/node.js

4.1 Dart e jQuery

Allo stato attuale jQuery è una tra le metodologie più utilizzate nello sviluppo di script lato client. Per questo motivo Dart si trova a dover competere con questo potente strumento di programmazione. In particolare jQuery e Dart possono essere messi a confronto su due tematiche fondamentali della programmazione web client-side:

- manipolazione del DOM

- interazione client-server

4.2 Introduzione a jQuery

jQuery [7] è una libreria di funzioni (framework) JavaScript, *cross-browser* per le applicazioni web, che si propone come obiettivo quello di semplificare la programmazione lato client delle pagine HTML.

Il motto di jQuery è "write less, do more". La sua sintassi infatti è stata progettata per rendere più semplice la navigazione di un documento, selezionare elementi del DOM, creare animazioni, manipolare gli stili CSS e gli elementi HTML, collegare eventi e sviluppare applicazioni Ajax. Il tutto

senza modificare nessuno degli oggetti nativi JavaScript. Le API di jQuery comprendono metodi per eseguire tutte queste operazioni, mantenendo la compatibilità tra browser diversi e standardizzando gli oggetti messi a disposizione dall'interprete JavaScript del browser.

Di seguito sono elencate le categorie di funzioni presenti nella libreria.

- Selettori: utilizzati per ottenere elementi della pagina, ad esempio in base al suo id, alla classe, ad attributi o contenuti.

- Attributi: ottenuti o modificati in maniera diversa a seconda del browser. In particolare jQuery aiuta lo sviluppatore offrendo un'unica funzione di frontend valida sia come *getter* che come *setter* .

- DOM Traversing: metodi e funzioni per attraversare e scorrere il DOM del documento.

- Manipolazione del DOM: aggiungere e rimuovere elementi alla pagina, sostituire elementi o eliminare tutti gli elementi contenuti in un certo nodo.

- CSS: cambiare, rimuovere o aggiungere proprietà grafiche di tutti gli elementi selezionati, ottenere e sostituire velocemente proprietà difficili da manipolare.

- Eventi: riconoscere oggetti di tipo *event* e provvedere a modificare le loro proprietà semplificando la loro gestione.

- Effetti: manipolare la visibilità degli elementi selezionati (effetto fading o sliding).

- Ajax: gestione delle chiamate asincrone semplificata e funzioni per caricare contenuti dinamicamente, eseguire richieste asincrone e interagire con JavaScript.

4.3 Manipolazione del DOM

Abbiamo descritto nel capitolo precedente come Dart interagisce con il DOM. In questo capitolo verranno ripresi gli elementi fondamentali utili al puro confronto con jQuery.

jQuery propone una forte ottimizzazione e semplificazione rispetto al passato. Innanzitutto per accedere ad un elemento sfrutta i cosiddetti selettori. I selettori vengono passati come paramentro della funzione *jQuery()* o del suo alias *$()*.

I selettori sono quelli che ci permettono di selezionare un elemento in base al tipo di tag, alla classe a cui esso appartiene o in base all'ID. Vediamo alcuni esempi:

```
$('#myDiv')  //  per selezionare l'elemento con id = myDiv
$('div') // per selezionare tutti i div

$('input[value="foo"]') // per selezionare tutti i campi con
                                    //valore = foo

$('.myClass') // per selezionare tutti gli elementi di
                    //classe = myClass
```

Dart invece utilizza le funzioni *query()* e *queryAll()*. La modalità per accedere agli elementi è quella Object-Oriented. Ricordiamo due esempi:

```
elem.query('#foo');
elem.queryAll('div');
```

Analizzando invece l'accesso agli attributi di un elemento, jQuery e Dart possiedono circa lo stesso set di funzioni. In jQuery, tuttavia, vengono spesso sfruttati gli stessi metodi per più funzioni, come ad esempio la funzione *attr()* che viene di fatto usata sia come metodo *getter* che come *setter*.

```
$('#myDiv').attr('name')
```

Tale funzione viene utilizzata per selezionare il valore dell'attributo *name*, se l'attributo non è settato ritorna *undefined*.

Se si vuole invece rimuovere un attributo si usa la seguente funzione:

```
$('#myDiv').removeAttr('name')
```

Dart invece sfrutta le proprietà degli elementi e quindi si usa, per eseguire le operazioni sopra descritte, la seguente sintassi:

```
elem.attributes['name']; //seleziona un attributo
elem.attributes.remove('name');  //elimina un attributo
```

Per quanto riguarda l'aggiunta di nuovi elementi al DOM, jQuery fa uso di unico metodo, *add()*, che può accettare come parametri sia elementi puri del DOM, sia codice HTML, sia altri selettori.

```
$('#myDiv').add("div")
```

Dart invece usa la seguente sintassi:

```
new Element.tag('div');
```

Anche dal punto di vista degli eventi jQuery risulta assai rivoluzionario rispetto a JavaScript. Nelle API esistono infatti tantissime funzioni, ognuna delle quali registra un *handler* ad hoc per l'evento generato. Per esempio, se si vuole lanciare un *alert* dopo aver clickato su un *div*, si deve usare la seguente sintassi:

```
$('#myDiv').click(function(){
   alert('Hello World!');
})
```

è equivalente a:

```
$('.menu').bind('click', function() {
   alert('Hello World!');
});
```

Si dovrà quindi passare come parametro alla funzione in questione direttamente l'*handler* che si occuperà della gestione delle operazioni conseguenti alla cattura dell'evento.

Dart lavora in modo molto simile e per assegnare un evento si dovrà scrivere:

```
elem.on.click.add(
(event) => print('click!'));
```

4.4 Interazione con il server

Prima di parlare di come interagiscono jQuery e Dart con un server è necessario introdurre due tecnologie fondamentali per comprendere il funzionamento delle chiamate client-server.

4.4.1 JSON

L'esigenza di potersi interfacciare in modo universale con diverse tecnologie ha dato luce a JSON (*JavaScript Object Notation*), un sistema di scambio dati molto semplice basato su una formattazione di testo tale da rendere differenti tipi di variabili facilmente rappresentabili. JSON è un formato adatto per lo scambio dei dati in applicazioni client-server. È basato sul linguaggio JavaScript, ma ne è indipendente. La semplicità di JSON ne ha decretato un rapido utilizzo specialmente nella programmazione in AJAX. I tipi di dato che possiamo usare con JSON per lo scambio di informazioni sono due:

- array con soli indici numerici
- oggetti

La scelta limitata è ricaduta su questi tipi di contenitori perché presenti nella maggior parte dei linguaggi di sviluppo. Anche i tipi che si possono racchiudere al loro interno sono abbastanza comuni. Essi infatti possono contenere dati primitivi come:

- numeri
- booleani
- stringhe
- valore nullo (*null*)

Usare JSON per l'invio di solo testo, di un numero o di un singolo valore booleano non è un approccio utile né adatto alla conversione dei valori, che dovranno essere sempre racchiusi all'interno degli unici contenitori riconosciuti, array ed oggetti, anche qualora non dovessero avere alcun dato. Esempio:

```
// variabile non adatta a JSON
var str = "test";

// variabili adatte a JSON
var arr = [str];
var obj = {testo:str};
```

4.4.2 Ajax

AJAX (*Asynchronous JavaScript and XML*) è una tecnica di sviluppo per la realizzazione di applicazioni web interattive. Lo sviluppo di applicazioni HTML con AJAX si basa su uno scambio di dati in background fra web browser e server. Questo consente l'aggiornamento dinamico di una pagina web senza esplicito ricaricamento da parte dell'utente.

AJAX inoltre è asincrono nel senso che i dati sono richiesti al server e caricati in background senza interferire con il comportamento della pagina esistente.

Normalmente le funzioni richiamate sono scritte con il linguaggio JavaScript. Tuttavia, e a dispetto del nome, l'uso di JavaScript e di XML non è obbligatorio, come non è necessario che le richieste di caricamento debbano essere necessariamente asincrone.

La tecnica Ajax utilizza una combinazione di:

- HTML (o XHTML) e CSS per il markup e lo stile;
- DOM (*Document Object Model*);
- l'oggetto *XMLHttpRequest* per l'interscambio asincrono dei dati tra il browser dell'utente e il web server;
- un formato di scambio di dati, per lo più XML o JSON.

AJAX permette ad una pagina web di richiedere nuovi dati dopo che è stata caricata sul browser web, spesso in risposta ad un'azione dell'utente. Un esempio può essere questo: mentre l'utente sta digitando qualcosa in un box di ricerca, il client manda quello che è stato digitato fino a quel momento al server, il quale risponder à con una lista di possibili termini presi dal database. Questi possono essere visualizzati in una lista drop-down, così che l'utente possa smettere di scrivere e selezionare direttamente una stringa.

4.4.3 jQuery e le chiamate AJAX

Per comprendere il modo in cui jQuery gestisce AJAX bisogna introdurre gli eventi. Questi sono una serie di eventi che si susseguono prima, durante e dopo una richiesta e vengono suddivisi in due categorie:

- locali: eventi che si verificano all'interno di una chiamata e che possono essere impostati solo all'interno di una funzione di '$'.
- globali: eventi che coinvolgono tutto il DOM e che quindi possono essere rintracciati ed impostati su una collezione di elementi con metodi come *bind()*.

La funzione principale per inviare richieste AJAX è il metodo statico *$.ajax()*. Dati i molti aspetti della chiamata che possono essere personalizzati, la funzione accetta un unico oggetto JavaScript con i parametri di base ed altri necessari per sovrascrivere i valori di default. I parametri di base sono:

- *url*: l'indirizzo al quale inviare la chiamata.
- *success*: funzione da lanciare se la richiesta ha successo. Accetta come argomenti i dati restituiti dal server (interpretati di default come html o xml) e lo stato della chiamata.
- *error*: funzione lanciata in caso di errore. Accetta un riferimento alla chiamata *XMLHttpRequest*, il suo stato ed eventuali errori notificati.

Con questi tre parametri è possibile impostare una prima semplice chiamata AJAX di esempio:

```
$.ajax({
    url : "mioserver.html",
    success : function (data,stato) {
        $("#risultati").html(data);
        $("#statoChiamata").text(stato);
    },
    error : function (richiesta,stato,errori) {
        alert("E' evvenuto un errore.
                Lo stato della chiamata: "+stato);
    }
});
```

Nell'esempio precedente, se la chiamata ha successo i dati verranno inseriti all'interno di specifici elementi DOM, altrimenti verrà mostrato un messaggio di errore.

Mentre altri linguaggi devono lavorare direttamente sull'oggetto XMLHttpRequest, jQuery offre diversi astrazioni e mette a disposizione utili metodi per implementare AJAX in modo molto semplice. Innanzitutto esiste la funzione *$.ajax()* che ritorna un oggetto *XMLHttpRequest* e offre molte opzioni.

Se si vogliono salvare dati sul server sarà necessario richiamare una funzione AJAX che ha come *type* "POST". Ad esempio:

```
$.ajax({
   type: "POST",
   url: "some.php",
   data: "name=John&location=Boston",
   success: function(msg){
     alert( "Data Saved: " + msg );
   }
 });
```

Se si vogliono invece chiedere dati al server si utilizzerà la stessa funzione ma con *type* settato a "GET ".

```
$.ajax({
  type : 'GET',
  url : 'search.php',
  success: function (data) {
    //gestione dei dati risultanti
  }
});
```

Si possono usare anche chiamate specifiche di questo tipo *$.post* e *$.get*. Queste due funzioni presentano la stessa dicitura e necessitano dei seguenti parametri:

```
$.post( url [, data] [, success(data, textStatus, jqXHR)]
                                          [, dataType] )
```

- *url*: stringa contenente l'url verso il quale viene inviata la richiesta.
- *data*: stringa o array "chiave-valore" contenente i dati da inviare al server insieme alla richiesta.
- *success(data, textStatus, jqXHR)*: funzione di *callback* che viene eseguita se la richiesta ha avuto successo.

- *dataType*: Tipo di dato in risposta alla richiesta (xml, json, script, text, html).

È importante sottolineare il fatto che attraverso il *dataType* è possibile specificare come formato dati JSON. Il modo più semplice per caricare dati JSON in jQuery è usare la funzione *$.getJSON()* e usando la funzione di *callback* per eseguire operazioni con i dati. La signature della funzione è:

```
$.getJSON( url [, data] [, success(data, textStatus, jqXHR)] )
```

E possiamo usarla in questo modo:

```
$.getJSON('/json/somedata.json', function(data) {
    // do something with the data here
});
```

Se si necessita di settare qualche altra proprietà AJAX si può usare la funzione *$.ajax()*. Il seguente codice è equivalente al precedente.

```
$.ajax({
    dataType: 'json',
    success: function(data) {
        // do something
    },
    url: '/json/somedata.json'
});
```

È necessario però che questi dati vengano convertiti in dati leggibili in JavaScript . Per cui se, ad esempio, viene restituita da una di queste funzioni una stringa JSON, non decodificata da jQuery automaticamente,questa può essere parserizzata in un regolare array tramite la funzione *$.parseJSON()*. Se *string* nell'esempio sottostante contiene una stringa JSON essa sarà convertira in un array:

```
data = $.parseJSON(string);
```

La stringa può potenzialmente essere recuperata come testo dal server usando AJAX e poi convertita in un array. Altrimenti si può usare il *dataType json* e lasciare che jQuery se ne occupi:

```
$.ajax({
    dataType: 'text',
    success: function(string) {
        data = $.parseJSON(string);
        // do something
    },
    url: '/json/somedata.json'
});
```

4.4.4 Dart e i JsonObject

Quando si comunica con un server, si usano le *XMLHttpRequest* della libreria *dart:html*. In particolare per ottenere oggetti da un server si usa il metodo *HTTP GET*. *XMLHttpRequest* fornisce un costruttore named chiamato *getTEMPNAME* che prende un URL e una funzione di *callback* da chiamare quando il server risponde. Esempio:

```
getLanguageData(String languageName, onSuccess(XMLHttpRequest req)) {
  var url = "http://my-site.com/programming-languages/$languageName";

  // call the web server asynchronously
  var request = new XMLHttpRequest.getTEMPNAME(url, onSuccess);
}

// print the raw json response text from the server
onSuccess(XMLHttpRequest req) {
   print(req.responseText); // print the received raw JSON text
}
```

```
getLanguageData("dart", onSuccess);
```

Per creare un nuovo oggetto sul server invece si usa il metodo *HTTP POST*. Se si vuole avere la notifica quando la richiesta è stata completata si può usare il listener *readyStateChange*. L'esempio sotto chiama la funzione *onSuccess* quando la richiesta è stata completata.

```
saveLanguageData(String data, onSuccess(XMLHttpRequest req)) {
  XMLHttpRequest req = new XMLHttpRequest(); // create a new XHR

  var url = "http://example.com/programming-languages/";
  req.open("POST", url); // POST to send data

  req.on.readyStateChange.add((Event e) {
    if (req.readyState == XMLHttpRequest.DONE &&
        (req.status == 200 || req.status == 0)) {
                onSuccess(req); // called when the POST
                          //successfully complete
    }
  });

  req.send(data); // kick off the request to the server
}

// print the raw json response text from the server
onSuccess(XMLHttpRequest req) {
   print(req.responseText); // print the received raw JSON text
}

String jsonData = '{"language":"dart"}'; // etc...
saveLanguageData(stringData, onSuccess); // send the data to
                                         // the server
```

Anche in Dart esistono metodi per utilizzare gli oggetti JSON che si ottengono in risposta da queste chiamate. A questo proposito esistono due funzioni: *JSON.parse()* e *JSON.stringify()*.

La funzione *parse()* converte una stringa JSON in una lista di valori o in una mappa di coppie "chiave-valore", a seconda del formato di JSON:

```
String listAsJson = '["Dart",0.8]'; // input List of data
List parsedList = JSON.parse(listAsJson);
print(parsedList[0]); // Dart
print(parsedList[1]); // 0.8

String mapAsJson = '{"language":"dart"}';  // input Map of data
Map parsedMap = JSON.parse(mapAsJson);
print(parsedMap["language"]); // dart
```

È possibile usare *JSON.parse()* per convertire il responso di *XMLHttpRequest* da un semplice testo ad un oggetto Dart:

```
onSuccess(XMLHttpRequest req) {
  Map data = JSON.parse(req.responseText); // parse response text
  print(data["language"]); // dart
  print(data["targets"][0]); // dartium
  print(data["website"]["homepage"]); // www.dartlang.org
}
```

Uno dei benefici derivanti dall'uso di Dart, come detto più volte, è il supporto per ai tipi opzionali statici. I tipi statici aiutano a trovare velocemente gli errori permettendo al tool di trovare discordanze di tipo prima di eseguire il codice e di generare eccezioni non appena si presenta un problema a runtime. Un altro beneficio è il fatto che l'editor di Dart utilizza le informazioni sui tipi per dare informazioni di auto-completamento utili quando si incontrano nuove librerie o nuove strutture dati. Idealmente, si può accedere ai dati JSON in modo strutturale, ottenendo il vantaggio di catturare gli errori. Il seguente esempio mostra una struttura più OO tipica di Dart:

```
Language data = // ... initialize data ...

// property access is validated by tools
print(data.language);
print(data.targets[0]);
print(data.website.homepage);
```

La soluzione ideale è quella di combinare la flessibilità delle mappe con le interfacce. Per questo motivo viene introdotto un nuovo tipo: *JsonObject*.

Conoscendo la struttura dei dati che ci si aspetta, si può infatti definire un insieme di interfacce per ottenere questa struttura.

```
interface LanguageWebsite extends JsonObject {
  String homepage;
  String api;
}

interface Language extends JsonObject {
  String language;
  List<String> targets;
  LanguageWebsite website;
}
```

È interessante notare che entrambe le interfacce estendono la classe *JsonObject*. Una delle caratteristiche sorprendenti di Dart è che le classi sono anche interfacce e si può dire che Dart supporta "interfacce implicite per le classi". La combinazione di *JsonObject* con le interfacce permette di usare i tipi statici per i dati JSON, come nel seguente esempio:

```
// assign a new JsonObject instance to a variable of type Language
Language data = new JsonObject.fromJsonString(req.responseText);

// tools can now validate the property access
print(data.language);
```

```
print(data.targets[0]);

// nested types are also strongly typed
LanguageWebsite website = data.website; // contains a JsonObject
website.homepage = "http://www.dartlang.org";
```

JsonObject permette inoltre di creare un nuovo oggetto vuoto basato su questa interfaccia, usando il costruttore di default, senza dover necessariamente convertire il dato JSON:

```
Language data = new JsonObject();
data.language = "Dart";
data.website = new LanguageWebsite();
data.website.homepage = "http://www.dartlang.org";
```

JsonObject inoltre implementa l'interfaccia *Map*, il che significa è possibile usare la sintassi standard di questo tipo di oggetto:

```
Language data = new JsonObject();
data["language"] = "Dart"; // standard map syntax
```

Per lo stesso motivo si può passare un oggetto *JsonObject* a *JSON.stringify()*, metodo che converte una mappa in oggetto JSON per poter poi inviare dati al server:

```
Language data = new JsonObject.fromJsonString(req.responseText);

// later...
// convert the JsonObject data back to a string
String json = JSON.stringify(data);

// and POST it back to the server
XMLHttpRequest req = new XMLHttpRequest();
req.open("POST", url);
req.send(json);
```

4.5 Dart e node.js

4.6 Introduzione a node.js

Node.js [8] è un framework *event-driven* per il motore Javascript V8 [1] relativo all'utilizzo di JavaScript lato server. Node.js è stato progettato per scrivere applicazioni web scalabili, in particolare server web. La sua caratteristica principale risiede nella possibilità che offre di accedere alle risorse del sistema operativo in modalità event-driven e non sfruttando il classico modello basato su processi o thread concorrenti, tipico dei classici web server. Ogni azione quindi risulta asincrona.

Il framework presenta una serie di API proprietarie che permettono di effettuare praticamente qualsiasi operazione a livello di rete e di socket, oltre alle librerie di alto livello come HTTP, SSL o DNS. Vediamo ora i principali moduli disponibili.

- *Globals*: è un modulo atipico in quanto non rappresenta una vera e propria libreria ma una serie di API incluse nel namespace globale dell'applicazione e quindi richiamabili direttamente. Tra le funzioni di questo pseudo-modulo vi sono funzioni per includere moduli aggiuntivi, lavorare con i timer o accedere allo standard input e allo standard error.

- *Http*: include la classe bootstrap per avviare un server web e una serie di oggetti che permettono di utilizzare il protocollo facilmente.

- Url: permette di lavorare con gli url, creandoli a partire da oggetti oppure convertendoli a partire da stringhe.

- *Path*: vi sono metodi per navigare tra le cartelle e per montare path complessi a partire da più stringhe, cosa che è di vitale importanza per applicazioni che lavorano con il *filesystem*.

[1] V8 è un motore Javascript open source sviluppato da Google, attualmente incluso in Google Chrome. Esegue e compila Javascript, ha un garbage collector altamente efficiente e gestisce in modo efficace l'allocazione di memoria per gli oggetti.

- *FS - File System*: permette, insieme al precedente, di lavorare a stretto contatto con il *filesystem* della macchina, eseguendo tutte le operazioni tipiche di questo ambito come ad esempio la copia, la rinomina, la cancellazione, la lettura e la scrittura di file e cartelle.

- *Util*: include una serie di funzioni di utilità. Vi sono una serie di funzioni logiche, funzioni per la formattazione di stringhe e funzioni di monitoraggio del flusso, come quelle per il debug.

- *Net*: permette agli sviluppatori più esperti di implementare applicazioni client-server a prescindere dal protocollo HTTP, sfruttando i socket come strumenti di basso livello e lavorando direttamente con le connessioni.

4.7 Gestione I/O con node.js

L'idea introdotta dall'*event driven* è quella di avere un unico processo che acquisisce gli eventi in ingresso ed associa ad ogni operazione di I/O una specifica funzione di *callback* che ne abbia la responsabilità. Il processo principale (*EventLoop*) scorre gli eventi in input ed esegue una funzione di callback associata. Successivamente sarà proprio questa funzione ad avvertire di aver completato l'operazione.

Per leggere un file si utilizzano metodi della libreria POSIX. Esempio di lettura:

```
var fs = require('fs');
fs.readFile('/etc/passwd', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

La forma asincrona prende in ingresso una funzione di *callback* come ultimo argomento. Gli argomenti passati a tale funzione dipendono dal metodo,

ma il primo argomento deve necessariamente essere riservato ad una eccezione. Se l'operazione è stata completata con successo, allora il primo argomento ritornerà *null* o *undefined*. Quando si usa la forma sincrona ogni eccezione viene immediatamente generata. Si può usare il costrutto *try/catch* per gestire l'eccezione o permettere di catturarla. Esempio di chiamata sincrona:

```
var fs = require('fs');

try {
  var data = fs.readFileSync('README.md', 'ascii');
  console.log(data);
}
catch (err) {
  console.error("There was an error opening the file:");
  console.log(err);
}
```

In node.js è possibile fare la stessa operazione usando un *ReadStream*. Grazie a questo buffer si possono leggere file molto grandi. Esempio:

```
var fs = require('fs');

var read_stream = fs.createReadStream('README.md', {encoding: 'ascii'});
read_stream.on("data", function(data){
  process.stdout.write(data);
});
read_stream.on("error", function(err){
  console.error("An error occurred: %s", err)
});
read_stream.on("close", function(){
  console.log("File closed.")
});
```

4.8 Operazioni di I/O in Dart

La libreria *dart:io* è finalizzata alla scrittura di codice server-side. Dart è un linguaggio di programmazione single-thread. Se un'operazione blocca un thread Dart, l'applicazione non farà progressi finché l'operazione non sarà completata. È importante però che nessuna operazione di I/O blocchi l'esecuzione. Per questo motivo *dart:io* usa un modello di programmazione asincrona ispirata a node.js.

La libreria *dart:io* fornisce accesso ai file e alle cartelle attraverso le interfacce *File* e *Directory*. Il seguente esempio stampa la sorgente del proprio codice.

```
#import('dart:io');

main() {
  var options = new Options();
  var file = new File(options.script);
  Future<String> finishedReading = file.readAsText(Encoding.ASCII);
  finishedReading.then((text) => print(text));
}
```

Notare che il metodo *readAsText()* è asincrono; esso ritorna un oggetto *Future* che restituisce il contenuto del file una volta che è stato letto dal sistema sottostante. Questa asincronicità permette ai thread Dart di eseguire altri lavori mentre si attende che l'operazione di I/O sia completata. Prendiamo ora ad esempio l'apertura di un file e lettura di una parte di esso. Si può fare in due modi: aprendo il file per un accesso casuale o aprire un *InputStream* per il file.

Questo codice di esempio apre il file per leggerlo e poi legge un byte alla volta finché non trova il carattere ';'.

```
#import('dart:io');
main() {
```

```
  var options = new Options();
  var semicolon = ';'.charCodeAt(0);
  var result = [];
  new File(options.script).open(FileMode.READ).then
                                    ((RandomAccessFile file){
    // Callback to deal with each byte.
    void onByte(int byte) {
      result.add(byte);
      if (byte == semicolon) {
        print(new String.fromCharCodes(result));
        file.close();
      } else {
        file.readByte().then(onByte);
      }
    }
    openedFile.readByte().then(onByte);
  });
}
```

L'uso di *then()* implica un oggetto Future in azione [2]. Entrambi i metodi *open()* e *readByte()* ritornano un oggetto *Future*. Questo codice mostra un semplice utilizzo delle operazioni di accesso random, che possono permettere di scrivere, posizionarsi in un punto del file, troncare un file e molto altro.

Il seguente codice utilizza invece gli *InputStream*. Questi sono oggetti attivi che iniziano a leggere i dati non appena vengono creati. Dovunque i dati siano disponibili, l'*handler onData* viene chiamato e i dati possono essere trasferiti fuori dallo stream usando il metodo *read()*. L'*handler onData* continua ad essere chiamato in tale modo fino a che i dati possono essere letti dallo stream.

[2]Un oggetto Future è usato per ottenere un valore in un futuro. I Receivers di un oggetto Future possono ottenere il valore passando una funzione di callback a then.

```
#import('dart:io');

main() {
  Options options = new Options();
  List result = [];

  InputStream stream = new File(options.script).openInputStream();
  int semicolon = ';'.charCodeAt(0);
  stream.onData = () {
    result.addAll(stream.read(1));
    if (result.last() == semicolon) {
      print(new String.fromCharCodes(result));
      stream.close();
    }
  };
}
```

Gli *InputStream* sono usati in diversi modi nella libreria *dart:io* quando si lavora con stdin, file, socket, connessioni http e molto altro e la stessa cosa vale per gli *OutputStream*.

4.9 Web Server in node.js

Scrivere un server in node.js è molto semplice. Ecco un esempio:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

La funzione di *callback* sarà invocata dal motore V8 ogni volta che il server riceverà una richiesta e assumerà come parametri:

- *request*, un oggetto della classe *http.ServerRequest* che rappresenta la richiesta HTTP e dalla quale poter leggere eventuali parametri inviati in *GET* o *POST*;

- *response*, un'istanza della classe *http.ServerResponse*, utile per impostare il contenuto che si vorrà inviare all'utente ed eventuali altre informazioni.

Vediamo un esempio di invio di dati tramite *POST*. Per poter accedere al corpo della richiesta dobbiamo sfruttare due eventi esposti dall'oggetto *http.ServerRequest*: *data* e *end*. Il primo viene generato più volte, ad ogni pacchetto di dati ricevuto; il secondo invece è scatenato solo una volta, al completamento della ricezione. Guardiamo un esempio:

```
var body = '';
req.on('data', function(data) {
  body += data;
});

req.on('end', function() {
  var post = require('querystring').parse(body);
  console.log(post);
});
```

Per prima cosa definiamo una variabile stringa vuota, che sarà a mano a mano popolata grazie alla funzione di *callback* agganciata all'evento *data*. Allo scatenarsi di end invece facciamo il *parser* del corpo della richiesta grazie al modulo *querystring* per ottenere una mappa "chiave-valore" dei parametri. Una caratteristica molto comoda è la possibilità di mandare diversi parametri con lo stesso nome: in questo caso il nostro valore della mappa non sarà un elemento scalare, ma un vettore di valori.

4.10 Web Server Dart

Per scrivere un semplice web server tutto ciò che si deve fare è creare un *HttpServer* e collegarlo ad un *defaultRequestHandler*. Qui di seguito il codice:

```
#import('dart:io');

main() {
 var server = new HttpServer();
 server.listen('127.0.0.1', 8080);
 server.defaultRequestHandler = (HttpRequest request,
                                 HttpResponse response) {
   response.outputStream.write('Hello, world'.charCodes());
   response.outputStream.close();
 };
}
```

Il percorso base per ogni file servito è la locazione dello script. Se non è specificato nessun path in una richiesta verrà fornito *index.html*. Se il file non si trova la risposta sarà uno stato "404 Not Found".

```
#import('dart:io');

send404(HttpResponse response) {
  response.statusCode = HttpStatus.NOT_FOUND;
  response.outputStream.close();
}

startServer(String basePath) {
 var server = new HttpServer();
 server.listen('127.0.0.1', 8080);
 server.defaultRequestHandler = (HttpRequest request,
```

```
                          HttpResponse response) {
    final String path = request.path == '/' ?
                    '/index.html' : request.path;
    final File file = new File('${basePath}${path}');
    file.exists().then((bool found) {
      if (found) {
        file.fullPath().then((String fullPath) {
          if (!fullPath.startsWith(basePath)) {
            _send404(response);
          } else {
            file.openInputStream().pipe(response.outputStream);
          }
        });
      } else {
        _send404(response);
      }
    });
  };
}

main() {
  // Compute base path for the request based on the location of the
  // script and then start the server.
  File script = new File(new Options().script);
  script.directory().then((Directory d) {
    startServer(d.path);
  });
}
```

L'analisi relativa all'invio e ricezione dati tramite JSON è stata già trattata nel confronto con jQuery.

Capitolo 5

Dart e CoffeeScript

5.1 Introduzione a CoffeeScript

CoffeeScript [14] è un piccolo linguaggio, creato da Jeremy Ashkenas, che compila in JavaScript. CoffeeScript è un tentativo di sfruttare le caratteristiche migliori di JavaScript in modo più semplice e la sua regola è: "It's just JavaScript". Quindi il codice viene compilato nell'equivalente JS e non vi è nessuna interpretazione a runtime. Si può inoltre usare qualunque libreria JavaScript in maniera scorrevole e viceversa.

CoffeeScript ha una particolarità: usa spazi bianchi per delimitare i blocchi di codice. Non è necessario quindi utilizzare ';' per terminare le espressioni. Stessa cosa vale per le parentesi graffe '{}' che normalmente si usano per racchiudere blocchi di codice nelle funzioni, negli if-statement e in altri costrutti. Al loro posto ci si serve dell'indentazione. Non è nemmeno necessario usare le parentesi per passare un argomento. Basterà scrivere:

```
console.log sys.inspect object
//equivale a:
console.log(sys.inspect(object));
```

CoffeeScript può essere incluso direttamente all'interno del codice HTML tramite il seguente tag:

```
<script type="text/coffeescript">
```

5.2 Funzioni

Le funzioni sono definite da tre elementi: una lista opzionale di parametri posti fra parentesi, una freccia ('->') e il corpo della funzione. Esempio:

```
square = (x) -> x * x
```

Le funzioni possono anche avere argomenti di default ed è possibile sovrascriverli passando un argomento non nullo.

```
fill = (container, liquid = "coffee") ->
  "Filling the #{container} with #{liquid}..."
```

CoffeeScript possiede un'ulteriore particolarità. Se non si dichiarano statement di ritorno alle funzioni di CoffeeScript, quest'ultime ritorneranno in ogni caso un valore finale. Il compilatore CoffeeScript si deve assicurare quindi che tutti gli statement possano essere usati come espressione. Esempio:

```
grade = (student) ->
  if student.excellentWork
    "A+"
  else if student.okayStuff
    if student.triedHard then "B" else "B-"
  else
    "C"


eldest = if 24 > 21 then "Liz" else "Ike"
```

Anche se le funzioni ritornano sempre il loro valore finale, è comunque possibile e consigliato far ritornare alla funzione qualcosa scrivendo un ritorno esplicito (tramite *return value*).

Inoltre, poiché le dichiarazioni delle variabili si trovano all'inizio dello *scope*, gli assegnamenti possono essere usati dentro le espressioni stesse, anche per le variabili che non sono state usate prima:

```
six = (one = 1) + (two = 2) + (three = 3)
```

Ricordiamo che in Dart le funzioni possono essere definite in tre modi diversi (vedi sez. 2.5). Uno dei tre è molto simile alla notazione usata in CoffeeScript e fa uso della freccia '=>'. Il significato però non è lo stesso e non si può assolutamente dire che tutto è un'espressione in Dart. Vediamo come la funzione, precedemente scritta in CoffeeScript, può essere scritta in Dart con tale notazione:

```
num square(x) => x*x ;
```

Si può notare come la notazione Dart sia molto più intuitiva.

Per quanto riguarda i parametri di default abbiamo visto come sia possibile impostarli in Dart nelle funzioni con paramentri opzionali in tale modo:

```
String say(String from, String msg, [String device]) {
   var result = "$from says $msg";
   if (device != null) {
     result = "$result with a $device";
   }
   return result;
}
```

5.3 Oggetti, Classi e Interfacce

5.3.1 Gli oggetti in CoffeeScript

Il modo di definire oggetti e array in CoffeeScript somiglia molto a quello in JavaScript. Gli oggetti possono essere creati usando l'indentazione invece di parentesi esplicite. Esempi:

```
song = ["do", "re", "mi", "fa", "so"]

singers = {Jagger: "Rock", Elvis: "Roll"}

bitlist = [
  1, 0, 1
  0, 0, 1
  1, 1, 0
]

rectangle =
   width: 3
   height: 4

   area: ->
      return width*height
```

Per creare un nuovo oggetto si possono usare i metodi *factory*.

```
createRectangle = (width, height) -> {
   width
   height

   area: -> return width*height
}
```

5.3.2 Classi in CoffeeScript

CoffeeScript fornisce una struttura base per le classi che permette di dare un nome ad una classe, settare una superclasse, assegnare proprietà *prototype* e definire costruttori, in una singola espressione di assegnamento. Anche qui, proprio come in Dart, vi sono i costruttori *named*.

Vediamo un esempio:

```
class Animal
  constructor: (@name) ->

  move: (meters) ->
    alert @name + " moved #{meters}m."

class Snake extends Animal
  move: ->
    alert "Slithering..."
    super 5

class Horse extends Animal
  move: ->
    alert "Galloping..."
    super 45

sam = new Snake "Sammy the Python"
tom = new Horse "Tommy the Palomino"

sam.move()
tom.move()
```

L'operatore *extends* permette di lavorare meglio con le proprietà *prototype* e può essere usato per creare una catena di ereditarietà tra due coppie qualunque di funzioni costruttori. Usare '::' dà facile accesso al prototipo di

un oggetto. *Super()* invece è una chiamata ad un metodo con lo stesso nome della classe genitore.

```
String::dasherize = ->
  this.replace /_/g, "-"
```

In CoffeeScript esiste una scorciatoia per chiamare this ed è il simbolo '@'. Quindi in tale modo si può assegnare una qualunque proprietá all'oggetto riferito alla classe corrente:

```
@property: value
```

5.3.3 Dart e gli oggetti

Dart, come più volte sottolineato, ha un approccio Object-Oriented e Class-Based per cui la struttura di un oggetto si costruisce attraverso le classi (vedi sez. 2.2). Per definire un oggetto, ad esempio un rettangolo, si scrive:

```
class Rectangle {
  num width, height;

  num area() {
    return width * height;
  }
}
```

Abbiamo anche visto che per creare un oggetto basta richiamare il costruttore preceduto dalla parola chiave *new*, proprio come nei linguaggi OO:

```
Rectangle r = new Rectangle(width, height);
```

5.4 Stringhe

In CoffeeScript si creano stringhe in questo modo:

```
author = "Wittgenstein"
quote  = "A picture is a fact. -- #{ author }"
```

Si possono definire anche stringhe multilinea. Esempio:

```
mobyDick = "Call me Ishmael. Some years ago --
 never mind how long precisely -- having little
 or no money in my purse, and nothing particular
 to interest me on shore, I thought I would sail
 about a little and see the watery part of the
 world..."
```

Anche Dart ha moltissime funzionalità per creare stringhe(vedi sez. 3.3.3). Si possono interpolare le stringhe utilizzando però il carattere '$'. Riprendiamo il primo esempio:

```
var author = "Wittgenstein";
var quote = "A picture is a fact. $author";
```

È possibile creare una stringa anche concatenando semplicemente diverse stringhe senza l'uso di ulteriori operatori. Ricordiamo:

```
var s = 'String ''concatenation'
   " works even over line breaks.";
```

5.5 Scope Lessicale

CoffeeScript supporta lo scope lessicale. Questo significa che una variabile che è stata dichiarata all'interno di una funzione non è visibile dall'esterno. Dopo la dichiarazione una variabile vive dentro lo *scope* in cui un valore è stato assegnato senza la necessità di essere ridichiarata. Poiché non è possibile

usare la parola chiave *var*, non è nemmeno possibile dichiarare semplicemente una variabile e usarla in un contesto differente senza assegnargli un valore. Esempio:

```
someFunction = -> podcast = 'Hardcore history'
podcast = 'Astronomy cast'
someFunction()

console.log podcast        # This outputs 'Astronomy cast'
```

Importante è il fatto che una variabile che è stata dichiarata all'interno di un contesto non può oscurare una variabile con lo stesso nome che è stata dichiarata in un contesto esterno. Per questo motivo è necessario porre attenzione a non riutilizzare il nome di una variabile esterna accidentalmente nel caso in cui si scrivano funzioni fortemente innestate.

5.6 Bind di funzioni

La freccia '=>' (*fat arrow*) può essere usata per definire una funzione o per fare il bind di essa al corrente valore di this. Questo è utile quando si usa una libreria basata sulle funzioni di *callback* come *Prototype* o *jQuery*, per creare funzioni iteratrici o funzioni *event-handler* da collegare a *bind*. Le funzioni create con tale simbolo sono in grado di accedere alle proprietà di *this* dove sono definite. Esempio:

```
Account = (customer, cart) ->
  @customer = customer
  @cart = cart

  $('.shopping_cart').bind 'click', (event) =>
    @customer.purchase @cart
```

Se si '->' nella funzione di *callback*, *@customer* avrà il riferimento alla proprietà indefinita *customer*' dell'elemento del DOM e provando a chiamare *purchase()* su di essa sarà generata un'eccezione.

Quando si usa nella definizione di una classe, i metodi dichiarati con '=>' saranno automaticamente collegatii ad ogni istanza della classe con quando questa viene costruita.

Per fare il bind di una funzione Dart si scrive semplicemente (vedi sez. 3.4.4):

```
window.on.click.add((event) {
print('You clicked the window.');
})
```

Conclusioni

Nello scenario attuale esistono moltissime possibilità per scrivere applicazioni web sempre più innovative.

Abbiamo visto come il linguaggio JavaScript sia il più utilizzato per la scrittura di programmi web-based. Esso infatti offre funzionalità in grado di creare pagine web dinamiche in modo piuttosto semplice, anche se presenta alcuni limiti. Innanzitutto non supporta il concetto di classe, sebbene sia un linguaggio Object-based. Per questo motivo non esiste un meccanismo di ereditarietà ben definito, ma si sfruttano delle proprietà degli oggetti chiamate *prototype*. Questo implica una mancanza di strutturalità che in Dart permette una miglior gestione del codice e un suo possibile riutilizzo in futuro. Altro aspetto importante è la tipizzazione. Sebbene sia dinamica in entrambi i linguaggi, in Dart esiste la possibilità di introdurre i tipi e di aver un controllo su di essi. Infine JavaScript è stato il primo linguaggio a rendere interattiva la pagina con l'introduzione di eventi e di funzioni in grado di lavorare sugli elementi del DOM. I meccanismi che permettono queste operazioni risultano però poco intuitivi. Il team Dart, oltre alla semplificazione di questi meccanismi e all'uso del paradigma Object-oriented anche in questo aspetto, si è preoccupato di rivoluzionare alcune regole di base del DOM rendendo l'accesso agli elementi più generalizzato.

A competere con queste nuove caratteristiche di Dart è una libreria JavaScript lato client, chiamata jQuery. Grazie ad essa la scrittura di molte funzioni, in particolare di interazione con il DOM, è stata semplificata notevolmente. Questa libreria è importante anche perchè ha permesso di intro-

durre le chiamate AJAX, permettendo una gestione asincrona delle chiamate client-server. In questo ambito Dart supera jQuery nella gestione degli oggetti JSON. Infatti mentre in jQuery non esistono oggetti veri e propri di questo tipo ed è necessario convertirli in tipi primitivi di JavaScript, Dart implementa un nuovo tipo di oggetto mirato proprio a sfruttare al meglio JSON. Abbiamo visto che la combinazione di *JsonObject* e interfacce permette di avere accesso a strutture dati anche molto complesse attraverso la *dot notation*.

Abbiamo detto che Dart permette di sviluppare non solo lato client ma anche lato server. Tutto ciò non esiste nel nativo JavaScript ma grazie all'introduzione di node.js è possibile scrivere applicazioni server-side piuttosto facilmente. Dart e node.js condividono il paradigma *event-driven* e permettono la gestione di chiamate I/O asincrone in modo tale da evitare blocchi nell'esecuzione del programma. Node.js si basa su librerie POSIX mentre la libreria Dart assomiglia molto a quella Java. Infatti nel secondo caso l'uso, ancora una volta, del paradigma OO permette di facilitare l'accesso ai file e la loro modifica o lettura.

Nonostante queste librerie, come altre JavaScript, rendano il linguaggio sempre più completo, vi sono problematiche di base. CoffeeScript è un linguaggio derivato da JavaScript che si propone di modificare alcune caratteristiche troppo complesse, rendendole più semplici. Come Dart, CoffeeScript sfrutta le classi e l'ereditarietà OO anche se la sua scrittura è molto semplificata grazie all'uso di '->'. CoffeeScript rimane comunque molto legato a JavaScript e questo lo rende comprensibile soprattutto ai veterani di questo linguaggio.

Dart è invece un'ottima alternativa per coloro che hanno una predilizione per i linguaggi Object-oriented come Java. Inoltre permette una maggiore uniformità nelle applicazioni sia lato client che server senza la necessità di studiare due tecnologie diverse. Rimane comunque il fatto che Dart sia un linguaggio del tutto nuovo e non tutte le funzionalità siano state sviluppate. Nel futuro però potrebbe sostituire JavaScript proprio per la sua versatilità.

Bibliografia

[1] W3C - Worl Wide Web Consortium. http://www.w3.org/.

[2] Bob Nystrom. Idiomatic Dart. http://www.dartlang.org/, March 2012.

[3] Gilad Bracha. Optional Types in Dart. http://www.dartlang.org/, October 2011.

[4] Dartlang. http://www.dartlang.org/.

[5] Sigmund Cherem and Vijay Menon. Embedding Dart in HTML. http://www.dartlang.org/, October 2011.

[6] ECMA International. http://www.ecma-international.org/.

[7] jquery - write less, do more.l. http://jquery.com/.

[8] node.js. http://nodejs.org/.

[9] Wikipedia, l'enciclopedia libera. http://it.wikipedia.org.

[10] Mozilla developer network. http://developer.mozilla.org/it/.

[11] w3cschools. http://www.w3schools.com/.

[12] Html.it. http://www.html.it/.

[13] Html5 today. http://www.html5today.it/.

[14] Coffeescript. http://coffeescript.org/.