TYPESCRIPT

appunti di un programmatore per programmatori

CHRISTIAN PALAZZO

APPUNTI DI TYPESCRIPT

Indice

0. Introduzione

Prerequisiti

TypeScript è un super set di JavaScript, aggiunge nuove funzionalità al linguaggio, bisogna quindi avere familiarità con JavaScript.

Introdurre TypeScript

TypeScript è un super set di JavaScript. Vediamo la differenza tra linguaggi statici e dinamici: entrambi hanno dei tipi fondamentali, Ma mentre nel linguaggio dinamico se commettiamo errori questi li vediamo solo a runtime, con il linguaggio statico, più rigido sull'assegnazione dei tipi, e si ha la possibilità di trovare l'errore in fase di sviluppo prima di andare a runtime, questo perché in fase di dichiarazione della variabile si specifica il tipo, e il tool di sviluppo di segnala l'errore prima di far girare il codice.
Sebbene il codice dinamico è molto valido per l'object model dei web browser, il linguaggio statico migliora la stabilità e la manutenibilità del codice.
JavaScript è un linguaggio dinamico, TypeScript è statico, dato che TypeScript transpila in JavaScript, usiamo il linguaggio statico in fase di sviluppo e il linguaggio dinamico a runtime. Sebbene la tipizzazione forte sia caldamente raccomandata in TypeScript, è anche possibile sviluppare a tipizzazione debole.

Definire JavaScript

Anche se questo è un corso su TypeScript, è bene definire cosa sia JavaScript.
JavaScript nasce nel 1996 con il broswer NETScapes, nel 1997 viene definita una specifica del linguaggio da ECMAScript. Nel corso del tempo la ECMAScript ha rilasciato varie specifiche del linguaggio. Ad oggi la versione rilasciata è ES6. I vari browser via via implementano le diverse funzionalità quindi questo vuol dire che i diversi browser implementano le diverse funzionalità in tempi diversi.
Il vantaggio di utilizzare i transpilatori è quello di utilizzare le nuove funzionalità da subito e transpilare in una versione di JavaScript implementata da tutti i browser.

Scrivere la prima funzione TypeScript
Sul sito ufficiale di TypeScript:

http://www.TypeScriptlang.org

è presente un editor online con un syntax highlighting per TypeScript che transpila in JavaScript. Scriviamo una funzione in TypeScript dichiarando il tipo:

```
function speak(value: string): string {
        document.write(value);
        return value;
}
```

scrivendo in questo modo, dichiariamo il tipo del parametro value della funzione e anche il tipo della variabile restituita dalla funzione.

1. Configurare il tuo ambiente

Scegliere l'editor per TypeScript

Vanno bene gli stessi tool utilizzati per lo sviluppo in JavaScript, è un linguaggio diffuso quindi esistono plugin per gli editor più importanti ed è integrato nel IDE più importanti.
Il transpilatore di TypeScript si può utilizzare come plugin all'interno di un IDE come ad esempio Visual Studio oppure si può installare come pacchetto di sistema tramite npm.

Installare TypeScript in Visual Studio

Ci sono varie versioni di Visual Studio e ciascuna di esse offre un diverso supporto a TypeScript. Con Visual Studio 2013 TypeScript è installabile come plugin dell'IDE. L'ultima release di Visual Studio supporta TypeScript nativamente, ma è una versione del linguaggio un po' datata quindi si consiglia di installare l'ultima versione del plugin disponibile.

Installare TypeScript da riga di comando

Molti utenti preferiscono la riga di comando, o piuttosto si vuole il tranpilatore solamente, senza installare un pesante IDE. TypeScript viene distribuito come modulo npm, per cui bisogna installare node.js e npm e dare l'istruzione per l'installazione del modulo globale TypeScript. L'installazione offre il comando tsc.
Si consiglia di installare anche il server lite-server per il watch del codice sorgente di TypeScript e la transpilazione e refresh del codice automatizzati.
Lite server si installa come modulo node globale.

Creare un progetto TypeScript

TypeScript può essere utilizzato ovunque viene utilizzato JavaScript, sia questo un client o un server. Comunque la maggior parte delle volte, TypeScript verrà utilizzato per produrre applicazioni web.
Il progetto inizia con il creare la root directory. Si crea l'index.html popolandolo con un po' di markup. Si inserisce uno script tag che punta a un file app.js.

Si crea il file source app.ts.
Il modo più semplice per transpilare file TypeScript è quello da riga di comando con il comando tsc, che ha un flag -w per rimanere in watch sul file.
TypeScript legge un file JSON di configurazione tsconfig.json.
Vediamo come può essere editato:

```
{
        “compilarerOptions”: {
        “target”: “es5”
        }
}
```

In questa istruzione diciamo al transpilatore in che versione di JavaScript transpilare.

2. Funzionalità del linguaggio ES6

Panoramica delle funzionalità ES6

TypeScript permette di utilizzare le funzionalità ES6. Vediamo le funzionalità più importanti dello standard in questo capitolo, si tratta di syntactic sugar.

Parametri di default

Le funzioni accettano parametri di default valorizzati.

```
function countDown (initial, final=0, interval=1) {
        //logic
}
```

In questo modo si può passare solo il primo parametro, gli altri sono valorizzati di default. Se invece vengono passati, i parametri di default vengono sovrascritti.

Template string

Si usa per inserire variabili all'interno di stringhe.

```
var displayName = `Todo #${todo.id}`;
```

E' utile quando si utilizzano porzioni di html come template stringhe. Nei template stringhe si può andare a capo.
All'interno delle parentesi si possono inserire espressioni come ad esempio i condizionali o gli operatori ternari.

Let e const

Sono delle dichiarazioni per la definizione di variabili. Prendiamo l'esempio di una variabile contatore dichiarata per un ciclo for.

```
for (var i = 0; x <= 5; x++) {
        var counter = x;
}
```

```
console.log(counter);
```

in questo caso la variabile counter è accessibile al console.log, perché in JavaScript lo scope delle variabili è definito a livello di funzione e non a livello di blocco. Questo comportamento è stato modificato con la dichiarazione let che introduce lo scope a livello di blocco in JavaSCript.
Const si usa per definire costanti.
let si può utilizzare in molte occasioni, allo stesso modo in cui si utilizza var in ES5.

Cicli for of
Si utilizza per i loop di array od oggetti:

```
var array = [1,2,3,4,5,6];

for (var value of array) {

}
```

La differenza con il ciclo for classico è che questo cicla gli indici dell'array, il for of cicla direttamente i valori.
Anche questo è syntactic sugar in quanto in ES5 avevamo a disposizione il ciclo for in, ma in questo modo il codice risulta più leggibile.

Lambda
In JavaScript la parola riservata this introduce dei comportamenti inaspettati nel codice. Vediamo un esempio:

```
var container = document.getElementById('container');

function Counter(el) {
    this.count = 0;

    el.innerHTML = this.count;

    el.addEventListener('click', function () {
```

```
        this.count += 1;
        el.innerHTML = this.count;
        });
}

new Counter(container);
```

Mi aspetto che la funzione starti il contatore da zero e che ad ogni click venga incrementato di uno.
La funzione non va perché il listener è riferito allo scope del browser, non a quello della funzione Counter, pertanto otteniamo un Nan per this.count.
Per ovviare a questo bisogna aggiungere al listener un riferimento allo scope della funzione Counter. Ciò si può fare in vari modi, per esempio definendo una nuova variabile all'interno della funzione:

```
var contsainer = document.getElementById('container');

function Counter(el) {
        this.count = 0;

        el.innerHTML = this.count;

        let _this = this;

        el.addEventListener('click', function () {
        _this.count += 1;
        el.innerHTML = _this.count;
        });
}
```

ES6 introduce le arrow function che aiutano in questo tipo di situazioni. La sintassi è abbastanza semplice, in altri linguaggi vengono definite funzioni lambda:

```
var contsainer = document.getElementById('container');

function Counter(el) {
	this.count = 0;

	el.innerHTML = this.count;

	el.addEventListener('click', () => {
	this.count += 1;
	el.innerHTML = this.count;
	});
}
```

Se la funzione arrow è breve si possono omettere la parola return e le parentesi e mettere tutto su una riga.

Destructuring
E' una funzionalità che si usa meno spesso, comunque in alcune occasioni può risultare utile. Si tratta di un assegnamento multiplo usando degli elementi che si trovano in un array:

```
var array = [123, 'string', false];
var [id, title, completed] = array; //definisco 3 array
```

questo corrisponde a scrivere:

```
var array = [123, 'string', false];
var id = array[0], title = array[1], completed] = array[2];
```

Il beneficio di questa sintassi non è molto chiaro in questo esempio, vediamone un altro in cui è più conveniente utilizzare la funzionalità, cioè quando abbiamo bisogno di una variabile temporanea:

```
var a = 1;
var b = 5;

[a,b] = [b,a];
```

Il destructoring funziona anche con gli oggetti facendo match dei nomi delle proprietà con i nomi dei valori;

```
var todo = {
        id: 123,
        title: ‘string’,
        completed :false
}
var {id,title,completed} = todo;
```

l’ultima riga di codice corrisponde a scivere:

```
var id = todo.id, title = todo.title, completed = todo.completed;
```

Definiamo così delle nuove variabili i cui nomi sono le proprietà dell’oggetto e i valori le corrispondenti proprietà.
Il destructuring funziona anche quando le funzioni di mapping ritornano dei valori:

```
function getTodo (id) {
        var todo = {
        id: 123,
        title: ‘string’,
        completed :false
        }
        return todo;
}
var {id,title,completed} = getTodo(123);
```

L’uso più comune di questa funzionalità è quella di ridurre una lunga lista di parametri metodo in un singolo oggetto opzione che contiene tutti i parametri come proprietà:

```
function countDown (initial, final=0, interval=1) {
        var current = initial;

        while (current > final) {
```

```
        console.log(current);
        current -= interval;
        }
}
```

questa funzione ha 3 parametri, vediamo come usare il destructuring:

```
function countDown (options) {
        var options = options === undefined ? {} : options,
            initial = options.initial === undefined ? 10 : options.initial,
            final = options.final === undefined ? 0 : options.final
            interval = options.interval === undefined ? 1 : options.interval;

        var current = initial;

        while (current > final) {
        console.log(current);
        current -= interval;
        }
}
```

controllo ciascuna proprietà se è definita prima di assegnarla alla variabile. Applicando il destructuring direttamente all'oggetto options:

```
function countDown ({initial, final: final = 0, interval: interval = 0, initial:
current}) {
        while (current > final) {
        console.log(current);
        current -= interval;
        }
}
```

Alcuni programmatori considerano questa funzionalità molto utile e facile da manutenere, ad altri non piace, è comunque una funzionalità utile da sapere per utilizzarla se lo si vuole.

L'operatore spread

Anche questa funzionalità non è molto utilizzata, ma quando la si usa genera un codice molto elegante e bello da vedere:
Il caso d'uso è una funzione che prende un qualunque numero di argomenti. Prendiamo una funzione che prende queste variabili e le assegna a un array su cui iterare:

```
function add () {
	var values = Array.prototype.splice.call(arguments, [1]),
		total = 0;

	for (var value of values) {
	total += value;
	}

	return total;
}
```

Questo codice è lungo e stiamo chiamando un metodo dell'oggetto nativo Array. Vediamo lo spread operator:

```
function add (...values) {
	var total = 0;

	for (var value of values) {
	total += value;
	}

	return total;
}
```

Ogni parametro in numero indefinito viene passato come elemento del parametro values, un array. Oltre all'operatore spread possiamo passare altri parametri di tipo qualunque alla funzione. L'operatore spread può essere passato come elemento di un altro array, in questo caso otteniamo un nuovo array con tutti gli elementi dell'array dell'operatore spread. Possiamo anche usare i metodi con cui frequentemente lavoriamo gli array, come ad esempio push().

Proprietà computate
Poco utilizzata, ma utile se si vogliono sfruttare le proprietà dinamiche del linguaggio JavaScript. Prendiamo un oggetto:

```
var support = {
        ‘os_windows’: isSupported(‘Windows’),
        ‘os_iOs’: isSupported(‘iOS),
        ‘os_android’: isSupported(‘Android’)
};

function isSupported(os) {
        return Math.random() >= 0.5;
}
```

Permette di definire una proprietà di un oggetto con un nome computato dinamicamente a runtime.
Nell’esempio sopra abbiamo definito un oggetto che iniziano tutti con la substring os_ . Questa è una sintassi standard che voglio utilizzare per tutte le proprietà dell’oggetto:

```
const osPrefix = ‘os_’;

var support = {
        [osPrefix  + ‘windows’]: isSupported(‘Windows’),
        [osPrefix  + ‘iOs’]: isSupported(‘iOS),
        [osPrefix  + ‘android’]: isSupported(‘Android’)
};

function isSupported(os) {
        return Math.random() >= 0.5;
}
```

Questo esempio è abbastanza semplice, ma i casi di uso in cui utilizzare questa nuova funzionalità sono situazioni più complicate.

3. Tipi fondamentali

Introduzione ai tipi JavaScript

Iniziamo a illustrare tutti i tipi disponibili in JavaScript. L'ES5 definisce 5 tipi fondamentali: boolean, number e string sono immutabili, cioè una volta definiti non possono più essere modificati. Poi ci sono due tipi particolari: null / undefined e object. Il tipo oggetto può essere visto come una lista le cui chiavi sono i nomi delle proprietà. Le proprietà possono essere modificate, aggiunte e tolte dinamicamente.
Funzioni e array sono di tipo oggetto. Pertanto, le funzioni possono essere passate come argomento di altre funzioni, e una funzione può ritornare una funzione (si dice che il linguaggio è funzionale). Funzioni e Array ereditano dai rispettivi prototipi.
L'object literal è una sintassi semplificata per definire e assegnare proprietà ad un oggetto nello stesso tempo.

Capire l'inferenza dei tipi

Vediamo ora le funzionalità di TypeScript relativamente alle variabili introducendo l'inferenza dei tipi: che si dichiari il tipo oppure no, il transpilatore TypeScript fa del suo meglio per determinare il tipo di una variabile. Per esempio, prendiamo l'oggetto, in puro ES5:

```
var todo = {
        id: 123,
        title: 'string',
        completed :false
}
```

TypeScript determinerà i tipi della variabile e delle proprietà dell'oggetto, andando in errore se questi vengono modificati. In TypeScript si può definire un type any per avere il massimo della dinamicità su una variabile. Comunque, usando questo tipo, si perde il vantaggio di TypeScript di avere un linguaggio JavaScript maggiormente tipizzato. In questo modo si evitano degli errori in fase di sviluppo del codice e rende la manutenibilità e la lettura più semplice.

Specificare i tipi JavaScript
Vediamo come specificare esplicitare i tipi delle variabili e i return delle funzioni.

```
function totaleLength (x, y: string): number {
        var total: number = x.length + y.length;
        return total;
}
```

Perché specificare il return della funzione come number se TypeScript è abbastanza intelligente da leggere il corpo della funzione e capire che il valore ritornato è un numero? Più siamo espliciti nello sviluppo del codice, meglio TypeScript è in grado di aiutare lo sviluppatore durante lo sviluppo, notificando gli errori prima di andare a run time. Per speficare un typo array che contiene tipi diversi utilizziamo questa sintassi:

```
function totaleLength (x: any[], y: string): number {}
```

Specificare i tipi dei parametri delle funzioni
Per specificare che un determinato parametro può accettare un tipo o un altro tipo si usa l'operatore pipe:

```
function totaleLength (x: (string |  any[]), y: (string | any[])): number {
        var total: number = x.length + y.length;
        x.slice(0);
        x.push('abc');
        x.substr(1);
        return total;
}
```

length e slice sono metodi che vanno bene tanto per gli array quanto per le stringhe, i metodi push e substr invece sono specifici del tipo, come fare ad eseguirle se nei parametri della funzione possono essere passati come array o come stringhe?

```
function totaleLength (x: (string |  any[]), y: (string | any[])): number {
        var total: number = x.length + y.length;
```

```
        x.slice(0);

        if (x instanceof Array) {
        x.push('abc');
        }

        if (x instanceof String) {
        x.substr(1);
        }
        return total;
}
```

Dato che String è un tipo primitivo utilizziamo l'operatore instanceof. Potevo anche utilizzare typeof.

Aggiungere function overload

Alcune volte i tipi union visti sopra non sono sufficientemente restrittivi. Voglio esporre due diverse signature al metodo, uno quando vengono passati array, l'altro quando vengono passate delle stringe. Definire due diverse funzioni con lo stesso nome che hanno corpi funzione diversi a seconda dei parametri passati si chiama overload e in altri linguaggi come C# e Java è un pattern abbastanza diffuso. Semplicemente si da alle due funzioni lo stesso nome, cambiando il tipo dei parametri passati ed ovviamente i corpi funzione. Questo però in JavaScript non funziona. Per fare l'overload in JavaScript:

```
function totaleLength (x: string, y: string): number
function totaleLength (x: any[], y: any[]): number {
        var total: number = x.length + y.length;
        x.slice(0);

        if (x instanceof Array) {
        x.push('abc');
        }

        if (x instanceof String) {
        x.substr(1);
```

```
        }
        return total;
}
```

Si può pensare all'overload a dei metadati alla stregua della definizione di variabili. Il comportamento della funzione non viene modificato. Negli helper del code editor avremo le diverse opzioni per la funzione a seconda se passiamo stringhe o array.

4. Tipi personalizzati

Definire tipi custom tramite le interfacce
Una delle migliori funzionalità messe a disposizione da TypeScript è quella di definire dei tipi personalizzati per le variabili. Ci sono 3 modi per farlo:

- interfacce;
- classi;
- enums;

Vediamo il più semplice, le interfacce. Le interfacce in TypeScript non si discostano da quelle di altri linguaggi, sono funzionalità che agiscono come un contratto che descrive il dato e il comportamento che l'oggetto espone ad altre entità con cui interagisce. Per definire una interfaccia in TypeScript:

```
interface Todo {
        name;
        completed;
}
```

ho definito due proprietà dell'interfaccia. Ora specifico il tipo:

```
interface Todo {
        name: string;
        completed: boolean;
}
```

Le interfacce sono utilizzate a compile time e non interessano il run time: se scriviamo una interfaccia e transpiliamo il codice, vediamo che il codice JavaScript generato è nullo. Pensiamo alle interfacce a dei contenitori di informazioni per oggetti JavaScript;

```
var todo: Todo;
```

possiamo utilizzare anche un'altra sintassi per assegnare un interfaccia ad una variabile:

```
var todo = <Todo>;
```

E' preferibile comunque la prima.

```
var todo: Todo = { name: 'string', completed :false};
```

Se non siamo in grado di soddisfare l'interfaccia inserendo tutte le proprietà possiamo, nella definizione della stessa, inserire delle proprietà opzionali in questo modo:

```
interface Todo {
        name: string;
        completed?: boolean;
}
```

Le interfacce non definiscono il tipo della variabile, servono solo a descrivere che forma TypeScript si deve aspettare per un determinato oggetto, che proprietà deve o non deve avere.
La funzionalità è comoda perché è possibile definire tutte le interfacce in un solo punto, per poi modificare in una unica parte piuttosto che scorrere tutto il codice a cercare dove è stato utilizzato l'oggetto.
Vediamo adesso come implementare una method signature:

```
interface ItodoService {
        add(todo: Todo): Todo;
        delete(todoId: number): void;
        getAll(): Todo[];
        getById(todoId:number): Todo;
}
```

Il metodo getById ritorna un oggetto così come descritto nell'interfaccia Todo.
Abbiamo visto qui come definire dei tipi custom tramite interfaccia.
Vedremo nei capitoli successivi come implementare le interfacce così

definite tramite le classi.

Usare le interfacce per descrivere funzioni

```
interface jQuery {
	(selector: string): HTMLElement;
	version: number;
}
```

poi, per forzare TypeScript ad utilizzare l'interfaccia:

```
var $ = <jQuery>function(selector) {

}

$.version = 1.12;
```

Abbiamo così definito un tipo custom jQuery che è una funzione che ci permette di utilizzare jQuery all'interno di TypeScript tramite l'uso di una interfaccia.

Estendere la definizione delle interfacce
Dato che JavaScript è un linguaggio dinamico, è possibile estendere oggetti con proprietà e metodi dinamicamente e questo ci permette di fare la stessa cosa con le interfacce TypeScript.

```
interface Todo {
	name: string;
	completed?: boolean;
}

interface jQuery {
	(selector: (string | any)): jQueryElement;
	fn: any;
	version: number;
}
```

```
interface jQueryElement {
        data(name: string): any;
        data(name: string, data: any): jQueryElement;
}

var todo = {name: 'string'};
var container = $('#container');
container.data('todo', todo);
var savedTodo = container.data('todo');
```

Abbiamo qui definito tramite interfaccia quello che fa jQuery, fondamentalmente ritorna un oggetto contenente metodi e proprietà per lavorare con il DOM .
Proviamo ora a estendere l'interfaccia che contiene elementi custom, vogliamo aggiungere il metodo todo() a jQuery:

```
container.todo(todo);
```

Quando si utilizzano librerie di terze parti, non è buona norma andare a modificare interfacce proprietà e metodi della libreria, per questo usiamo le estensioni: Per estendere una interfaccia si definisce una nuova interfaccia con lo stesso nome di quella da estendere e si definiscono le nuove proprietà :

```
interface jQueryElement {
        todo (): Todo;
        todo (todo: Todo): jQueryElement;
}
```

le due definizioni non interferiscono tra loro. E' buona norma utilizzare questo approccio per lavorare con interfacce esistenti e soprattutto provenienti da librerie di terze parti.

Definire valori costanti con enum
Vediamo ora come definire i custom type con enums. Gli enums sono un modo per definire un set di valori constanti e con significato che si possono

utilizzare per effettuare dei replace nelle magic strings e numeri che si vogliono utilizzare.
Vediamo un esempio:

```
interface Todo {
	name: string;
	state: number;
}

var todo: Todo = {
	name: ‘string’,
	state: 1
}
```

gli stati sono dei numeri, ho bisogno di definire stati diversi corrispondenti a numeri diversi. Definiamo un enum:

```
interface Todo {
	name: string;
	state: todoState;
}

var todo: Todo = {
	name: ‘string’,
	state: TodoState.New
}

enum TodoState {
	New = 1,
	Active,
	Complete,
	Deleted
}
```

in JavaScript puro, sto definendo un oggetto con delle proprietà New, Active ecc. che hanno valore 1,2,3 ecc.

Vediamo come l'interfaccia enum ha a che fare con valori costanti, che spesso sono utilizzati per definire stati dell'applicazione. Per utilizzare l'enum:

```
function delete(todo: Todo) {
        if (todo.state != TodoState.Complete) {
        throw 'An error string';
        }
}
```

In questo modo stiamo utilizzando i valori numerici 1,2,3 degli stati, ma il codice è molto più leggibile.

Definire i tipi anonimi

I tipi anonimi sono interfacce definite inline ovunque che accettano un tipo. Definiamo per esempio una variabile che è un oggetto che ha una proprietà. Piuttosto che definire una interfaccia con una sola proprietà, scrivo una cosa del genere:

```
var todo: {name: string};
```

Sto assegnando alla variabile todo un oggetto con una proprietà name.

5. Classi

Capire l'ereditarietà del prototype

Un secondo tipo per definire tipi custom è l'utilizzo delle classi. TypeScript implementa ereditarietà, astrazione ed incapsulazione. TypeScript non introduce il concetto di classe, in quanto questo è introdotto da ES6, si tratta comunque di syntactic sugar. La programmazione a oggetti resta basata sul prototyping, gli oggetti istanziati sono sempre riferiti ad un prototype. Quando viene invocato il metodo, JavaScript guarda se c'è all'interno dell'oggetto istanziato, se non c'è si riferisce al prototype a cui l'oggetto è referenziato. Il prototype stesso è un oggetto. Da dove vengono questi prototype? Sono forniti dal linguaggio stesso. Il metodo più comune con cui JavaScript assegna un prototype all'oggetto è tramite il metodo di construct, tramite la funzione di construct viene creato l'oggetto, poi il prototype ad esso associato, poi esegue la funzione chiamata con la parola riservata new:

```
function TodoService() {
	this.todos = [];
}

TodoService.prototype.getAll = function () {
	return this.todos;
}

var service = new TodoService();
service.getAll();
```

Definire una classe

Il supporto alle classi di ES6 viene esteso da TypeScript con tutto il supporto alla tipizzazione visto fino ad ora:

```
class TodoService {

	todos: Todo[] = []; //tipo definito dall'interfaccia e variabile inizializzata
```

```
        constructor () {
        this.todos = todos;
        }

        getAll () {
        return this.todos;
        }
}
```

Faccio refactoring di quanto scritto sopra scrivendo in modo più sintetico:

```
class TodoService {

        constructor (private todos: Todo[]) {
        }

        getAll () {
        return this.todos;
        }
}
```

Applicare proprietà statiche
Si utilizzano per proprietà che si mantengono uguali attraverso più componenti dell'applicazione. Per esempio un counter che viene incrementato in più punti e mantiene il suo valore:

```
class TodoService {
        static lastId: number = 0;

        constructor (private todos: Todo[]) {
        }
        add(todo: :Todo)  }{
        var newId = TodoService.getNextId();
        }
```

```
        getAll () {
        return this.todos;
        }
        static getNextId () {
        return TodoService.lastId +=1;
        }
}
```

Usando static, una voltsa transpilato il codice si nota come la variabile lastId viene agganciata direttamente all'oggetto (funzione) TodoService.

I metodi statici e le proprietà statiche ad ogni modo è buona norma evitarle quando possibile. Devono essere utilizzati come ultima risorsa.

Rendere le proprietà più intelligenti con gli accessor
Sono proprietà che hanno dei metodi getter e setter.

```
interface Todo {
        name: string;
        state: todoState;
}

enum TodoState {
        New = 1,
        Active,
        Complete,
        Deleted
}

var todo: Todo = {
        name: 'string',
        state: TodoState.Complete
}
```

vediamo come implementarle sull'oggetto todo:

```
var todo: Todo = {
```

```
        name: 'string',
        get state() {
        return this._state;
        },
        set state (newState) {
        this.state = newState;
        }
}
```

Dal punto di vista dell'utilizzo dell'oggetto todo, non ho percezione dell'implementazione dei getter e dei setter, accedo a state normalmente:

```
todo.state = TodoState.Complete;
```

Per avere una utilità in questi metodi, si usa inserire della logica nel setter che in questo caso cambia lo stato sono se sono soddisfatte determinate condizioni.
In ogni caso comunque, piuttosto che inserire i getter e i setter nell'object literal, è preferibile utilizzarli nelle classi. Ad ogni modo, utilizzare getter e setter significa aggiungere logica alla classe, che sarebbe meglio inserire in un servizio.

Ereditare il comportamento da una base class
Le classi figlie possono ereditare proprietà e metodi della classe madre o sovrascriverli.
Immaginiamo di avere una classe TodoStateChanger e di volerla estendere con una classe figlia:

```
class CompleteTodoStateChanger extends TodoStateChanger {
}
```

Tecnicamente questo basta per estendere la classe. Se non si scrive la funzione di construct, resat valida quella della classe madre, e in genere è una cosa che è meglio evitare, ma se si rende necessario bisogna chiamare il constructor della base class:

```
class CompleteTodoStateChanger extends TodoStateChanger {
```

```
        constructor () {
        super(TodoState.Complete);
        }
}
```

Per sovrascrivere un metodo della classe madre, basta definire il nuovo metodo nella classe figlia con lo stesso nome. Se nella logica lo si vuole solo estendere, nel corpo funzione utilizziamo super per ereditare il comportamento del metodo della classe madre.

Implementare una classe astratta

Una classe astratta viene sviluppata al solo scopo di ereditare altre classi che vengono istanziate. Le classi astratte non vengono mai istanziate. Questa è una funzionalità TypeScript non presente in JavaScript ES6. Le classi astratte vengono utilizzate come classi base e in TypeScript basta utilizzare la parola riservata abstract. In questo modo avremo un errore ogni volta che proveremo ad istanziarla. La parola abstract si utilizza anche per la definizione dei metodi della classe astratta.

Controllare la visibilità con gli access modifier

Gli access modifier sono le parole riservate public, protected e private usate per controllare l'accesso a proprietà di classi allo stesso modo di come avviene in altri linguaggi come Java e C#. Si possono applicare al constructor, ai metodi, alle proprietà, ai getter e ai setter. Getter e setter devono essere comunque pubblici perché sviluppati appositamente per accedere a proprietà e metodi della classe.
Definendo private, la proprietà non è accessibile al di fuori della classe, con protected abbiamo accesso da una classe figlia alla proprietà, con public abbiamo accesso ovunque dall'esterno della classe. L'access modifier public è quello usato di default da JavaScript. Es:

```
class SmartTodo {
        constructor (public name: string) {

        }
}
```

JavaScript non supporta gli access modifier, pertanto scrivendo gli access modifier in TypeScript abbiamo solo effetto sul codice TypeScript, e servono più che altro ad esprimere gli intenti di chi sviluppa il codice. Le convenzioni specificano che le proprietà private di una classe vanno scritte con un underscore iniziale.

Implementare le interfacce

La ragione primaria per cui esistono le interfacce è quelle di applicarle ad una classe.
Tecnicamente, è sufficiente dichiarare l'interfaccia per un determinato oggetto e poi utilizzare l'oggetto nella classe per fare in modo che TypeScript controlli che l'oggetto rispetti l'interfaccia. Comunque, come regola generale è sempre bene esplicitare le cose. Vediamo quindi come fare per implementare una interfaccia in una classe.
Per fare ciò è sufficiente dichiarare la parola riservata implements dopo la dichiarazione di una classe seguita dal nome dell'interfaccia :

```
class TodoService implements ItodoService {

}
```

vediamo l'interfaccia:

```
interface ItodoService {
        add(todo: Todo): Todo;
        delete(todoId: number): void;
        getAll(): Todo[];
        getById(todoId: number): Todo;
}

interface Todo {
        id: number;
        name: string;
        state: TodoState;
}
```

```
enum TodoState {
        New = 1,
        Active,
        Complete,
        Deleted
}
```

Così facendo la classe TodoService deve avere i metodi definiti dall'interfaccia.
Se la classe implementa più interfacce, basta inserire i nomi delle interfacce separate da una virgola dopo la parola riservata implements.

6. Generici

Introdurre i generics

Un'altra funzionalità tipica dei linguaggi orientati agli oggetti che viene offerta da TypeScript sono i generics. I generics sono un modo per creare funzioni e classi che definiscono un comportamento che può essere riutilizzato attraverso tipi differenti. Vediamo un esempio:

```
function clone (value) {
        let serialized = JSON.stringify(value);
        return JSON.parse(serialized);
}
```

La funzione ritorna lo stesso tipo di oggetto passato come parametro. Usando un generic risolviamo il problema di definire il tipo del valore che ritorna dalla funzione, inoltre possiamo riutilizzare lo stesso codice per any e per tutti gli altri tipi della nostra applicazione. Per definire un generic :

```
function clone<T> (value) {
        let serialized = JSON.stringify(value);
        return JSON.parse(serialized);
}
```

Definito questo tipo generico, possiamo utilizzare un qualunque tipo per questa funzione.

```
function clone<T> (value: T): T {
        let serialized = JSON.stringify(value);
        return JSON.parse(serialized);
}
```

T non è un tipo, rappresenta un qualunque tipo che possiamo utilizzare per la funzione.
Ogni volta che ci troviamo nella situazione di copiare più volte il codice cambiando semplicemente il tipo utilizzato, allora è il caso di utilizzare un

generic.

Creare classi generiche
I generics possono essere applicati anche alle classi. Per esempio JavaScript utilizza la classe generic per definire la classe Array.
Il caso d'uso migliore per le classi di generics sono quelli che hanno tipo costituiti da coppie chiave-valore:

```
class KeyValuePair<TKey, Tvalue> {
        construct () {
        public key: Tkey,
        public value: TValue
        }
}

let pair1 = new KeyValuePair(1, 'First');
```

TypeScript di default interpreta la chiave come numero e il valore come stringa, per modificare questo comportamento:

```
let pair2 = new KeyValuePair<string, number>('second', 2);
```

Posso utilizzare così un qualunque tipo per la coppia chiave valore.

Applicare vincoli ai generic
Vediamo come modificare i generic per i consumers che li utilizzano.

```
function totalLength<T>(x: T, y: T) {

}
```

in questo modo con il generic costringo gli utilizzatori ad usare lo stesso tipo per entrambi i parametri, ma così facendo perdo l'informazione sui tipi che posso passare. Per ovviare:

```
function totalLength<T extends {length: number}>(x: T, y: T) {
```

```
}
```

oppure posso utilizzare una interfaccia:

```
interface IhaveALength {
        length: number;
}

function totalLength<T extends IhaveALength>(x: T, y: T) {

}
```

7. Moduli

Capire la necessità dei moduli in JavaScript

Da alcuni anni a questa parte JavaScript ha iniziato ad essere utilizzato in ambito industriale non solo nel contesto del browser, pertanto hanno iniziato a moltiplicarsi pattern di sviluppo e best practice. Uno dei più importanti è il module pattern che permette di modularizzare le applicazioni in componenti indipendenti.

Organizzare il codice con i namespace

L'implementazione in TypeScript dei namespace è molto simile a quella degli altri linguaggi di programmazione:

```
namespace Model {
        export interface Todo {
        name: string;
        state: todoState;
        }
        export enum TodoState {
        New = 1,
        Active,
        Complete,
        Deleted
        }
}
```

posso anche utilizzare più namespace nello stesso file. I namespace possono essere importati con la parola riservata import.

Usare i namespace per incapsulare dei private member

Viene illustrato il design patter IIFE (Immediatly Invoked Funcion Expression) che incapsula il codice mentre viene eseguito e poi sceglie quale parte di codice viene esposto specificandolo come valore di ritorno. Per prima cosa di scrive una determinata funzione e la si invoca dopo la definizione:

```
(function defineType () {

})();
```

Vediamo il precedente esempio di jQuery seguendo questo pattern:

```
var jQuery = {
        version: 1.19,
        fn: {}
};

(function defineType ($) {
        if ($.version < 1.15 )
        throw 'Plugin requires jQuery version 1.15+'

        $.fn.myPlugin = function () {
        //plugin code
        }
})(jQuery);
```

ho usato il nome jQuery nella chiamata a funzione e $ come parametro della funzione interna ma mi sto riferendo allo stesso oggetto.
Se vediamo il codice JavaScript generato da un namespace, vediamo che questo è solo syntactic sugar che genera delle closure con uno scope interno. Per esporre del codice devo utilizzare la parola chiave export.
Pertanto, usando il namespace il codice rimane incapsulato a meno che non lo esporti esplicitamente con un export e i namespace sono solo delle funzioni di closure, utilizzate per creare proprietà e funzioni private.
Le classi possono essere utilizzate con i namespace, pertanto non è necessario utilizzare gli accessor, ed si utilizza l'export per esportare proprietà e metodi.

Capire la differenza tra moduli esterni ed interni
Vediamo cosa sono i moduli esterni e come utilizzarli. Anche il modulo esterno implementa l'incapsulamento e l'esposizione di codice, ma lo fa in modo diverso rispetto al modulo interno: utilizza il file stesso come module

scope. Tutto il componente è definito in quel modulo. Ciò richiede che i moduli vengano esportati ed importati.
TypeScript implementa due varianti per effettuare l'importazione dei moduli, uno è il require, utilizzato anche da Node.js e la sintassi ECMAScript 2015. Le due sintassi sono completamente equivalenti, generano lo stesso codice. Si può scegliere la sintassi che si vuole, sebbene quella ECMAScript 2015 è nativamente supportata dai browser.

Cambiare da moduli interni a esterni
I moduli esterni utilizzano il file come limite dello scope del componente, il file diventa il modulo.
Per trasformare un modulo interno in esterno tolgo le dichiarazioni di namespace, mantenendo le parole chiave export.
Nel file tsconfig edito la proprietà module:

"module": "system"

Importare i moduli utilizzando la sintassi CommonJS
Qui vediamo come utilizzare la sintassi require:

```
import Model = require('./model');
```

Non si specifica l'estensione del file. In questo modo importiamo in un file, un oggetto esportato dal file model.

Importare moduli utilizzando la sintassi ECMAScript 2015
Per importare un modulo con questa sintassi:

```
import * as Model from './model';
```

posso anche utilizzare un'altra sintassi per importare determinati oggetti esportati dal modulo:

```
import {Todo as TodoTAsk, TodoState} as Model from './model';
```

Questa sintassi, essendo uno standard è caldamente raccomandata rispetto alla sintassi require.

Caricare moduli esterni

Non c'è uno standard definito per il caricamento di moduli esterni. Bisogna scegliere un module loader, ce ne sono molti disponibili e TypeScript ne supporta molti. Per gli scopi di questo corso si utilizza il loader System.js, che è quello che implementa le specifiche ECMAScript 2015. Così quando lo standard sarà definito, avrò già una implementazione nativa.
Con un module loader, nel file html uso un solo script tag per caricare il loader.
Dopodiché nello stesso html si scrive un piccolo script utilizzato per l'importazione:

```
<script type="text/javascript">
        System.defaultJSExtensions = true;
        System.import('app');
</script>
```

Nel file app.js utilizzo tutte le dichiarazioni import dei vari moduli che utilizzo per l'applicazione.

8. Sviluppo di applicazioni reali

Introdurre l'applicazione JavaScript campione

Spostiamo l'attenzione dalla sintassi del linguaggio ai pattern di sviluppo di una intera applicazione. Prendiamo una applicazione ES5 esistente per poi convertirla in TypeScript.
L'applicazione ha nel suo HTML i link a Bootstrap e alcuni script tag riferiti a file JavaScript esterni. Viene integrato anche jQuery.

Convertire un file JavaScript in TypeScript

Si crea il file di configurazione di TypeScript, si converte l'estensione del file da .js a .ts e poi si procede con le modifiche al codice. Per prima cosa si converte il tutto in una classe, poi si introducono delle interfacce, che via via andranno spostate in un file esterno. Piuttosto che specificare ogni volta i tipi dei parametri per le varie funzioni, è preferibile ricorrere alle interfacce.

Generare i file di dichiarazione

Non tutte le librerie sono scritte in TypeScript, per tanto spesso ci troviamo nella situazione di dover inserire una libreria JavaScript in un progetto TypeScript.
Per esempio jQuery, quando inseriamo la libreria abbiamo un errore in TypeScript perché non viene riconosciuto $ cioè l'oggetto jQuery. Si ricorre alla parola riservata declare :

```
declare var $: any;
```

Questo però non offre una tipizzazione forte per jQuery. Per descrivere il tipo nella dichiarazione si usano dei declaration files. TypeScript li può generare impostando il
file di configurazione in questo modo:

```
{
    "declaration": true
}
```

Quando si compila il file ts, il transpilatore genererà anche i declaration file.

Riferimenti a librerie di terze parti
Ad oggi ci sono già librerie js che includono i declaration file nei loro pacchetti di distribuzione. Il tool tsd serve a scaricare declaration file delle librerie js più diffuse. Tsd deve essere installato separatamente.
Attualmente tsd è deprecato in favore di typings, ma qui si usa ancora tsd a causa della maggiore stabilità.
Per sapere se un declaration file per una determinata libreria esiste, effettuiamo una query nel modo seguente:

tsd query jquery

per installare il file:

tsd install jquery --save

con il flag save salviamo il riferimento al declaration file nel tds.json.

Conversione a moduli esterni
Vengono spiegati alcuni trucchi aggiuntivi rispetto a quanto già detto precedentemente sui moduli.
Se scrivo export default nomeComponente, esporto un oggetto di default, e quando lo importo non uso più la sintassi con le parentesi graffe ma scrivo direttamente il nome dell'oggetto esportato di default.

Debuggare TypeScript con i source map
La console di Chrome rileva i bug nel codice JavaScript transpilato, ho bisogno di un source map per risalire alla riga e al file dove si trova il bug TypeScript. Il source map permette al debugger di rilevare dove si trova il bug nel codice sorgente.
Per attivare la generazione dei source map, si agisce nel file di configurazione:

“sourceMap”: true

9. Decoratori

Implementare i metodi decorator

I decorator sono una proposal per il nuovo standard del linguaggio che implementato il decorator design pattern per modificare il comportamento di classi, proprietà, metodi, parametri.
Si definisce un comportamento comune in un punto centralizzato e applicare facilmente il comportamento nei vari punti dell'applicazione.
Si tratta di metodi che wrappano altri metodi aggiungendogli comportamenti. Usato in modo appropriato, questo pattern è molto potente.
Per applicare un decorator si usa una sintassi a chiocciola, es:

```
@log
```

posta sulla riga sopra al metodo cui vogliamo applicare il decorator. Si tratta semplicemente di una funzione, applicata con una diversa sintassi.
Il decorator function viene definito così:

```
function log(target: Object, methodName: string, descriptor:
        TypedPropertyDescriptor<Function>) {
        //decorator logic
}
```

Al momento il compilatore TypeScript restituisce un errore perché al momento i decorator sono dei proposal e il compiler assume un approccio conservativo.

Implementare i class decorator

I class decorator modificano i constructor delle classi. La sintassi è la stessa vista sopra, solo che la funzione decorator prende solo la funzione classe come parametro.

Implementare i property decorator

l'implementazione della funzione decorator è la seguente:

```
function required (target: Object, propertyName: string) {
        //decorator logic
}
```

La sintassi usata per applicare il decorator resta esattamente la stessa di quella vista sopra.

Implementare i factory decorator
Servono a passare dei parametri al decorator. E' una funzione che prende i parametri e ritorna una decorator function.

10. Conclusioni

Come passi successivi a questo corso si consiglia di trovare risorse online su TypeScript, come repository GitHub e seguire i canali ufficiali per restare aggiornati sulle ultime evoluzioni del linguaggio, che è un progetto Open Source.