Javascript Patterns:

Dalle basi dell'OO
fino alle Tecniche e ai Pattern Avanzati

Motivazioni

- Web 2.0 in generale
  - Pattern OO fondamentali per applicazioni avanzate e complesse
  - Potere delle Open API (Facebook, Google, Amazon, Yahooo....)
- Efficienza
  - Quando possibile la logica si sposta lato client
  - Conseguente necessità dei principi e pattern dell'ingegneria del software e dell'OO
- Sicurezza (veramente molto sottovalutata)
  - Applicazioni web vulnerabili possono causare il furto di dati sensibili
  - Le applicazioni web fanno un uso intensivo di librerie di terze parti
  - E' bene proteggere i propri dati da eventuali vulnerabilità scoperte nel codice che non è direttamente sotto il nostro controllo

Sommario

- Alcuni concetti introduttivi: l'essenza del linguaggio javascript
- Gli Oggetti
  - Modalità di creazione
  - OO Pattern:
    - Metodi – Classi – Condivisione – Incapsulamento
  - Ereditarietà
- Pattern Avanzati
  - Le funzioni anonime
  - Module Pattern
  - Revealing Module Pattern
  - Come creare librerie
  - Revealing Prototype Pattern
- Conclusioni
  - Singleton: un rapido esempio
  - Referenze

Alcuni concetti introduttivi

- Javascript rappresenta l'implementazione dello standard internazionale ECMAScript
- Dalle referenze ufficiali ECMA:

  " ECMAScript is object-based: basic language and host facilities are provided by objects, and an ECMAScript program is a cluster of communicating objects. An ECMAScript object is a collection of properties. Properties are containers that hold other objects, primitive values, or functions. "
- Javascript NON è un linguaggio funzionale, TUTTAVIA:
  - Sono presenti alcune *feature* caratteristiche dei linguaggi funzionali
    - Funzioni trattate come oggetti di prima classe:
      - Possono essere passate come argomento ad altre funzioni, ritornare come valore di una funzione, essere assegnate ad una variabile o una struttura dati
  - Effettivamente le funzioni sono un meccanismo chiave:
    - Ma vengono comunque trattate come OGGETTI
    - Ogni funzione è un oggetto Function

Alcuni concetti introduttivi (cont...)

```
function double(x){ return 2*x }
double(2)                          // 4
typeof double                      // "function"
typeof double(2)                   // "number"
double instanceof Object           // true
double instanceof Function         // true

DOUBLE = new Function("x","return 2*x")
DOUBLE(2)                          // 4
typeof DOUBLE                      // "function"
typeof DOUBLE(2)                   // "number"
DOUBLE instanceof Object           // true
DOUBLE instanceof Function         // true

DOUBLE === double          // false → OGGETTI DIVERSI
double(2) === DOUBLE(2)    // true  → 4 == 4  "number"=="number"
```

Oggetti: creazione mediante letterali

- Dalle referenze ufficiali ECMA:

  " ECMAScript does not use classes such as those in C++, Smalltalk, or Java. Instead objects may be created in various ways including via a literal notation..... "

```
oggetto = {
        proprieta1 : "stringa" ,    // Literal Notation
        proprieta2 : 4
}

typeof oggetto                      // "object"
oggetto instanceof Object           // true
oggetto instanceof Function         // false
oggetto.proprieta1                  // "stringa"
oggetto.proprieta2                  // 4

JSON.stringify(oggetto)             // "{"proprieta1":"stringa","proprieta2":4} "
```

Oggetti: creazione mediante costruttori

- Continua dalle referenze ECMA:

  " .....or via constructors which create objects and then execute code that initialises all or part of them by assigning initial values to their properties. Each constructor is a function..... . Objects are created by using constructors in new expressions... ."

```
ogg = function() {
        this.proprieta1 = "stringa" ;
        this.proprieta2 = 4
 }

typeof ogg                          // "function"  → COSTRUTTORE
ogg instanceof Object               // true
ogg instanceof Function             // true

OGG = new ogg()                     // ogg {proprieta1: "stringa", proprieta2: 4}
OGG.proprieta1                      // "stringa"
OGG.proprieta2                      // 4

typeof OGG                          // "object"
OGG instanceof Function             // false
OGG instanceof ogg                  // true
```

Oggetti: costruttori (cont.) + stravaganze

```
JSON.stringify(OGG)                    // "{"proprieta1":"stringa","proprieta2":4}"

OGG === oggetto                                          //   false
JSON.stringify(OGG) === JSON.stringify(oggetto)          //   true
```

- Continua dalle referenze ECMA:
  - *" Invoking a constructor without using new has consequences that depend on the constructor. "*

```
ogg()                          // undefined

ogg = function() {
        this.proprieta1 = "stringa";
        this.proprieta2 = 4;
        return "qualcosa"
}

ogg()                          // "qualcosa"
OGG = new ogg()                // ogg {proprieta1: "stringa", proprieta2: 4}
```

Oggetti: e i metodi ?

- Ricordo: Funzioni trattate come oggetti di prima classe
  - Possiamo assegnarle a strutture dati

```
ogg = function() {
        this.proprieta1 = "stringa";
        this.proprieta2 = 4;
        this.addp1 = function(st){
                this.proprieta1 += st;
                return this.proprieta1;
        }
        this.doublep2 = function(){
                this.proprieta2 *= 2;
                return this.proprieta2;
        }
}

OGG = new ogg()
        //ogg {proprieta1: "stringa", proprieta2: 4, addp1: function, doublep2: function}

OGG.addp1(" concatenata")       // "stringa concatenata"
OGG.doublep2()                  // 8
```

Un utile esempio: emulare le classi

- Con gli esempi mostrati è possibile emulare in modo semplice le classi pur non avendo un esplicita sintassi e/o supporto
  - È bene usare la convenzione dei nomi che iniziano con la lettere maiuscola per quelle funzioni che vengono usate come costruttori

```
function Car( model, year, miles ) {              // Car = function(model.....
        this.model = model;
        this.year = year;
        this.miles = miles;

        this.toString = function () {
                return this.model + " has done " + this.miles + " miles";
        };
}

var civic = new Car( "Honda Civic", 2009, 20000 );
var mondeo = new Car( "Ford Mondeo", 2010, 5000 );

mondeo.toString()          // "Ford Mondeo has done 5000 miles"
```

- NB: le funzioni e le variabili definite con var sono visibili nello scope corrente

Oggetti: e i vantaggi delle vere classi?

- ANCORA: Javascript non offre le CLASSI
  - Secondo il principio appena visto, ogni istanza del costruttore contiene la definizione delle funzioni trattate come metodi
  - Non c'è condivisione dei metodi tra le istanze come in Java
  - La memoria può diventare un problema

- Come fare ? Possibile che il linguaggio non offra meccanismi alternativi a supporto di tali problemi e/o caratteristiche ?
  - SOLUZIONE → PROTOTYPE PATTERN

Finalmente la definizione completa:

" ECMAScript does not use classes such as those in C++, Smalltalk, or Java. Instead objects may be created in various ways including via a literal notation or via constructors which create objects and then execute code that initialises all or part of them by assigning initial values to their properties. Each constructor is a function that has a property named ***prototype*** *that is used to implement prototype-based inheritance and shared properties. Objects are created by using constructors in new expressions. Invoking a constructor without using new has consequences that depend on the constructor. "*

La proprietà Prototype

“ Each constructor is a function that has a property named “prototype” that is used to implement prototype-based inheritance and shared properties. “

```
function Car( model, year, miles ) {
        this.model = model;
        this.year = year;
        this.miles = miles;
}

Car.prototype.toString = function () {
        return this.model + " has done " + this.miles + " miles”;
};

var civic = new Car( "Honda Civic", 2009, 20000 );
var mondeo = new Car( "Ford Mondeo", 2010, 5000 );

civic.toString();                // “Honda Civic has done 20000 miles “
```

- Ogni oggetto istanza di Car condivide il metodo toString
- E’ possibile notare le differenze dei due approcci usando la console del browser

Prototype: proprietà condivise

```
function Interruttore(nome){ this.nome = nome }

Interruttore.prototype.stato = "spento”
Interruttore.prototype.toString = function(){
        return "Interruttore " + this.nome +" " +this.stato
}

lampada = new Interruttore("lampada”)
lampada.toString()                  // "Interruttore lampada spento”

condizionatore = new Interruttore("condizionatore”)
condizionatore.toString()           // "Interruttore condizionatore spento”

Interruttore.prototype.stato = "acceso”
lampada.toString()                  // "Interruttore lampada acceso”
condizionatore.toString()           // "Interruttore condizionatore acceso”

lampada.stato = "guasto"
lampada.toString()                  // "Interruttore lampada guasto”
condizionatore.toString()           // "Interruttore condizionatore acceso"
```

Prototype: un contatore di istanze

- Ancora un ESEMPIO:

```
function Persona(nome){
        this.nome = nome;
        Persona.prototype.count++;
}

Persona.prototype.count = 0

Andrea = new Persona("Andrea")
Andrea.count                                    // 1

Marco = new Persona("Marco")
Marco.count                                     // 2

Andrea.count                                    // 2
```

- Come estensione, si potrebbe far eseguire ad ogni persona diversi compiti in base al numero totale di persone disponibili

- NB: La proprietà prototype contiene un oggetto

Prototype: è adatto in tutti i casi?

- Non vi siete accorti di nulla? Come sono le proprietà degli oggetti visti nei precedenti esempi ? → SONO PUBBLICHE
  - Tutte le proprietà definite con il this sono accessibili /modificabili
  - E se ci fosse il bisogno di avere variabili private accessibili / modificabili solo da appropriati metodi ? Ad esempio per valori che devono essere controllati prima di essere assegnati a proprietà (ad esempio mediante RegExp)

```
function BadClass(){ var private = "privata" }

BadClass.prototype.getPrivate = function() {return private }
BadClass.prototype.setPrivate = function(val) { private = val }

badobj = new BadClass()
badobj.getPrivate()                    // ReferenceError: private is not defined

badobj.setPrivate("cambio privato")    // nessun errore
badobj.getPrivate()                    // "cambio privato"
```

- !!!!!!!!!! ATTENZIONE !!!!!!!!!! → Qual'è lo scope dei metodi get/set (window)

```
window.private                         // "cambio privato"
```

Prototype: attenti agli orrori !!!!!

- Perchè non inserire i metodi prototype nello scope della variabile privata ?

```
function BadClass(){
        var private = "privata";
        BadClass.prototype.getPrivate = function() {return private };
        BadClass.prototype.setPrivate = function(val) { private = val };
}

badobj_1 = new BadClass();
badobj_1.getPrivate()                   // "privata"
badobj_1.setPrivate("1 - privata");
badobj_1.getPrivate()                   // "1 – privata"

badobj_2 = new BadClass();
badobj_2.getPrivate();                  // "privata"
badobj_1.getPrivate();                  // "privata"

window.private                          // undefined

badobj_2.setPrivate("2 - privata");
badobj_1.getPrivate();                  // "2 : privata"
badobj_2.getPrivate();                  // "2 : privata"
```

Variabili private: la giusta soluzione

- L'ambiente dell'oggetto prototype è condiviso con tutte le istanze
- Se vogliamo agire sulle variabili private l'unico modo è usare metodi in this, ovvero creare una chiusura

  " A closure is a special kind of object that combines two things: a function, and the environment in which that function was created. The environment consists of any local variables that were in-scope at the time that the closure was created."

```
function CorrectClass(){
        var x = 7;
        this.GetX = function() { return x;}
        this.SetX = function(val) { x = val; }
}

obj = new CorrectClass();
obj2 = new CorrectClass();

console.log(obj.GetX() + ' ' + obj2.GetX());     // 7 7

obj.SetX(14);

console.log(obj.GetX() + ' ' + obj2.GetX());     // 14 7
```

Troppi concetti ? Mettiamoli insieme !!!

```
function Person(name,age,descrizione){

        // metodi privati
        function checkAge(age){
                if(typeof age != "number" || age<0) return false;
                return true
        }
        function checkName(name){
                if(typeof name != "string" || name === "" ) return false
                return true
        }

        if(!checkAge(age)) throw new Error("Invalid Age")
        if(!checkName(name)) throw new Error("Empty Name")

        // proprietà private
        var _name = name;
        var _age= age;

        // proprietà pubblica
        this.descrizione = descrizione;

        // proprietà condivisa da tutte le istanze
        Person.prototype.count ? Person.prototype.count++ : Person.prototype.count= 1 ;
```

Troppi concetti ? Mettiamoli insieme !!!

```
        // metodi privilegiati
        this.updateAge = function(age){
                if(!checkAge(age)) throw new Error("Invalid Age")
                _age = age;
        }

        this.getAge = function(){
                return _age
        }
        this.getName = function(){
                return _name;
        }

}
// metodo pubblico
Person.prototype.toString = function(){
      return "Sono "+this.getName()+" e ho "+this.getAge()+" anni. " + this.descrizione;
}

mario = new Person("Mario",10, "Ciao a tutti");
mario.toString()                //"Sono Mario e ho 10 anni. Ciao a tutti"
mario.updateAge(15)
mario.toString()                //"Sono Mario e ho 15 anni. Ciao a tutti"
```

Ereditarietà: il pattern corretto

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

Person.prototype.sayName = function(){ return "Mi chiamo "+this.name }

function Student(name,number){
        Person.call(this,name);
        this.number = number
}

Student.prototype = new Person()              // erase del costruttore di Student
Student.prototype.constructor = Student       //  → best practice

Student.prototype.sayInfo = function(){
        return this.sayName()+ ", matricola: "+this.number
}

marco = new Student("Marco",1234)
marco.sayInfo()              // "Mi chiamo Marco, matricola: 1234"
```

- Altri approcci usano la copia delle proprietà: si perde la potenza del prototype

Le Funzioni Anonime

- è un costrutto che può sembrare strano ma ha una suo perché...
  - Prima definisco una funzione anonima (non ha un nome)
  - Poi la eseguo

```
( function(){
        console.log("prova")                    // codice
}) ()

( function(val1,val2){
        console.log(val1 + ":" + val2)          // codice
}) (val1,val2)
```

- Estremamente semplice
- Apparentemente inutile
- Praticamente fondamentale: soprattutto assegnandogli un nome

Module Pattern: un esempio immediato

```
var Module = (function () {
        var myPrivateVar, myPrivateMethod;
        myPrivateVar = 0;
        myPrivateMethod = function( v) { console.log( v); };

        return {
                myPublicVar: ”public”,
                myPublicFunction: function( v) {
                        myPrivateVar++;
                        myPrivateMethod( v);
                 }
        };
}) ();

Module.myPublicVar = “ciao”
Module.myPublicFunction(“hello”)    // stampa “hello” sulla console
```

Module Pattern: considerazioni

- Utilizzo delle chiusure in maniera intelligente
- Incapsula un mix di variabili/metodi pubblici e privati
- Evita collissioni di namespace: variabili/funzioni globali
- Fornisce un API di accesso alla chiusura del modulo
- Rende possibile la creazione di namespace personali
- SICUREZZA !!!!!! Variabili e metodi privati non accessibili
- Nel Module Pattern classico:
  - Tutto ciò che si vuole rendere pubblico deve essere definito nell'oggetto che viene restituito dall'esecuzione del modulo
  - In generale la parte privata non può accedere alla parte pubblica
  - Sono possibili anche strategie di strutturazione differenti ovviamente con pregi e difetti

Revealing Module Pattern

```
var myRevealingModule = function () {

        // tutte funzioni / variabili private
        var privateCounter = 0;
        function privateFunction() { privateCounter++; }
        function publicStartFunction() { publicIncrement(); }
        function publicIncrement() { privateFunction(); }
        function publicGetCount(){  return privateCounter; }

        // rivelo all'esterno quello che voglio
        return {
                start: publicStartFunction,
                increment: publicIncrement,
                count: publicGetCount
        };
}();

myRevealingModule.start();
myRevealingModule.count()                    // 1
myRevealingModule.increment()
myRevealingModule.count()                    // 2
```

Revealing Module Pattern: considerazioni

- Vantaggi:
  - Sintassi più consistente
  - Rende più chiare quali sono le variabili/funzioni accessibili al pubblico
- Svantaggi:
  - Nel module pattern le proprietà pubbliche possono essere aggiornate sia esternamente al modulo che a run-time
  - Nel revealing module pattern un aggiornamento di una proprietà pubblica può portare effetti indesiderati
    - Se una funzione privata (NB: che non viene resa pubblica) usa una funzione pubblica (NB: che viene poi resa pubblica ma è effettivamente privata) la prima continuerà ad eseguire la versione privata della seconda nonostante l'aggiornamento della funzione pubblica
    - Stesse considerazioni in generale per oggetti e variabili

Creare librerie: un semplice esempio

```
Var Mylib = (function(){

        var Module1 = (function(){
                ...
        })()

        var Moudle2 = (function(){
                ...
        })()

        ...

        return {
                Module1 : Module1,
                Module2 : Module2
                ...
        }
})();

Mylib.Module3 = ( function(){  ...  } )();

Mylib.Module1.func1();
Mylib.Module2.func1();
```

Alcune considerazioni

- Moudule Pattern e Revealing Module Pattern sono solo il punto di partenza per la costruzione e strutturazione di librerie manutenibili, estendibili e sicure.
- Possiamo applicare a questi Pattern i principi fondamentali (visti nelle slide precedenti) in particolare sfruttando la potenza del prototype
- Molti pattern sono stati studiati come estensioni di quelli visti fino a questo punto: Revealing Prototype Pattern ad esempio.
- I pattern comuni possono essere realizzati senza problemi: magari riadattandoli al principio dei prototipi di javascript
- Una volta padroneggiati i concetti presentati in queste slide, l'unico limite è la fantasia.

Revealing Prototype Pattern

```
var Class = function(p1,p2){
        this.p1 = p1
        this.p2 = p2
}

Class.prototype = ( function(){

        var private_var = "some private value”

        var public_var = "some public value”

        var privateFunction = function() { }

        var publicFunction = function() { }


        return{
                publicF1 : publicFunction,
                publicV1 : public_var
        }
})()

Class.prototype.constructor = Class
```

Un esempio: Singleton

```
var mySingleton = (function () {

        var instance;

        function init() {
                function privateMethod(){
                        console.log( "I am private" );
                }
                var privateVariable = "Im also private“;
                var RandomNumber = Math.random();
                return {
                        publicMethod: function () {
                                console.log( "The public can see me!" );
                        },
                        publicProperty: "I am also public”,
                        getRandomNumber: function() {
                                return RandomNumber;
                        }
                };
        };
```

Un esempio: Singleton (cont...)

```
        return {
                getInstance: function () {
                        if ( !instance ) {
                                instance = init();
                        }
                        return instance;
                }
        };
})();

var instance = mySingleton.getInstance();

instance.publicMethod();
console.log(instance.publicProperty)
```

Riferimenti

- *Learning JavaScript Design Patterns* : http://addyosmani.com/resources/essentialjsdesignpatterns/book/
  - Altamente consigliato: contiene una raccolta completa dei pattern comuni (compresi quelli visti nel corso), oltre ad una serie di nozioni e tecniche molto utili
- *Standard ECMA-262*
- *Introduction to Object-Oriented JavaScript*: https://developer.mozilla.org/en-US/docs/JavaScript/Introduction_to_Object-Oriented_JavaScript
- *Revealing Prototype Pattern - Techniques, Strategies and Patterns for Structuring JavaScript Code*: http://weblogs.asp.net/dwahlin/archive/2011/08/03/techniques-strategies-and-patterns-for-structuring-javascript-code-revealing-prototype-pattern.aspx