# Typescript: organizzare il codice con interfacce, classi e moduli

<http://blog.boschin.it/post/2013/04/03/Typescript-organizzare-il-codice-con-interfacce-classi-e-moduli.aspx>

Pur se lo static type checking è un elemento importante nell'utilizzo di Typescript, tanto da essere già una ragione più che valide nella sua adozione, è chiaro che si può fare di più nella semplificazione del codice Javascript e nella sua organizzazione. Nel corso degli anni si sono consolidate alcune pratiche comuni nella stesura di Javascript, per venire incontro alla carenza dei concetti di base dell'object orientation che sono particolarmente labili e limitati. Tra essi si possono annoverare alcune naming convention - ad esempio prependere un underscore ad una variabile la identifica come privata - e dei veri e propri pattern quali le closure. Tutto ciò in effetti è più che altro una notazione stilistica che una vera e propria caratteristica del linguaggio e spesso e volentieri tali pratiche tendono a renderlo di difficile comprensione e manutenzione. Typescript ha tra i suoi costrutti alcuni che hanno proprio lo scopo di semplificare nettamente il codice mascherando la vera e propria complessità che origina dalla sua compilazione.

Interfacce e classi

Nel precedente articolo abbiamo visto che grazie allo structured typing, Typescript è in grado di riconoscere i tipi semplicemente analizzando la corrispondenza delle proprietà. L'esempio seguente riassume in breve la questione:

1: function getArea(s: { width: number; height: number; }): number

2: {

3: return s.width \* s.height / 2;

4: }

5:

6: var area = getArea({ width: 20, height: 30 });

7: console.log(area.toString());

E' del tutto chiaro che questo tipo di notazione, pur se tutelata dal compilatore, è eccessivamente prolissa e a lungo andare puà essere oggetto di errori e incomprensioni. Per questa ragione Typescript ammette innanzitutto la creazione di interfacce, che altro non sono che la definizione del contratto di un tipo cui il valore deve conformarsi. un concetto più che normale per chi mastica programmazione ad oggetti. Vediamo un esempio:

1: interface Shape

2: {

3: width: number;

4: height: number;

5: }

6:

7: function getArea(s: Shape): number

8: {

9: return s.width \* s.height / 2;

10: }

11:

12: var area = getArea({ width: 20, height: 30 });

13: console.log(area.toString());

Nelle prime righe dello snippet è visibile la dichiarazione di una interfaccia "Shape". Essa riporta le proprietà "width" e "height" e viene usata come parametro della funzione getArea, al posto della precedente notazione estesa. L'argomento passato alla getArea è rimasto invariato, ma se provate a modificarne i nomi vi renderete contro che lo structured typing continua a funzionare come prima validando il parametro contro la definizione dell'intefaccia. Le interfacce ammettono oltre alle proprietà anche metodi e proprietà opzionali (usando il ?). Ecco un esempio:

1: interface Shape

2: {

3: width: number;

4: height: number;

5: color?: string;

6: getArea(): number;

7: }

8:

9: function getArea(s: Shape): number

10: {

11: return s.getArea();

12: }

13:

14: var area = getArea(

15: {

16: width: 20,

17: height: 30,

18: getArea: function() { return this.width \* this.height / 2; }

19: });

20:

21: console.log(area.toString());

L'esempio in questione estremizza la definizione dell'interfaccia Shape richiedendo una proprietà opzionale "color" (che poi non viene passata più sotto) e un metodo che effettua il calcolo dell'area. In tal modo la funzione getArea non deve fare altro che chiamare l'omonimo metodo dell'interfaccia per ottenere il valore calcolato. Si tratta di un primo passo verso una conversione object oriented dell'esempio. A questo punto potremmo voler implementare l'interfaccia Shape in diverse figure e fornire una diversa implementazione del calcolo. Lo possiamo fare grazie alla presenza delle classi. Vediamo come:

1: interface Shape

2: {

3: width: number;

4: height: number;

5: getArea(): number;

6: }

7:

8: class Triangle implements Shape

9: {

10: constructor(

11: public width: number,

12: public height: number) { }

13:

14: getArea(): number {

15: return this.width \* this.height / 2;

16: }

17: }

18:

19: class Square implements Shape

20: {

21: constructor(

22: public width: number,

23: public height: number) { }

24:

25: getArea(): number {

26: return this.width \* this.height;

27: }

28: }

29:

30: function getArea(s: Shape): void

31: {

32: var area = s.getArea();

33: console.log(area.toString());

34: }

35:

36: getArea(

37: new Square(20, 30));

38: getArea(

39: new Triangle(20, 30));

Grazie alla keyword "class" è possibile creare delle vere e proprie classi che, a differenza di quello che succede per le interfacce che spariscono nel codice Javascript, generano una struttura che chi è avvezzo alle strutture tipiche di Javascript riconoscerà sicuramente.

1: var Triangle = (function () {

2: function Triangle(width, height) {

3: this.width = width;

4: this.height = height;

5: }

6: Triangle.prototype.getArea = function () {

7: return this.width \* this.height / 2;

8: };

9: return Triangle;

10: })();

Nel precedente snippet abbiamo anche la dimostrazione che una classe può implementare una specifica interfaccia, per mezzo della keyword "implements" e così facendo il compilatore verificherà a tempo di compilazione che la classe supporti i metodi e le proprietà da essa richieste. Siamo a questo punto arrivati ad una programmazione ad oggetti del tutto raffinata che poco ha a che vedere con la complessità cui si è abituati con Javascript, totalment mascherata dal compilatore Typescript.

Oltre all'implementazione di interfacce è del tutto possibile estendere classi esistenti. Per fare questo dovremo utilizzare la keywork "extends" al posto di "implements". Vediamo come usare l'ereditarietà per create una classe "Cube" derivando da "Square":

1: class Cube

2: extends Square

3: {

4: constructor(

5: width: number,

6: height: number,

7: public depth: number)

8: {

9: super(width, height);

10: }

11:

12: getArea(): number

13: {

14: return (super.getArea() \* 2) +

15: (this.depth \* this.width \* 2) +

16: (this.depth \* this.height \* 2);

17: }

18: }

In questo esempio vediamo che al costruttore della classe viene aggiunto un ulteriore parametro "depth" che identifica l'altezza del parallelepipedo. Avendo modiificato la firma il compilatore richiede che la prima chiamata nel body del costruttore sia la funzione "super" che ha lo scopo di chiamare il costruttore della classe base. Questa deve essere specificata come faremmo usando "base" in C#. La medesima keyword può essere usata anche per chiamare i metodi della classe base. Ad esempio il metodo getArea richiama l'omonimo della classe base per poi sfruttare il risultato integrando la rimanente parte dell'area.

Usare i moduli

Una volta che abbiamo classi e interfacce i benefici che ne derivano sono numerosi, soprattutto in termini di organizzazione logica del codice e di manutenzione. Il passo successivo è di organizzare il codice in moduli - i programmatori c# li conosceranno meglio come "namespace" - per riuscire a creare vere e proprie librerie i cui nomi siano univoci. Anche in questo Typescript ci aiuta; grazie alla keywork "module" infatti sarà possibile creare dei veri e propri namespace:

1: module Shapes

2: {

3: export class Square implements Shape

4: {

5: constructor(

6: public width: number,

7: public height: number) { }

8:

9: getArea(): number

10: {

11: return this.width \* this.height;

12: }

13: }

14: }

Interessante notare che la classe definita nel modulo "Shapes" è stata decorata con "export". Infatti, una volta che abbiamo messo una classe (o qualunque altro costrutto) in un modulo possiamo renderlo visibile o invisibile all'esterno beneficiando di un incapsulamento che in termini di librerie riutilizzabili è prezioso.

Come si è abituati a fare con i namespace in C#, anche in Typescript i moduli possono essere annidati in modo del tutto analogo, creandole effettivamente l'uno nell'altro:

1: module Shapes

2: {

3: export class Square implements Shape

4: {

5: constructor(

6: public width: number,

7: public height: number) { }

8:

9: getArea(): number

10: {

11: return this.width \* this.height;

12: }

13: }

14:

15: export module ThreeD

16: {

17: export class Cube extends Square

18: {

19: // ... omissis

20: }

21: }

22: }

Oppure usando una notazione puntata

1: module Shapes.ThreeD

2: {

3: export class Cube extends Square

4: {

5: // ... omissis

6: }

7: }

Ciascuna delle due notazioni può essere tranquillamente utilizzata assieme all'altra creando vere e proprie composizioni in cui i moduli si combinano. Una volta che i moduli sono stati creati sarà possibile raggiungere i tipi definiti nei moduli specificando l'intero namespace:

1: var square: Shapes.Square;

2: var cube: Shapes.ThreeD.Cube;

Data la notevole lunghezza e ridondanza che i nomi completi di namespace possono raggiungere è del tutto possibile creare degli shortcut che siano in grado di semplificare la scrittura del codice:

1: import sh = Shapes;

2: import sh3d = Shapes.ThreeD;

3:

4: var square: sh.Square;

5: var cube: sh3d.Cube;

Come a casa propria

Sono convinto che i programmatori C#, ma in generale qualunque sviluppatore sia avvezzo all'uso di un linguaggio evoluto basato sui paradigmi della programmazione ad oggetti, leggendo il codice Typescript si senta bene come a casa propria. In effetti se si da uno sguardo veloce al codice generato dal compilatore si comprende come Typescript sia in grado di fornire strumenti che la programmazione Javascript può dare solo a caro prezzo. Per intenderci vediamo un esempio di cosa sia possibile fare:

1: module Shapes

2: {

3: export interface Shape

4: {

5: width: number;

6: height: number;

7: getArea(): number;

8: }

9:

10: export enum ShapeType

11: {

12: Square,

13: Triangle

14: }

15:

16: export class ShapeFactory

17: {

18: static create(type: ShapeType, width: number, height: number): Shape

19: {

20: switch (type)

21: {

22: case ShapeType.Square:

23: return new Square(width, height);

24: case ShapeType.Triangle:

25: return new Triangle(width, height);

26: }

27:

28: return null;

29: }

30: }

31:

32: class Triangle implements Shape

33: {

34: constructor(

35: public width: number,

36: public height: number) { }

37:

38: getArea(): number

39: {

40: return this.width \* this.height / 2;

41: }

42: }

43:

44: class Square implements Shape

45: {

46: constructor(

47: public width: number,

48: public height: number) { }

49:

50: getArea(): number

51: {

52: return this.width \* this.height;

53: }

54: }

55: }

56:

57: import sh = Shapes;

58: var sq = sh.ShapeFactory.create(sh.ShapeType.Square, 20, 30);

59: console.log(sq.getArea());

Credo che la combinazione di moduli, classi, interfacce ed enumeratori di questo esempio, assieme con l'applicazione di metodi statici e dell'incapsulamento nei moduli sia molto significativa di come si possa scrivere con proprietà un codice molto efficace.