Just Enough TypeScript in 30 Minutes or Less

by Daniel Jebaraj

Syncfusion
Deliver innovation with ease

Introduction

What is TypeScript?

TypeScript is a new, open-source programming language that has been developed by Microsoft. It is a superset of the JavaScript language. Its goals are to provide compile-time type checking and object-oriented constructs on top of the core JavaScript language.

TypeScript does not add anything to JavaScript itself. Once compiled, the resulting output runs on any JavaScript interpreter within any environment. TypeScript is not in any way tied to Microsoft-specific platforms or code.

Why learn TypeScript?

JavaScript is a powerful language that is now the de facto programming language of the web. The open nature of JavaScript's type system is always a source of concern when developing and maintaining a large code base with a large team. The provision of a defined contract on top of the JavaScript language greatly assists with tooling. TypeScript enables great IntelliSense support. Without the provision of a typed contract, even the best JavaScript IntelliSense implementations are at best, spotty. TypeScript also provides possibilities for precise re-factoring. Automated re-factoring is currently possible only in a limited sense with plain JavaScript.

JavaScript's prototype-based inheritance model is powerful, but for many who come from a C#, C++, or Java background, the system feels quite alien. While it is entirely possible to create object-oriented abstractions in JavaScript, it involves some boilerplate code. Such code can, of course, be easily generated by a tool. This is exactly what TypeScript does. It allows programmers coming from a classical object-oriented background to define class, interface, and module constructs in a manner that is intuitive.

What will we learn?

In this white paper, we provide a quick introduction to TypeScript. It should take about 30 minutes to become acquainted with the basics.

Pre-requisites

1. **A good understanding of the basics of JavaScript.**
   JavaScript Succinctly (http://syncfusion.com/resources/techportal/ebooks/javascript), published by Syncfusion, is a good resource to become acquainted, or reacquainted as the case may be, with JavaScript.
2. **A working installation of Node.js.**
   The latest binaries for Windows are available for free download from http://nodejs.org/. Node runs JavaScript programs without embedding them in a browser. This makes it convenient to test your code. Most code samples in this white paper can be tested with Node.js. A few samples are tested with the full DOM inside a browser environment.
3. **The latest install of the TypeScript plugin for Visual Studio 2012**.
   This is available from http://www.typescriptlang.org.
4. **Code samples used in the book**.
   The samples are available for download from https://bitbucket.org/syncfusion/typescript_introduction. There is a wrapper solution named code.sln that groups code samples into sections for easy reference. Each section mentions that any sample files relevant to that section use the format below.

   Related code sample – {list of sample files}

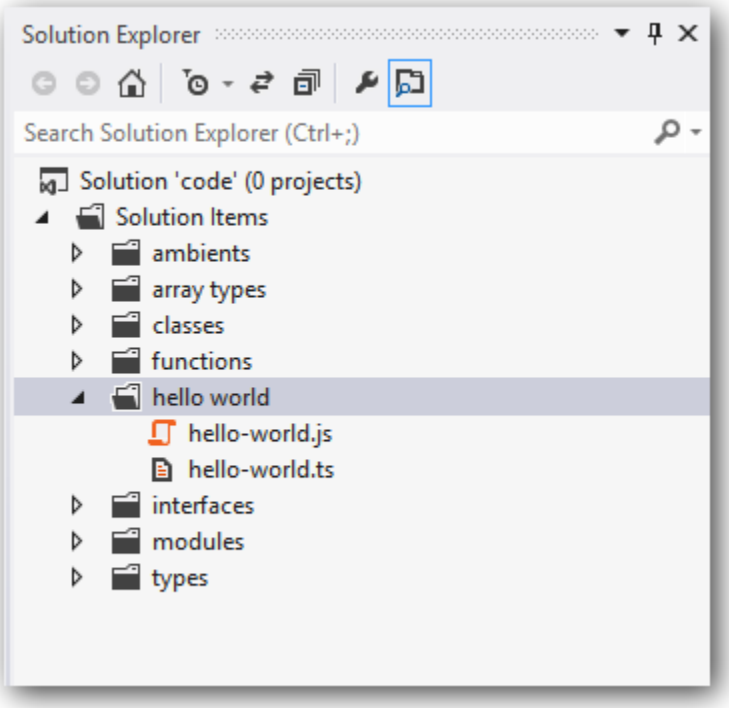
TypeScript Files

Related code sample: hello-world.ts

TypeScript files, by convention, end with extension *.ts*. When you program TypeScript, you work with these files. JavaScript (.js) file equivalents are generated by the TypeScript compiler named *tsc*. Generated JavaScript files can be used in the same manner as regular JavaScript.

In order to compile TypeScript files, you can run the TypeScript compiler as **tsc file-name.ts**. If your environment is configured properly, the TypeScript compiler will generate a JavaScript file with the same name and extension *.js*. You can then execute the resulting JavaScript file using node.js, *node file-name.js*.

Before you proceed further, please compile the TypeScript file *hello-world.ts* (available as a sample).



1. Open a command prompt.
2. Navigate to where the downloaded code samples exist.
3. Type "tsc.exe hello-world.ts"
4. If all goes well, a new JavaScript file name "hello-world.js" will be created in the same folder.
5. You may now execute *hello-world.js* using *node.js* as seen below.

    Node hello-world.js

TypeScript Language Elements

Type annotations

Related code sample: types.ts

TypeScript allows you to provide type annotations as an option alongside normal JavaScript code. Consider the TypeScript function below and its JavaScript equivalent.

TypeScript	`function addTwoNumbersTS(number1:number, number2:number) {`
	`    console.log(number1 + number2);`
	`}`

JavaScript	`function addTwoNumbers(number1, number2) {`
	`    console.log(number1 + number2);`
	`}`

TypeScript type annotations are added by placing a colon after the member being annotated. For instance, *number1* and *number2* are annotated per type number in the above function. Now consider the calls below.

```
// This works
addTwoNumbersTS(10, 20);

// This is not correct and TypeScript will flag this
addTwoNumbersTS("10", 20);
```

When you compile the above calls, the TypeScript compiler will detect that with the second call, you are passing in strings instead of the number type that the function expects. It will then warn you.

```
C:/git/learning/typescript/white paper code/types.ts(20,0): Supplied parameters do not
match any signature of call target
```

Contrasting the TypeScript function with the standard JavaScript version of the function, it is clear that the intent is to add two numbers. JavaScript does not, however, care about the type of object passed into the function. With plain JavaScript, both calls considered for TypeScript will work, but they will produce completely different results.

```
// This is correct.
addTwoNumbers(10, 20);

// This is not correct, but JavaScript will not mind.
addTwoNumbers("10", 20);
```

TypeScript's type annotations give the TypeScript compiler the information it needs to enforce compile-time type checking. TypeScript specifies the following types natively: number, bool, and string. Objects without a specific type are treated as the special type "any."

Type annotations and arrays

Related code sample: array-types.ts

Type annotations work with arrays as well. For instance, consider the code below.

```
function displayNames(names: string[]) {
    for (var i = 0, length = names.length; i < length; i++)
        console.log(names[i].toUpperCase());
}

var names = ['albert', 'ALFRED', 'Donald', 'Jack', 'Clarabel', 'Fran'];
displayNames(names);

// The following code will cause a compiler error.
// displayNames([1,2,3]);
```

The function **displayNames** is annotated as taking a single argument, an array of strings. The first call to displayNames works as expected. If you make changes to the displayNames function, you will notice that you get complete IntelliSense support with all string methods being picked up correctly for use on the indexed array element.

If you uncomment the second call to displayNames, passing in an array of numbers instead of strings, you will see the compiler flag an error, as seen below.

```
C:/git/learning/typescript/white paper code/array-types.ts(10,0): Supplied parameters
do not match any signature of call target:
       Could not apply type 'string[]' to argument 1, which is of type 'number[]'
```

User-defined types

Type annotations are of limited interest if all the TypeScript compiler is able to handle are number, string, and bool types. This is not the case. The TypeScript compiler allows you to specify and validate user-defined types, making the type system very flexible and powerful.

Interfaces

Related code sample: interfaces.ts

TypeScript offers support for interfaces. The support is simple, yet powerful. You can declare a simple interface, as seen below.

```
interface ITaxcalculator {
    getTax(purchase: number): number;
}
```

You can then implement this in an object literal or a class. Both examples are shown below.

Class implementing an interface

```
class DefaultTaxCalculator implements ITaxcalculator {

    getTax(purchase: number): number {
        return purchase * 0.1;
    }
}
```

We have not yet looked at the class construct (we will do so in the next section), but the idea should be clear. We implement a method that has the same signature as the method declared in the interface. We also declare that the class implements the interface ITaxCalculator.

Implementing a class is not required to implement an interface. TypeScript stays true to the dynamic nature of JavaScript. You can alternatively declare an object literal, as given below. This will also work and will satisfy the requirements of the interface specification.

Implementing an interface by simply implementing methods in an object literal

```
{
    getTax: function (purchase: number) {
        return purchase * 0.15;
    }

}
```

You can consume these two objects as shown below in a TypeScript function that accepts an argument of type, ITaxCalculator.

```
function displayTotal(purchase: number, taxCalculator: ITaxcalculator) {
    console.log("Total value is " + (purchase +
taxCalculator.getTax(purchase)).toString());
}

// This works.
var taxCalculator = new DefaultTaxCalculator();
displayTotal(100, taxCalculator);

// Implemented as an object, this also works.
displayTotal(100, {
    getTax: function (purchase: number) {
        return purchase * 0.15;
    }
});
```

Note that **displayTotal** is declared to take an object that implements **ITaxCalculator** as the second argument. If you pass in an object that does not fulfill this requirement, you will receive a compile-time error.

```
// Un-comment the line below to see the compiler catch an incorrect object being
passed in.
//displayTotal(100, {});
```

NOTE: A key aspect is that there are no run-time checks added. The checks happen at compile time. There is no run-time cost associated with these checks.

Classes

Related code sample: classes.ts

Classes are fundamental units of abstraction that are intuitive for programmers coming from C++, C#, or Java (CS-Dev) backgrounds.

JavaScript's prototypical inheritance model makes it possible to implement classes that are similar to the classes we are familiar with. The syntax for doing so is, however, different than what we are accustomed to.

For instance, consider the following class (or in JavaScript terms, constructor function), named **Car1**.

```
var Car1 = (function () {
    function Car1(horsePower, make, model) {
        this.horsePower = horsePower;
        this.make = make;
        this.model = model;
        console.log("Car1 has been created");
    }
    Car1.prototype.getHorsePower = function () {
        return this.horsePower;
    };
    Car1.prototype.getMake = function () {
        return this.make;
    };
    Car1.prototype.getModel = function () {
        return this.model;
    };
    return Car1;
})();
```

The code above does the following:

- It declares a constructor function that can be considered a sort of cookie cutter for new JavaScript objects that are to be initialized to a specific plan.
- The constructor function takes three arguments: horsepower, make, and model. It then assigns the provided values to properties on the current object.
- Three functions that act as *accessors* for the properties are defined.
    - The only unusual aspect is that they are defined on a shared prototype object that is held by the constructor function (for further details on this, please refer to *JavaScript Succinctly*).
    - Adding functions this way to the prototype allows them to be shared across object instances instead of being duplicated for each object created using the constructor function.

Consuming this "Class" in JavaScript is simple and quite similar to how things would work in CS-Dev.

The code is seen below.

```
var car1 = new Car1(100, "Toyota", "Camry");
console.log("Car's power level (HP)" + car1.getHorsePower());
console.log("Car's make " + car1.getMake());
console.log("Car's model " + car1.getModel());
```

No surprises here. The code feels like CS-Dev. One odd aspect with the code is that the definition code does not feel like CS-Dev. It definitely feels different though, the result is more or less similar in purpose.

TypeScript takes this concept and provides a CS-Dev-friendly class structure that is then translated by the TypeScript compiler to JavaScript code. Consider the same class implemented using TypeScript. The code looks and feels like a C# class. Running this code through the TypeScript compiler will produce the JavaScript code that we looked at earlier.

```
class Car1 {

    private horsePower: number;
    private make: string;
    private model: string;

    constructor (horsePower: number, make: string, model: string) {
        this.horsePower = horsePower;
        this.make = make;
        this.model = model;
        console.log("Car1 has been created");
    }
```

```
    getHorsePower() {
        return this.horsePower;
    }

    getMake() {
        return this.make;
    }

    getModel() {
        return this.model;
    }
}
```

Compare the differences with JavaScript:

- The code above will feel very familiar to developers coming from CS-Dev.
- There is a constructor function that takes three arguments and initializes three property values.
- Accessor functions are defined for each of the values.

Further improvements

While the above TypeScript class works well and has a familiar syntax, the initialization code is a bit verbose and can be shortened. When improvements are made, why not go all the way!

Consider the code below.

```
class Car2 {

 constructor (private horsePower: number, private make: string, private model:
string) {
        console.log("Car2 has been created");
    }

    getHorsePower() {
        return this.horsePower;
    }

    getMake() {
        return this.make;
    }

    getModel() {
        return this.model;
    }
}
```

The code makes the following changes to our initial TypeScript implementation:

- The code removes explicit assignment from values passed in the constructor to object values.
- Object property definitions for each of the passed-in values have been removed.
- The keyword **private** has been added to each of the values passed into the constructor. This by itself automatically causes the TypeScript compiler to provide the initialization code we had manually written earlier. This setting could be public, and the same essential code would be generated. The only difference is that when private access is specified, TypeScript code will not access members directly. Private or public settings have no impact on the generated JavaScript where everything is accessible by default.

Usage code is shown below for reference.

```
console.log("Car class Take 2");
var car2 = new Car2(100, "Toyota", "Camry");
console.log("Car's power level (HP)" + car2.getHorsePower());
console.log("Car's make " + car2.getMake());
console.log("Car's model " + car2.getModel());
```

Classes and inheritance

Related code sample: classes-inheritance.ts

Let us now consider a set of classes with an inheritance relationship. Class definitions are below.

```
class Man {

    constructor (private description: string) {
    }

    public getPower() {
        return 1;
    }

    getDescription() {
        return this.description;
    };
}

class Superhero extends Man {
     getPower() {
        return super.getPower()*2;
    };
}

class Batman extends Superhero {
     getPower() {
        return super.getPower()*3;
```

```
    }
}

class Superman extends Superhero {
     getPower() {
        return super.getPower()*10;
    }
}
```

The base class defines the **getPower** method, which is then overridden by each derived class. This is standard object-oriented code, but as you are likely aware, such constructs require substantial boilerplate code to implement in JavaScript. TypeScript neatly abstracts this boilerplate away and provides a clean class construct that behaves in a way that most programmers coming from a CS-Dev background would expect it to. You can review the generated boilerplate by looking at the generated JavaScript file. *JavaScript Succinctly* has the background information you need to understand what the boilerplate code does to accomplish inheritance.

Using the defined class is as expected. The JavaScript instance of checks works correctly, and it states that an instance of a derived class is an instance of its type, as well as any base classes.

```
var batman = new Batman("Drives batmobile");
console.log("Batman's power level " + batman.getPower());
console.log("Batman: " + batman.getDescription());

// JavaScript type checks.
console.log("Batman is of type Batman");
console.log(batman instanceof Batman);

console.log("Batman is of type Superhero");
console.log(batman instanceof Superhero);

console.log("Batman is of type Man");
console.log(batman instanceof Man);

console.log("Batman is of type Superman");
console.log(batman instanceof Superman);
```

It is worth noting that derived classes have access to their class through a special keyword, "super," as seen in the code above.

Functions

Functions are top-level constructs in JavaScript. TypeScript makes it possible to provide type annotations. In addition, it adds some syntactic sugar that makes things clearer for CS-Dev.

Related code sample: functions.ts

Simple definition

As we have seen earlier in this white paper, you can declare functions that annotate their parameters. You may also annotate the return value.

In the example below, we declare a function that takes two numbers and returns a number. The return type annotation is added right after the argument list. It is also acceptable to omit the return type annotation. The compiler will still see that you are returning a number and will infer the return type of the function to be a number.

```
function addTwoNumbers(number1:number, number2:number):number {
    console.log(number1 + number2);
    return number1 + number2;
}

addTwoNumbers(10, 20);
```

Lambda definition

There is also an alternate lambda-like way to define a function. This form is shown below.

```
var add = (n1: number, n2: number) :number => {
    console.log(n1 + n2);
    return n1 + n2;
}

add(10, 20);
```

In the definition above, the variable **add** is assigned to a function that takes two numbers and returns a number. This syntax should be very familiar if you are comfortable with C# lambda function syntax.

There is, of course, nothing that resembles the above code in JavaScript. The translated function is shown below and is just a standard JavaScript function.

```
var add = function (n1, n2) {
    console.log(n1 + n2);
    return n1 + n2;
};
```

The lambda is just syntactic sugar that TypeScript adds. There are other advantages to the lambda approach, as we will see shortly in the section titled “Function Context.”

Declaring objects that hold a specific function type

It is also possible to define a variable as declared to hold a function of a specific type and then assign it to an instance of a function that meets type specifications (think typed function pointer or reference).

The code below declares that the variable **addAndLogFunction** can hold a function that takes two integers and returns an integer. You can only assign this typed function pointer to compatible functions.

```
var addAndLogFunction: (n1: number, n2: number) => number;
```

In the code below, we assign it to the two compatible functions we create, **addTwoNumbers** and **add**.

Function definitions

```
function addTwoNumbers(number1:number, number2:number):number {
    console.log(number1 + number2);
    return number1 + number2;
}

addTwoNumbers(10, 20);


var add = (n1: number, n2: number) :number => {
    console.log(n1 + n2);
    return n1 + n2;
}

add(10, 20);
```

Assignment to our typed function variable

```
addAndLogFunction = addTwoNumbers;
addAndLogFunction(10, 20);
addAndLogFunction = add;
addAndLogFunction(10, 20);
```

Arrays of functions

Related code sample: function-array.ts

It is also possible to declare an array as a collection of functions of a certain type. For instance, consider the code below. Two functions are defined using lambda syntax. They both take a single string parameter and then output a message with the passed-in parameter.

```
var message1 = (message: string) => { console.log("message 1." + message); }
var message2 = (message: string) => { console.log("message 2." + message); }
```

In Visual Studio 2012, if you hover over the function declarations, you will see the following type picked up for both functions. Please note that **void** is picked up as the return value automatically since we are not returning anything from these functions.

```
(message:string) => void
```

We can now declare an array to hold multiple functions of the above type. The syntax takes a little getting used to, but makes sense if you parse it out.

```
var items: { (message: string): void; }[] = []
```

In order to understand this better, consider a standard array of strings. We would define it as below.

```
var items:string[] = ...
```

The form is "individual item type" followed by the array notation "[]". Breaking the array of functions using this model, we see that the function is of a type that takes a string argument and does not return anything. This definition is then enclosed within an object literal, perhaps to say that this is a custom type.

We add two function instances to the array, as shown below.

```
var items: { (message: string): void; }[] = []
items.push(message1);
items.push(message2);
```

We can then execute the functions.

```
var length = items.length;
var i = 0;

while (i < length) {
    items[i]("hello");
    i++;
}
```

When you compile the sample, take a look at the JavaScript that is produced. It will not contain any of the syntactic typing-related sugar that we see in TypeScript, but it should be quite simple to understand.

Function context

Related code sample: function-context-test.htm, function-context.ts, and function-context.js

One of the aspects of JavaScript that trips programmers used to traditional object-oriented languages is the behavior of the **this** reference. In C#, **this** always refers to the current object instance. It does not matter how the current code was invoked. The **this** reference can be safely accessed to extract values or call methods on the object.

This is not the case with JavaScript. With JavaScript, the **this** reference is merely the current context. When you receive a callback (say, when an AJAX call is completed) you have no guarantee that the **this** reference will point to any specific object instance.

As with everything in JavaScript, there are various ways to work around this and always obtain access to the current object. This involves boilerplate code that TypeScript can abstract away. When you use lambda-declaration syntax, as shown below, you are guaranteed that the **this** reference always points to the current object.

In the example below, we set up a **setInterval** call in two different ways: one using lambda syntax, a function named **start,** and the other, **starts**, using standard JavaScript function declaration syntax.

```
var messageRepeater = {
    message: "<p>Hello world</p>",
    displayMessage: function () {
        $("body").append(this.message);
    },
    start: function () {
        setInterval(() => this.displayMessage(), 1000);
    },
    startJS: function () {
        setInterval(
            function () {
                this.displayMessage()
            }, 1000);
    }
};
```

The HTML markup we use to test these two options is given below. You can try changing the calls between **start** and **startJS**. When **start** is used, the actual message "Hello World" will be traced, since access to the **this** reference works as expected. You will observe that this is not the case when **startJS** is used.

```
<script>
            $(function () {
                messageRepeater.startJS();
                console.log("started");
            });

        </script>
    </head>

    <body>
        <div id="log"></div>
    </body>
```

Modules

Related code sample: syncfusion.ts and main.ts

Modules serve to group portions of code that are related in a logical entity. They are similar to C# namespaces.

In the code below, we declare a module named **Utils**. The **export** keyword states that we intend to make functionality available for consumption when this module is imported into other code. We then export a class named **Formatter** that has a single method that takes text and displays it with two lines of hashes added, one before and one, after the actual text being displayed. In the provided sample code, we save the code below in a file named **Syncfusion.ts**.

```
export module Utils {
    export class Formatter {
        displayText(message: string) {
            console.log("############");
            console.log(message);
            console.log("############");
        }
    }
}
```

The code below (refer to the sample code **main.ts**) imports the module **Utils** from the file **Syncfusion.ts**.

```
import Syncfusion = module("Syncfusion");
```

```
module Main {

    var formatter = new Syncfusion.Utils.Formatter();
    formatter.displayText("Hello world");
}
```

It then creates an instance of the **Formatter** class that we exported from **Utils**. Please note that we refer to **Utils** within the scope of the file name that contains it, in this instance, *Syncfusion*.

What about Existing JavaScript Code?

Related code sample: ambients.d.ts, ambients.ts, testambients.ts, and testambients.html

We looked at several cool TypeScript features that will help you author and maintain large bodies of JavaScript code in a structured manner. But what about the JavaScript code that already exists? Is it possible to have TypeScript understand plain JavaScript code and provide such cool features as IntelliSense? The good news is that yes, TypeScript can handle JavaScript modules written in a very structured manner with no knowledge of TypeScript. We will illustrate this with a sample.

Consider the following JavaScript code that was authored without using TypeScript. It declares a constructor function named **Person**. A couple of functions are added to the constructor function's prototype: one to display the full name of the person and another to display the first name. We then create two named instances using the **Person** constructor, one person named Mickey and another named Donald.

```
// This is regular JavaScript code.
// There is no TypeScript in this file.

// Person constructor and related functions.
function Person(first, last) {
    this.first = first;
    this.last = last;
}
Person.prototype.displayName = function () {
    console.log(this.last + "," + this.first);
};
Person.prototype.displayFirstName = function () {
    console.log(this.first);
};

// Named instance of person.
```

```
var mickey = new Person("Mickey", "Mouse");

// Named instance of another person.
var donald = new Person("Mickey", "Mouse");
```

We can expose the pre-existing JavaScript code seen above and make it visible to TypeScript code in a structured manner using the TypeScript seen below.

```
// This is a TypeScript definition file that we create
// to expose the types in ambients.js.
// There are no changes to ambients.js.
// By declaring the interface to ambients.js, we can get complete IntelliSense
support
// when using the code.
interface IPerson {
    displayName():void;
    displayFirstName(): void;
}


class Person implements IPerson {
    public first: string;
    public last: string;
    constructor (first: string, last: string);
    displayName():void;
    displayFirstName(): void;
}
```

Let us look at the declarations step-by-step in order to understand what is being specified here.

First, we declare an interface named **IPerson**. We then declare the two functions that **Person** exposes in our pre-existing JavaScript code. Please note that the definition of an interface **IPerson** is not strictly needed. You can directly declare a class that mirrors the interface exposed by **Person**, and that will work just fine. We just want to show that you can expose facets of your underlying JavaScript code in a selective manner, should you choose to do so.

We then declare a class named **Person** that derives from **IPerson**. The **Person** class is a complete representation of the underlying **Person** constructor function with no implementation, of course. The two methods specified in the JavaScript **Person** are listed. Also listed is a representation of the actual constructor function that takes two strings, a first name and a last name. We also declare that **Person** has two properties named **first** and **last**.

This code is saved as *ambients.d.ts*. We can then include this TypeScript definition file using a specially formatted reference comment (as seen in the code below) in the TypeScript file that

will consume the plain JavaScript code we are exposing through TypeScript-friendly interfaces and classes.

```
/// <reference path="ambients.d.ts"/>
var p = new Person("Daisy", "Duck");

p.displayName();
p.displayFirstName();

console.log(p.first);
```

Once you include the *.d.ts* file that represents the JavaScript we wish to expose, you have complete IntelliSense support in Visual Studio for the underlying JavaScript code.

There is one aspect of the underlying JavaScript code we have not exposed yet, the predefined objects **mickey** and **donald**. We can expose these using the special keyword **declare**. In a way, **declare** works like the **extern** declaration in C. We are essentially communicating to the TypeScript compiler that when it sees the instances declared below, it should assume they can be treated as the specified type.

```
// Declare mickey as an instance of person.
declare var mickey: Person;

// Declare donald as an implementation of IPerson and not the class Person,
// though both mickey and donald are just instances of Person.
// Object properties will not show up through IntelliSense.
declare var donald: IPerson;
```

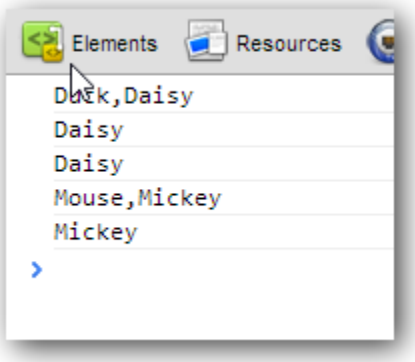
We declare **mickey** to be of type **Person** and declare **donald** to be of type **IPerson**. This is, of course, interchangeable since both are instances of **Person**. The idea is that whatever the underlying representation, you can choose to control the facet that is exposed to TypeScript. As you play around with the code, notice that you have complete IntelliSense support.

We have included a simple HTML file named *testambients.html* that includes the generated JavaScript file *test-ambients.js* and the original *ambients.js*.

```
<html>

<head>

    <script src="ambients.js"></script>

    <script src="test-ambients.js"></script>
```

```
</head>

</html>
```

If you open this file in the browser and look in the JavaScript control, the following text should be displayed.



One important thing to keep in mind about the **declare** keyword is that it does not really create a variable. It just exists to help define the type of an object at design time. There is no run-time purpose for it. It exists for the sole purpose of better IntelliSense and compile-time checking.

The example seen above is quite simple, but the idea that you can expose plain JavaScript code in a structured manner to be consumed by TypeScript is quite powerful. The same concept can be applied to complex libraries such as jQuery and Backbone.js.

Microsoft actually provides a definition file for jQuery 1.7.x. It is available as part of the TypeScript samples on the TypeScript CodePlex site: http://typescript.codeplex.com/SourceControl/changeset/view/fe3bc0bfce1f#samples%2fjquery%2fjquery.d.ts. Take a look at this file and compare it with the actual jQuery API documentation. It will help you become familiar with the declaration syntax.

Related FAQs of Interest

How is TypeScript different from CoffeeScript or Script#?

CoffeeScript and Script# attempt to provide a degree of abstraction over JavaScript. They do not encourage the use of JavaScript syntax. When working with CoffeeScript, you are for the most

part working with a different language that is based on ideas from Ruby and Python. When working with Script#, you are working with C# syntax.

TypeScript takes a different approach. Its goal is not to introduce another language, but to enhance JavaScript. All JavaScript code is valid TypeScript code. TypeScript then introduces a thin additional layer that provides valuable services such as compile-time type checking and object-oriented constructs.

It is worth noting that TypeScript generates very little additional JavaScript beyond what you write. There are a few instances where TypeScript generates additional JavaScript code, such as generating code to represent the Class construct. In most other instances, the additional TypeScript-specific annotations are simply stripped away to produce clean JavaScript. This is quite different from how CoffeeScript works. It adds a number of high-level abstractions, such as array comprehensions, that generate a substantial amount of JavaScript code.

One advantage to the approach that TypeScript takes is that developers do not have to switch between code written in different languages. Another advantage is that it is very easy to leverage existing JavaScript libraries such as jQuery. You do not have to try to figure out how to use these libraries because TypeScript is a superset of JavaScript and can consume JavaScript libraries with no additional work. As we saw earlier, it is possible to define strongly typed interfaces even for pre-existing JavaScript code that has no knowledge of TypeScript. Once such interfaces are defined, Visual Studio is able to provide great IntelliSense and re-factoring support.

Interested in Learning More?

1. Explore the TypeScript site on CodePlex: http://typescript.codeplex.com/.
2. There is an excellent video tutorial and several samples available on the TypeScript home page at http://www.typescriptlang.org/.

TypeScript Succinctly

Syncfusion is working on publishing a book on TypeScript named *TypeScript Succinctly*. This book should be available by the middle of November 2012. It will be available for free download from the Syncfusion Technology Portal at http://syncfusion.com/resources/techportal/ebooks.

Conclusion

We hope this white paper was helpful in gaining a working understanding of TypeScript. We are very excited about the potential for this language and look forward to using it to aid our JavaScript development. We hope you do as well.