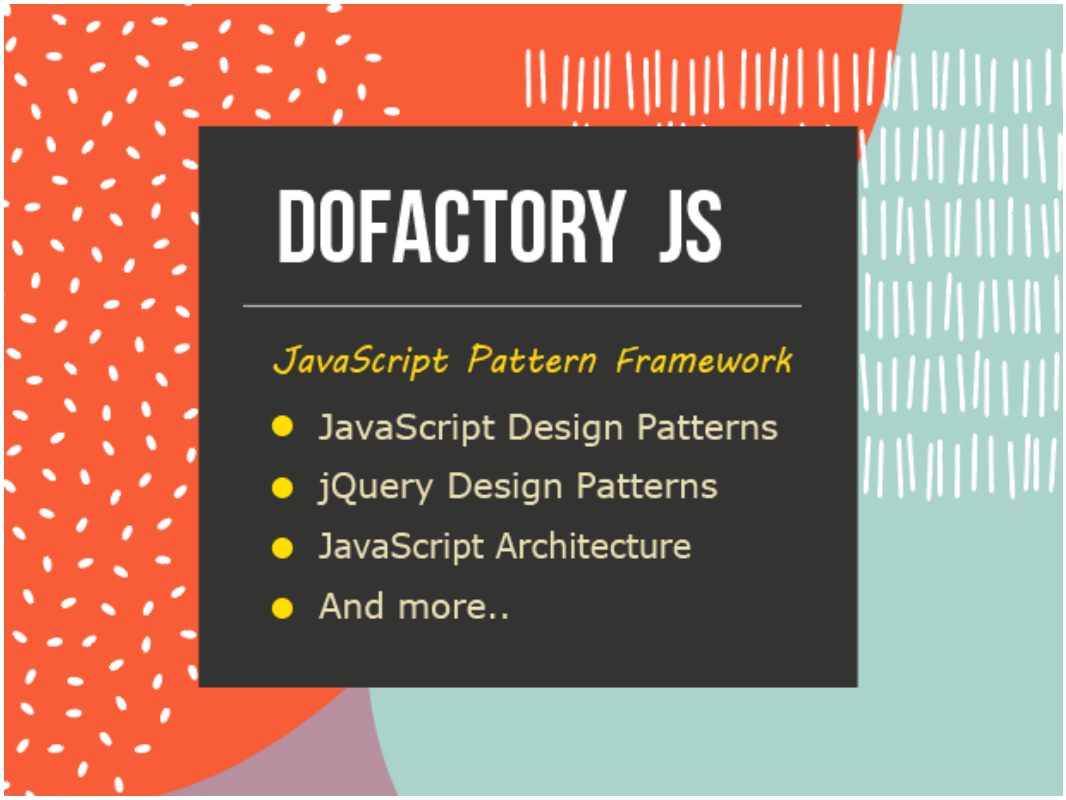
Dofactory JS 6.0

JavaScript Architecture Patterns



by

Data & Object Factory, LLC

www.dofactory.com

7. JavaScript Architecture Patterns

Index

Introduction

In prior sections we have seen *idioms* which are like mini patterns that offer small and succinct solutions to specific language-related problems. Next are *design patterns* which work at the object level. They offer solutions to recurring problems that are related to objects and their interdependencies. Finally, *architecture patterns* are solutions to recurring software problems related to the higher-level structure and design of projects as a whole, i.e. the architecture of applications. Architecture patterns is the topic of this section.

Although not discussed in their own section, MV* Patterns also fall under the category of architecture patterns. They were designed to bring organization and structure to large scale applications. MV* is a large subject area, why is why they ended up in their own section, but, again, they are true architectural patterns.

JavaScript was not designed to build large-scale applications. Language features and facilities that are available to most modern languages are lacking. These include: namespaces, typing, classes, interfaces, modules, importing, exporting, and others. Fortunately, JavaScript is a very flexible language and today many solutions and support tools are available to build JavaScript apps that are standing on a sound architectural footing.

In this section we will touch on a variety of topics; these include script loading, modularity, AMD & require.js, JavaScript transpilers, error handling, and JavaScript testing. Most of these are relevant to all but the most trivial applications.

Script Loading

Script loading is important if your goal is to build fast loading web apps. Loading a large number of script files by placing multiple <script> tags in the header will slow down your app because they are loaded and executed *before* any HTML markup is rendered. Minifying your scripts helps but it does not change the fact that script loading does block page rendering.

Script loaders are tools that can alleviate this problem by allowing the load process to take place asynchronously and/or in parallel (i.e. multiple script files at the same time). It is important to do performance and comparison testing, but in many cases script loaders make a difference in page load times.

The truth is that script loaders are not always necessary. If your goal is to minimize page render times, then an easy way to make a difference is by concatenating all script files into a single file and placing the script tag towards the end of the file, just before the closing </body> tag. Once loaded, browsers will do a good job caching this file.

However, merging all files is not always possible; for example when your script tags reference CDN hosted files, such as on Google, Microsoft and Yahoo, rather than pulling in files from your server.

Script loaders mostly follow a similar pattern to indicate what needs loading. Here is some generic code to show what it looks like:

```
ScriptLoader.load("jQuery.js", "app.js");
```

This will load the two named scripts asynchronously which will speed up loading. Unfortunately, this may pose a problem for other scripts that have dependencies on the above referenced scripts because when they execute it is not known whether the two files have completed loading or not.

Most script loaders offer a ready event to this timing issue. This event will fire when all scripts have completed loading and are available for use. You pass it a callback that will execute at that time. The syntax looks something like this:

```
ScriptLoader.ready(function () {
    // everything is loaded
}
```

Most script loaders allow you to combine the two above snippets in a single call in which you pass an array with scripts that require loading and a callback function which will be invoked when all scripts have completed loading, like so:

```
ScriptLoader.load(["jQuery.js", "app.js"], function () {
     // callback called when loading has completed
});
```

In case you're wondering how this works under the hood, most of these loaders simply insert a <script> element in the DOM document. Here is some skeleton code to get the idea:

```
var tag = document.createElement("script");
var src = http://mysite.com/js/app.js;
var first = document.getElementsByTagName("script")[0];

first.parentNode.insertBefore(tag, first);
```

This will insert the new script tag before the first <script> in the document, which most likely is a reference to the script loader's JavaScript file.

There are a number of script loaders available. The most popular ones include Require.js, Head.js, curl.js and LazyLoad (by the way, the name LazyLoad is a reference to the Lazy Load pattern, i.e. load on demand).

Some script loaders have a narrow focus and all they do is speed up file loading. Others offer additional services which we will look at in a moment. LazyLoad falls under the first category: it allows you to speed up the load processing by loading CSS and JavaScript files on demand. LazyLoad is a tiny tool which when minified is less than 1K. Here is an example of how you use it with two JavaScript files:

```
LazyLoad.js(["jQuery.js", "app.js"],function () {
    // executes when loading is complete
});
```

The js method loads the files in parallel. If your files need to be loaded sequentially and in order, then the statements can be nested:

```
LazyLoad.js("jQuery.js",function () {
    // executes when jQuery is complete
    LazyLoad.js("app.js", function () {
```

```
        // executes when both are complete
    });
});
```

LazyLoad has the ability to load CSS files in a similar way by using its *css* method.

Other script loaders do more than just loading scripts; they also help with organizing large applications. The reason for this dual purpose is that script loading is inextricably related to the organization of your modules and the management of their dependencies. Modularization is the topic of the next section followed by a review of these dual purpose loaders.

Modularity

When an application is said to be modular, it means that it has well-defined units of functionality are relatively independent from each other (that is, they are loosely coupled). Suppose you have an app that maintains Employee data. You may see modules like Employees, Admin, Reporting, and some cross-cutting modules, such as, Ajax, Security, and Utils (note: cross-cutting means they affect all areas in the app). Modularity is beneficial because it brings clarity to your code base; the programs are easier to understand, easier to test, and easier to maintain.

Most languages allow you to organize a large body of code in modular units. JavaScript does not natively support this, but the Module pattern allows us to organize the code into clearly defined units called modules. Usually there is one module per file.

The next step is the ability to import these modules where necessary. Most languages support native syntax for this, such as, `import`, `using`, or `require`. JavaScript does not have this and there is no easy way to import modules and help with the dependency management of these modules.

The ES6 (EcmaScript 6) proposal addresses the problem head-on with a new module system and the addition of these keywords: `module`, `export`, and `import`. Unfortunately, it will take some time before JavaScript developers are generally coding against the new ES6 standard.

In the meantime, many developers building large systems use an alternative called AMD (Asynchronous Modular Definition). AMD is a protocol: its goal is to provide a modular development API that allows JavaScript to define modules and their dependencies that can be loaded asynchronously (notice how script loading comes into play again).

In the next section we will review the AMD format specification as well as the most popular open-source tool that implements it: Require.js.

AMD & Require.js

AMD is a protocol and Require.js is the implementation of this protocol. Its goal is to make large web projects easier to manage. Projects that involve many JavaScript files are hard to maintain because of the dependencies that exist between these files. These dependencies force developers to carefully consider the order in which JavaScript files are referenced in each page. The complexity grows exponentially when more pages and files are added, and this can become a real stumbling block. Let's look at an example:

Say your project has 10 JavaScript files.

```
├── file1.js
├── file2.js  <- depends on?
├── file3.js
├── file4.js
├── file5.js
├── file6.js  <- utilities
├── file7.js  <- helpers
├── file8.js  <- support
├── file9.js
├── file10.js
```

You can probably already guess what is happening. Suppose file3 uses utility functions that are available in file6. File6 in turn depends on file7 because it has some handy helper functions. File9 requires File8 which contains essential support. And, file10 depends on file2 and file5 because they have the necessary base classes. In turn, the two latter ones also depend on File6 utilities. You get the idea.

We only have 10 files and already it has gotten very complex. For each page that you build you have to ensure that the correct script files are included and are placed in the right order. It is easy to see that managing a large project with dozens of files/modules by hand is difficult, error prone, and not very scalable. This is where AMD comes in.

The AMD module format is a specification that allows you to define modules and their dependencies which are then loaded asynchronously. AMD has been adopted by several popular JavaScript tools and libraries, including jQuery, Dojo, Mootools, and Firebug.

AMD offers a great deal of flexibility, first by being asynchronous, but also by introducing string IDs for dependencies and allowing these to be mapped to a different path which is great for creating mocks in unit testing. Furthermore, the module definitions in AMD are encapsulated in their own namespace which avoids polluting the global object's namespace.

We will look at Require.js which implements the AMD module API format. It is the most popular script loader and dependency manager available today.

Require.js is one of those tools that initially you may not quite understand what it does and why you need it. Then after a while, something clicks in your brain, and once that happens you wonder how you ever did without it.

Let's look at a simple example. Say we have the following project directory:

```
├── index.html
├── scripts/
|   ├── main.js
|   ├── require.js
|   ├── jquery.js
|   ├── modules/
|       ├── module1.js
|       ├── module2.js
|       ├── module3.js
```

The project has a single web page, `index.html`, and a `/scripts` directory with JavaScript files. The file `main.js` is our 'main' file discussed shortly. Two 3rd party libraries are used: `require.js` and `jquery.js`. Finally, in subdirectory `/modules` we have 3 modules that contain custom JavaScript code we wrote.

Here is what's in index.html.

```
<!doctype html>
<html>
    <head>
        <title>Title</title>
        <script src="scripts/require.js" data-main="scripts/main"></script>
    </head>
    <body>
        <h2>AMD and Require.js</h2>
        <button id="clicker" >Click here</button>
    </body>
</html>
```

Notice that is has only a single script tag in the entire file which is require.js. This tag includes a special `data-main` attribute that references the main JavaScript file which will be loaded when require.js itself is done loading. As far as Require.js is concerned, the .js file extensions are optional. So `main` means `main.js`, `jquery` means `jquery.js`, and so on. Let's open `main.js` which has configuration information and startup code.

```
require.config({
    paths: {
        jquery: "jquery",
        module1: "modules/module1",
        module2: "modules/module2",
        module3: "modules/module3",
    }
});

require(["jquery", "module1"], function ($, mod1) {

    // hook button up with click event
    $("#clicker").on('click', function () {
        mod1.go();
    });
});
```

The top half is the configuration section. Require.js offers dozens of configuration options but the one that is almost always used is `paths`. The `paths` property is a hash that

maps shorthand names (i.e. script IDs) to JavaScript file paths. This greatly simplifies using Require.js as these names are referenced throughout your files.

In our example the name `jquery` maps to jquery (which is jquery.js: remember that .js is optional). There's no directory mentioned which implies that jquery.js resides in the same directory as main.js. Next are our modules: `module1` maps to modules/module1.js, `module2` to modules/module2.js, and `module3` to modules/module3.js. By the way, we could have used any name for these modules, for example: `modA`, `modB`, and `modC`; they are just handles that are referenced throughout the app.

As an alternative you can import jQuery from a CDN. The configuration section would look like this:

```
require.config({
    paths: {
        jquery: "http://ajax.googleapis.com/ajax/libs/jquery/1.8.2/jquery.min",
        module1: "modules/module1",
        module2: "modules/module2",
        module3: "modules/module3",
    }
});
```

Before using the paths, let's review how Require.js loads in dependencies. It does this through the `require` function. To load up a script file you would write something like this:

```
require(["jsfile"], function (jsfile) {
    jsfile.doSomething();
});
```

The first argument is an array of dependencies. Here we have just one: `jsfile`. If `jsfile` is not a mapped name (using paths in the configuration section), then it refers to jsfile.js in the same directory as main.js.

The second argument is called the *factory function*. Its parameters typically match the dependency array. Here we just have one. Require.js has loaded jsfile and passes it as an argument to the factory function. Within the function you can be sure that jsfile is loaded

and ready to go. By the way: the name factory function is a reference to the Factory pattern as many of these functions return modules that are *manufactured* by the function.

With Require.js, the script files being loaded can be traditional script files, but more often they are modules defined by the Require.js `define` function (note: the Module pattern is discussed in the Modern Patterns section). Modules are defined like this:

```
define("moduleName", ["jquery", "myapp"], function ($, myapp) {
    // define your module...
});
```

The optional `moduleName` is the name of the module. Usually it is not provided because Require.js will then automatically convert the file name to the module name. The second argument is an array of dependencies. The last argument is the factory function which returns a newly created module. The parameters in this function typically match the dependencies: here there are two: `jquery` and `myapp`.

In summary, the two core functions in Require.js are `require` and `define`; `require` loads dependencies and `define` defines modules. Both are used in our example project. Indeed, it's time to go back to our example and complete the second half of the main.js file. We had looked at the configuration and now we will examine the second half which contains the startup code:

```
require(["jquery", "module1"], function ($, mod1) {

    // hook button up with click event
    $("#clicker").on("click", function () {
        mod1.go();
    });
});
```

With `require` we indicate there are two dependencies: `jquery` and `module1`. The factory function receives references to these, `$` and `mod1`, which are then used inside the body. The jQuery library is used to traverse the DOM, find the button, and attach a click handler. Inside the click handler the go function on `module1` is called.

Let's open the module definition of module1.js.

```
define(["module2"], function (mod2) {

    var go = function () {
        alert("- I am in Module1 \n" + mod2.go());
    };

    return { go: go };
});
```

The `define` method has no module name and therefore defaults to the filename which is `module1`. The dependencies array indicates that module1 has a dependency on a single module: module2. The second argument is the factory function that returns the module (this is the Revealing Module pattern). Inside the `go` function there is a nested `go` call but this time on `module2`. Well, let's open module2:

```
define(["module3"], function (mod3) {

    var go = function () {
        return "-- I am in Module2 \n" + mod3.go();
    };

    return { go: go };
});
```

It's déjà vu all over again. This file is almost the same as module1. Module2 is dependent on Module3. The incoming `mod3` argument is referenced inside the `go` function with another call to a `go` function on `mod3`. Finally, let's see module3:

```
define(function () {
    var go = function () {
        return "--- I am in Module3";
    };

    return { go: go };
});
```

This is the last module in our dependency chain which is without any external dependencies. The define method creates a Require.js module that has module3 as its name.

To summarize: require.config in main.js establishes the configuration settings for Require.js. The `require` function call in the same file establishes the first dependencies by setting up the button event handler. When the button is clicked, module 1 is loaded to execute the handler. But module1 depends on module2, which is then loaded. Similarly module2 depends on module3 which is then loaded.

All these dependencies and the loading of these dependencies are automatically handled by Require.js in a highly efficient and effective manner. Remember also that Require.js does not pollute the global namespace with any of these modules. Quite remarkable if you consider that only a single <script> reference is included in the index.html page.

When running it you'll get the following output.

First the index.html page displays. Clicking on the button shows the alert box with 3 messages, each coming from a different module. This confirms that Require.js has loaded all necessary dependencies for us.

As indicated earlier, AMD and Require.js are a stopgap measure until ES 6 (EcmaScript 6) will be available. Among other things ES6 will introduce the `module`, `import`, and `export` keywords to address asynchronous loading and module dependencies. Unfortunately, it will take some time before it is widely available.

JavaScript Transpilers

By now you are well aware that JavaScript was not designed for large scale development and that the language is lacking many features that are common in mature, object-oriented languages. One way to address this lack of features is by using Design Patterns as explained in the different sections this program.

Another approach is to build a new language with the features that are missing from native JavaScript. The only way to run this code in the browser is to translate and compile (transpile) the code into standard JavaScript that is supported by today's browsers. And this is exactly what they do.

Examples of these source-to-source compiled languages include: CoffeeScript, Dart, ClosureScript, and TypeScript. We will briefly touch on each one of these, but mostly TypeScript because it very beautifully shows how Design Patterns are used to generate the missing features in standard JavaScript. Let's look at each language.

CoffeeScript, an open source project by Jeremy Ashkenas, is a language with a very clean syntax that uses indentation rather than JavaScript's brackets, braces and semicolons. Its emphasis is on code brevity and it frequently takes up only half the size of similar JavaScript code. In most cases it is easy to see how CoffeeScript gets translated to JavaScript and vice versa. CoffeeScript is widely used and is particularly popular with Ruby developers.

Dart is an open source web programming language developed by Google. It has a C-like syntax and supports classes, interfaces, abstract classes, inheritance, type annotations, and more. An internal Google memo has stated that Dart's goal is to solve JavaScript's problems that cannot be solved by evolving the JavaScript language. Whether they will succeed remains to be seen of course. Dart prefers to run in its own VM, but like the other languages it also compiles to JavaScript.

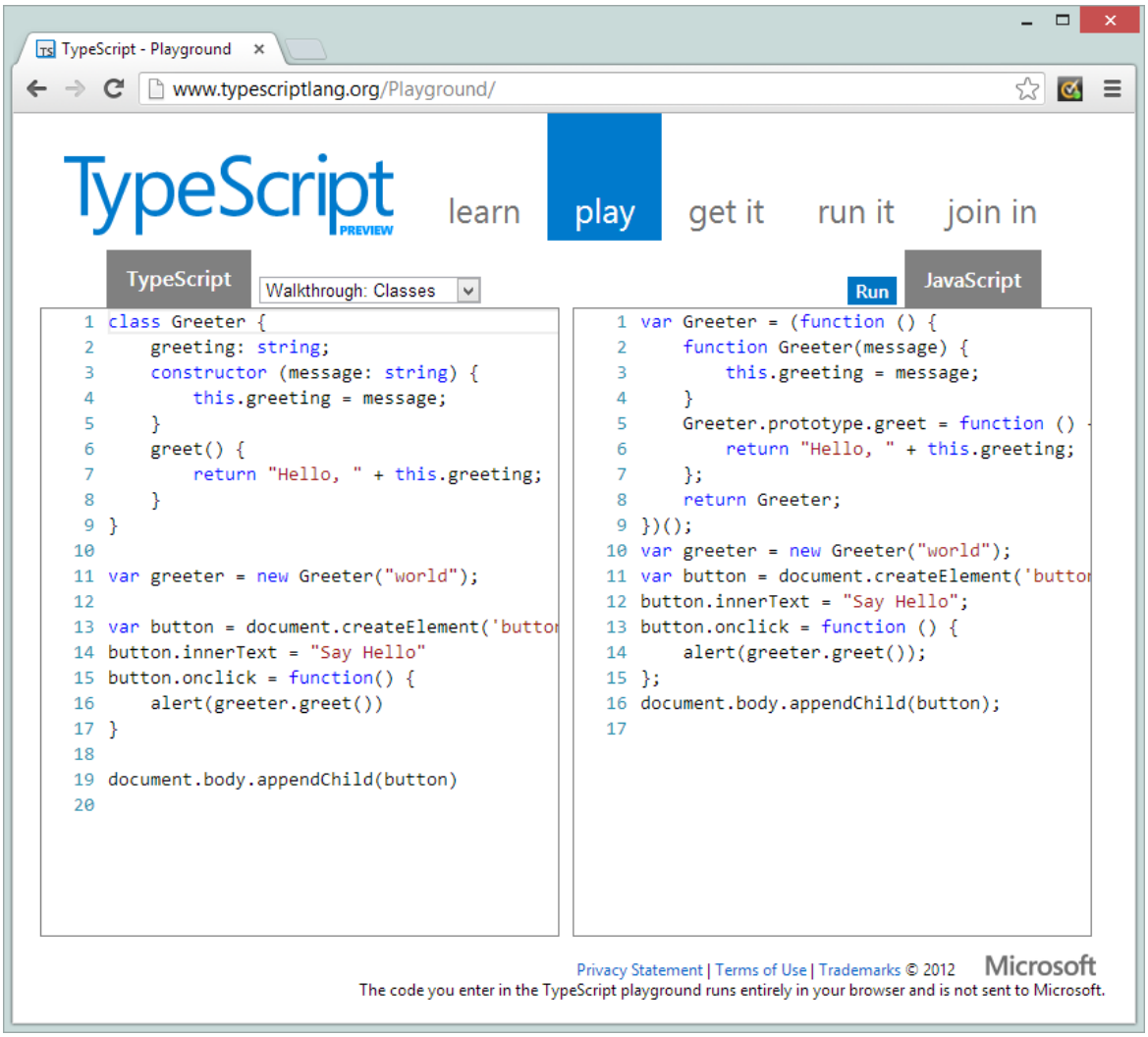
ClojureScript is a subset of Clojure which is an open source project by Rich Hickey. It is a Lisp-like language in which code is treated as data. It has many features found in Common Lisp and also includes a sophisticated macro system. The original Clojure compiles to Java byte code, but ClojureScript transpiles to JavaScript that runs on all browsers and mobile devices.

Finally, **TypeScript** is a language that was designed by Microsoft as an open source project. What is unique about TypeScript it that it is JavaScript itself -- all it adds is some 'syntactic sugar'. This is very different from CoffeeScript, Dart, and ClojureScript because these are unique languages with their own syntax, grammar, etc.

The 'syntactic sugar' of TypeScript includes types, classes, modules, and interfaces, but ultimately these get translated to pure JavaScript which will run on any OS and any platform or browser that runs standard JavaScript. This makes TypeScript particularly interesting because it allows us to easily study the mappings between missing language features (classes, modules, etc.) and pure JavaScript we use today.

We will see in the following paragraphs that TypeScript gets translated to JavaScript code that makes extensive use of Design Patterns. All of these patterns should be familiar to you because they have been discussed in prior sections.

The TypeScript team has made it very easy to compare TypeScript against the translated JavaScript. Simply visit their website at www.typescriptlang.org and select the play menu item. This will open a playground page where the TypeScript and the resulting JavaScript is displayed side-by-side. Here is a screenshot:



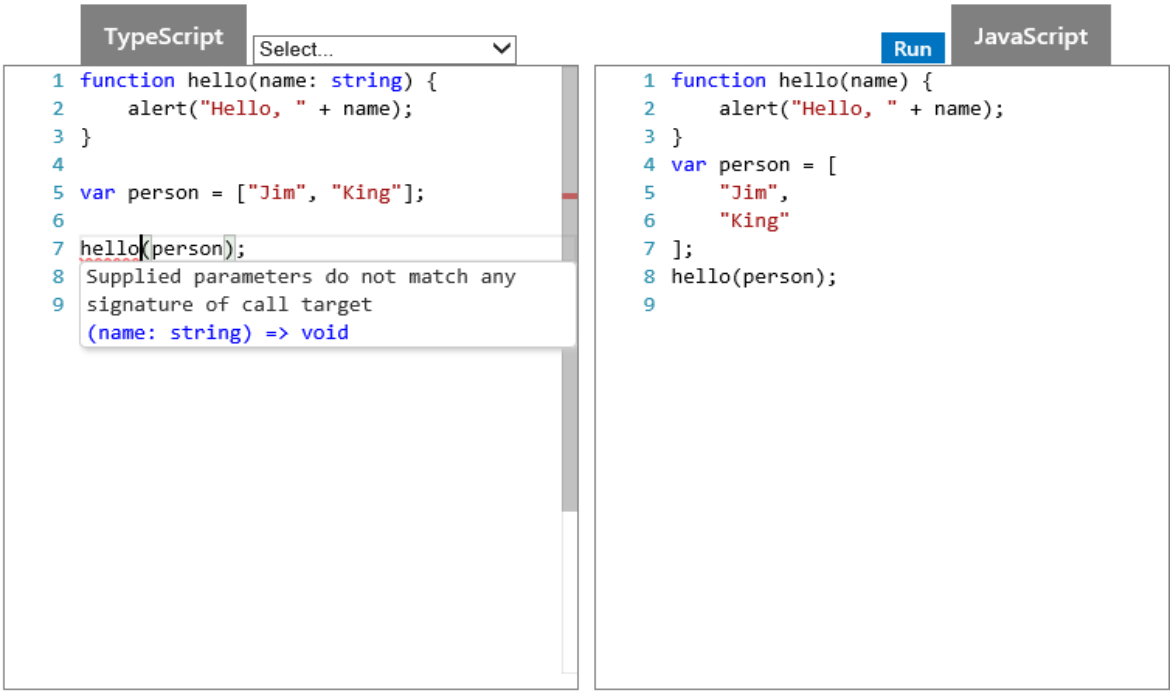
This screen offers instant gratification: you enter TypeScript on the left as you enter the code it gets immediately translated into JavaScript on the right. At the very bottom of the page it states that the playground runs entirely in the browser and no server calls are made. So the TypeScript compiler runs real-time on the client, which is rather impressive.

The dropdown above the TypeScript text box has some walkthrough examples of different features that you can select and study. The image above shows classes; other examples available are for types, modules, inheritance, and a ray tracer example that includes statics, constructors, exports, interfaces, and more.

Next we will examine several TypeScript-to-JavaScript translations. These include the following features and concepts: typing, `class`, `constructor`, `inheritance`, `module`, `interface`, `export`, `private`, and `public`.

Typing

Type annotations are an important part of TypeScript. They allow the compiler to enforce the intended contract between the code and a function or variable. Here is an example:

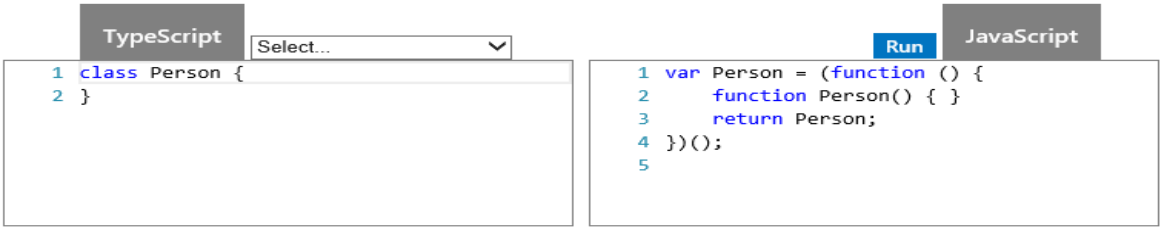


The parameter in the function `hello` has a type annotation of `string`. This informs TypeScript that only strings are allowed into `hello` function calls. However, the argument named `person` is an array and the compiler detects the problem. The error message states that the "Supplied parameter does not match the signature of the target", which is indeed the case.

Notice that the JavaScript on the right does not have any trace of the type. Typing is only used by the TypeScript editor and compiler to ensure that correct types are used at code and compile time, but the JavaScript output files themselves are not affected. Again, typing is a key aspect of TypeScript and is very helpful in building a robust IDE (integrated development environment), but for our purposes we will not focus on this too much.

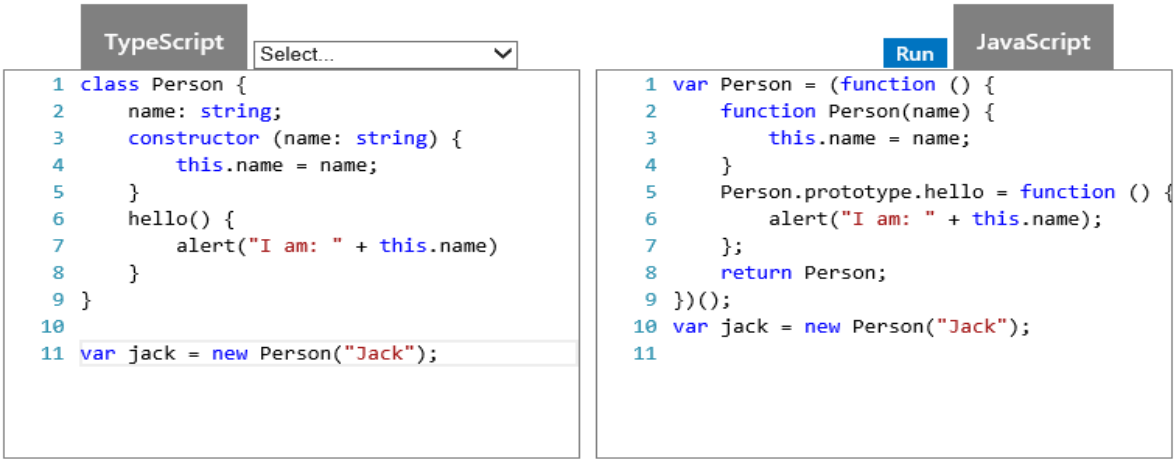
class

TypeScript supports classes although JavaScript is a class-less language. We will start off with an empty class.



The class gets mapped to an immediate function in JavaScript. It actually uses the Revealing Module pattern because the nested function named Person is returned. TypeScript considers this nested function the default constructor that has no parameters.

Next we'll add a typed property called `name`, a `constructor` that accepts a `name`, and a method named `hello`:



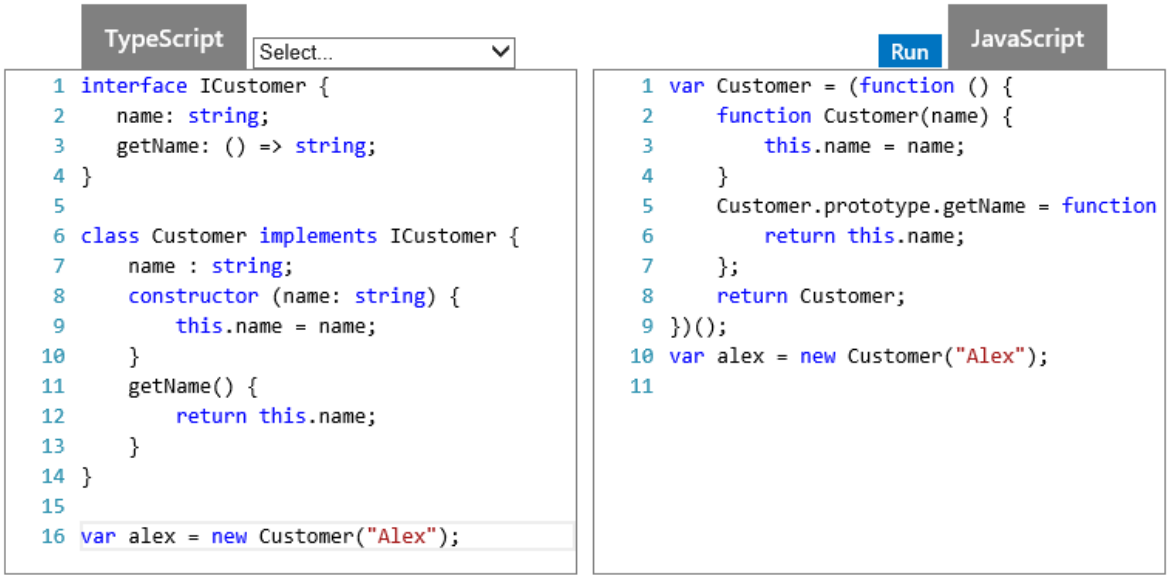
The typed string disappears and does not get translated. TypeScript uses it to verify that the name property and the incoming `name` argument in the `constructor` are of the same type.

The constructor maps to a private function in JavaScript with the same name. It accepts a `name` argument which is then assigned to a name property. The hello method is added to `Person`'s prototype allowing all person instances to share this method. The person instance is returned. Notice that the `Person` prototype method is assigned inside the Person

immediate function which actually is a nice way to encapsulate the Person's code. The generated code appears to be robust and highly effective.

interface

Interfaces are another feature supported by TypeScript. We'll look at these next:




```typescript
interface ICustomer {
   name: string;
   getName: () => string;
}

class Customer implements ICustomer {
    name : string;
    constructor (name: string) {
        this.name = name;
    }
    getName() {
        return this.name;
    }
}

var alex = new Customer("Alex");
```

```javascript
var Customer = (function () {
    function Customer(name) {
        this.name = name;
    }
    Customer.prototype.getName = function
        return this.name;
    };
    return Customer;
})();
var alex = new Customer("Alex");
```

The interface `ICustomer` has two members: a property of type `string` and a method which takes no arguments and returns a `string`. The `Customer` class implements this `interface`. TypeScript ensures that the interface members, `name` and `getName`, are indeed implemented in the class. If not, it shows a red underline under the word `class`.

Just like type annotations, interfaces are not translated to JavaScript; they are merely used to verify complete and valid implementations of derived classes.

module

Next are modules, which are interesting:

TypeScript

Select...

```typescript
module MyApp {
    export class Customer {
        name: string;
        constructor (name: string) {
            this.name = name;
        }
        hello() {
            alert("I am " + this.name);
        }
    }
}
var kelly = new MyApp.Customer("Kelly");
```

Run

JavaScript

```javascript
var MyApp;
(function (MyApp) {
    var Customer = (function () {
        function Customer(name) {
            this.name = name;
        }
        Customer.prototype.hello = functic
            alert("I am " + this.name);
        };
        return Customer;
    })();
    MyApp.Customer = Customer;
})(MyApp || (MyApp = {}));
var kelly = new MyApp.Customer("Kelly");

```

We have a module named `MyApp` that exports a `Customer` class. The class itself is the same as the one we saw before. At the bottom we create an instance of `Customer`, but prefixed with the module name.

On the right we have a global variable `MyApp` which is passed into the immediate function that creates the `Customer`. Inside we have the `Customer` immediate function which is then assigned to the `MyApp` function argument, which is the module name. The internals of the Customer immediate function are the same as we have seen before.

Notice the argument into the 'module' immediate function, which is `(MyApp || (MyApp = {})`. This ensures that `MyApp` exists and is not overwritten in case it was already defined elsewhere.

We have two nested Module patterns: one for the module and another for the class. The class is the Revealing Module pattern because it reveals (returns) the private constructor function of the same name. The module name `MyApp` turns into the root name of the namespace, hence we are also seeing the Namespace pattern in action. It is quite interesting to see how all these patterns fall into place.

Inheritance

Finally, we will review inheritance.

TypeScript

Select...

```typescript
class Person {
    constructor(public name) {}
    hello() {
        alert("Hi I am " + this.name);
    }
}

class Employee extends Person {
    constructor(name, public salary){
        super(name);
    }
    show() {
        alert("I make " + this.salary);
    }
}
```

Run

JavaScript

```javascript
var __extends = this.__extends || function (d, b)
    function __() { this.constructor = d; }
    __.prototype = b.prototype;
    d.prototype = new __();
};
var Person = (function () {
    function Person(name) {
        this.name = name;
    }
    Person.prototype.hello = function () {
        alert("Hi I am " + this.name);
    };
    return Person;
})();
var Employee = (function (_super) {
    __extends(Employee, _super);
    function Employee(name, salary) {
        _super.call(this, name);
        this.salary = salary;
    }
    Employee.prototype.show = function () {
        alert("I make " + this.salary);
    };
    return Employee;
})(Person);

```

In TypeScript we have a `Person` base class and an `Employee` class which extends `Person`. The Person `constructor` has a `name` parameter with a `public` access modifier. What this does is it automatically creates an object property with the same name, like `this.name` (you can see this on the right). This works because in JavaScript all object properties are public. But, interestingly, if you were to prefix it with `private` then you would see exactly the same (give it a try). The only difference is that TypeScript will ensure that private members are not externally accessed. Finally, having no access modifier leaves the assignment to the developer, thus, `this.name = name;` is not automatically created.

`Employee` extends `Person`. The TypeScript Employee `constructor` accepts another public parameter named `salary`. The base class's constructor is called with `super(name)`. `Employee` also adds its own method, named `show` which will display an employee's salary. Let's switch to the right and examine the generated JavaScript.

For inheritance to work, TypeScript provides an `__extends` function you can see at the top right. This function first checks if it already defined elsewhere; if not it create a new function. `__extends` is similar to the `inherit` function we have seen under the Mixin Pattern in the Modern Patterns (note: this function is also available as `Object.create` in newer browsers). The difference is that this function has a `this.constructor = d;` statement (by the way `d` = derived class and `b` = base class). If you have read the section

on prototypes it will be clear that this statement is actually very desirable because it ensures that the constructor property is set correctly in all derived objects.

The generated `Person` class is like all others we have seen before. The `Employee` is different because it inherits from `Person`. It accepts the base class as a parameter named `_super` in the immediate function. The `Employee` immediate function make a call to `__extends` with two arguments: the derived class and the base class, that is, `Employee` and `Person` (called `_super`). The `Employee` 'constructor' function also makes a call to `_super` to set the `name` argument. After that the `salary` is assigned to its own property. This line was also auto-generated because `salary` is prefixed with the `public` keyword in TypeScript.

This completes our review of TypeScript and how it gets translated to standard JavaScript. What we like about this language is that it so clearly validates the notion that Design Patterns are required to getting JavaScript up to a feature level that is generally only available in mature, object-oriented languages.

Whether you use TypeScript in your own work is left up to you. There are numerous JavaScript developers who love these languages that get transpiled to JavaScript because it allows them to increase productivity and the quality and robustness of the JavaScript code they produce. We suggest you experiment with these and see if there is one that meets your needs.

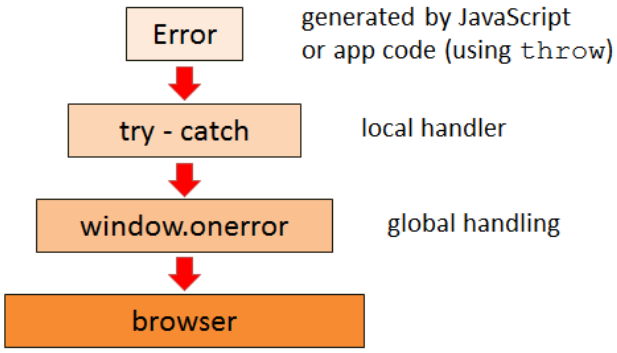
Error Handling

JavaScript has built-in support for error handling (also called *exception handling*). When an error occurs in the code, the JavaScript engine will create an `Error` object with a error message about the particular error and raises it (called *throwing an error*). Next, JavaScript will look for the nearest `try-catch` block and if it finds one the `Error` object gets passed to the `catch` block (called *catching the error*). Your code can decide how to recover from the error, but at that point JavaScript considers the error to have been handled.

If JavaScript does not find any `try-catch` blocks it will try to locate a global `window.onerror` event handler. If there is one then the `Error` gets passed to this handler and

if not then the `Error` continues on to the browser. A return value of true in `window.onerror` means the error was successfully handled and the browser does not get involved. If false is returned the `Error` will continue on to the browser.

Graphically we can depict the error flow in JavaScript as follows:



JavaScript supports a number of built-in error types. The default error type is `Error`, but there are also 6 specialized error objects: `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError`. If none of these are applicable to your project, then you can also create your own custom error type by extending the standard `Error` type.

JavaScript allows you to raise (or throw) your own error using the throw keyword, like so:

```
throw new Error("The start date is missing.");
```

Raising your own error will also start the same error flow we saw before: `try-catch` -> `window.onerror` -> `browser`.

As a developer you can catch and handle errors at two different levels: local and global. We will look at each in detail.

Local error handling

Local errors are processed by the nearest `try-catch-finally` block in which the error occurred. Here is an example:

```
try {
    a = b;
}
catch(e) {
    alert(e.name);      // => ReferenceError
    alert(e.message);   // => 'b' is not defined
}
finally {
    alert("I will always execute");
}
```

In this example the `try` block will attempt to execute the code within its curly braces. It will fail because none of the variables are declared or initialized. The JavaScript engine will create an `Error` object with an error message and raise it (throw it). JavaScript will search the stack for the first `try` block and passes the `Error` object to the associated `catch` block. The `catch` block receives the `Error` object and starts executing.

The `Error` object has two properties that are supported by all browsers: `name` and `message`. In the above example the name shows `ReferenceError` which means that the error type is one of the built-in types. The message property contains the actual error message string.

The `finally` block is optional but if one is included it always gets executed -- irrespective whether an error occurred or not. A common use for `finally` blocks is to clean up resources, such as deleting an object instance or closing a server connection.

As mentioned before the exception is considered processed once the `catch` block has completed execution. In the code above the `catch` block displays the error type and error message which is not exactly a good example of proper error handling. Usually some corrective action is necessary which will allow the user to continue what they were working on.

There are cases in which you want to log the error, but, you don't want the error to be considered handled and just disappear. This is the case when you want to log the details

of the error but no obvious corrective action is possible (by the way, error logging is discussed shortly). You can do this by 'rethrowing' the error in the `catch` block like so:

```
var productId = 44;
try {
    a = b;
}
catch(e) {
    log(1, "Error: " + e.message + " Product: " + productId);
    throw e;      // re-throw the error
}
```

This is a common pattern. The advantage is that you can add contextual information to the log that is only available at this location such as the product the user was working on. This information would be lost if the error is handled at a later stage. Once the error is logged the `Error` immediately gets re-thrown. It then is handled by the next `try-catch`, `window.onerror`, or the browser.

Developers that are new to `try-catch` tend to overuse it because they mistakenly assume it makes their program more robust and fool-proof. This is not exactly true. The rule is that you only implement a `try-catch` block if the error can be corrected, that is, the user can continue what it was doing (ideally without realizing that anything was wrong). If not, there is usually no point in handling it. In fact, catching errors and not handling them will give the impression that everything is fine; but it is not and this can result in hidden bugs.

Sometimes you run into code that uses a `try-catch` block as a replacement for an if-statement, like so:

```
try {
    process();  // May or may not exist
}
catch(e) {
    // ...
}
```

This is not a good idea. First of all, the overhead of `try-catch` is fairly large and many try-catch statements may slow the program. Secondly, it gives the false impression that we are dealing with an *unexpected* exception which, in the example above, is not the case. It can easily be rewritten with an `if` statement that tests whether process is a function or not.

```
if (process && typeof process == "function")
    process();
}
```

It is generally recommended to use `try-catch` sparingly and only when you are uncertain what conditions may occur, but you are certain that the code is able to recover from the error condition. An example would be invoking a 3rd part web service. If an unexpected error occurs you can try to re-connect and re-invoke but if that is unrealistic or not possible do not use `try-catch`.

Let's see how seasoned developers use `try-catch`. Analyzing the jQuery source code, for example, reveals that there are only about a dozen `try-catch` instances in roughly 10,000 lines of code. They mostly deal with browser and/or feature detection (typically IE) and many of their exceptions are handled silently, meaning there are no corrective actions taken in their catch blocks. The code looks like this:

```
try {
    ...
}
catch (e) { /* no action */ }
```

In the `try` block you may find code that attempts to access a certain browser specific feature. However, if the feature does not exist an exception gets throws. The code will then fall back to the default way of processing of whatever is was trying to do and nothing needs to be done in the `catch` block.

Next we will look at the `throw` keyword. Raising your own errors may seem strange because why would you purposely cause your program to fail? Let's take the example of parameter validation. Say you write an add function that adds two numeric values. In the

function you verify that both incoming arguments are numeric and if not you raise a `TypeError`, like so:

```
function add (x, y) {
    if (typeof x !== 'number') throw new TypeError("x is not numeric");
    if (typeof y !== 'number') throw new TypeError("y is not numeric");
    return x + y;
}

try {
    alert(add(4, 5));         // 9
    alert(add(4, "dog"));     // Error
} catch(e) {
    alert("Error: " + e.message);
}
```

This will prevent anyone from calling add with non-numeric values, which in turn keeps your function from malfunctioning.

In response to the above function you may think that all add calls should be wrapped in a `try-catch` or else unhandled exceptions may occur, but that is usually not a good idea. A much better approach is to proactively validate the argument date *before* it gets passed to the add function.

For example, if a user enters two values that are going to be added, you should check that these are numeric *before* handing them over to add: if they are not numeric then give the user the opportunity to make a correction. This has two advantages: 1) it creates a much better user experience and 2) the code will be simpler, cleaner, and more performant rather than having to write code trying to recover from an error condition.

If your application is both the producer and the consumer of the add function them the throw statements are probably overkill because you know the data that ultimately gets passed to add. However, if you are building a library or a public API that gets used by others then the above approach of validating the argument types and raising errors is highly appropriate. In these situations you have absolutely no control over what gets passed into the add function and you need to protect yourself against erroneous values. So, when throwing errors always consider who the end-user is.

Global error handling

Next we'll look at the global error handling with the `window.onerror` event handler. First off, note that this is a DOM handler which is not part of JavaScript language itself. The `window.onerror` handler offers a last chance to handle uncaught runtime errors before they get sent to the browser. As an aside, some older versions of Chrome and Safari do not support `window.onerror`, but more recent versions do.

All uncaught exceptions of an app are sent to `window.onerror`, so expect to see a wide range of error types. In most cases there isn't much you can do to recover from these errors. At a minimum you should log the error to the server so you know what is happening on the client side of your app (note: error logging is discussed in the next section).

Three arguments are passed to onerror: an error message, the url of the file that contains the script, and a line number of the code where the error occurred. If `onerror` returns a value of true then JavaScript considers it handled. A return value of false means the error will be passed on to the browser which, depending on the browser and the error severity, may or may not display an error dialog. Here is an example of onerror that logs the error to the server:

```
window.onerror = function (message, url, line) {
   log(2, "window.onerror: message: " + message +
          "url: " + url +
          "line: " + line);

   return true;
}
```

Of course, you should never display the raw error message to the user which would create a bad user experience. In most cases you just silently log the error and then return true. For extra safety you may consider placing the logging operation in its own `try-catch` block to avoid triggering another unhandled error if the script or the connections are really messed up.

```
window.onerror = function (message, url, line) {
   try {
```

```
        log(2, "window.onerror: message: " + message +
               "url: " + url +
               "line: " + line);
    }
    catch (e) { /* do nothing */ }

    return true;
}
```

Actually, an even better approach is to include the `try-catch` in the `log` function, because error logging is mostly not mission-critical. We will discuss this further in the next section.

Here is an example of a runtime error. It has an alert which you should never do, but this is for demonstration purposes only.

```
window.onerror = function (message, url, line) {
    alert( "window.onerror: message: " + message +
           "url: " + url +
           "line: " + line);

    return true;
}

execute();        // function does not exist
```

Error logging

It is important to log JavaScript errors to the server or otherwise you are really flying blind. There is no other way to find out what is happening on the client in your app. A clever way to log to the server is with code similar to this:

```
function log(level, message) {
    var image = new Image();
    image.src = "log/" + EncodeURIComponent(level) +
                  "/" + EncodeURIComponent(message);
}
```

This code uses the fact that images can be dynamically retrieved from the server by providing a url (the `src` attribute). To log the error details all we need is a mechanism that triggers a method to execute on the server; and this does the job. The actual image object is never used and is immediately discarded.

The first argument in `log` is the level (or severity) of the error. Usually the value ranges from 1 to 5 with 1=fatal, 2=error, 3=warning, 4=info, 5=debug. Recall that we have seen the same log function being used in the earlier `window.onerror` function. It had the `log` function wrapped in a `try-catch`, but it would be better to include this in the `log` function itself so that errors do not cause other, possibly recursive, errors.

```
function log(level, message) {
    try {
        var image = new Image();
        image.src = "log/" + EncodeURIComponent(level) +
                    "/" + EncodeURIComponent(message);
    }
    catch (e) { /* no action */ }
}
```

This version of the `log` function, may fail, but it never causes an error itself.

Another useful enhancement would be to create an asynchronous, non-blocking logging version of this function. This one will have minimal performance impact. Using jQuery makes this easy. It looks like this:

```
function log(level, message) {
    try {
        $.get ("log/" + EncodeURIComponent(level) +
                  "/" + EncodeURIComponent(message) );
    }
    catch (e) { /* no action */ }
}
```

Here is example of logging an error in a `try-catch` block.

```
try {
   processData();
```

```
}
catch(e) {
   log(2, " processData() failed. Error: " + e.message);
}
```

If the call to processData causes an error it will be caught and logged by our logging system. However, this example is most likely incomplete. As you know an error caught by a `try-catch` it is considered handled by JavaScript, but in this case we only log the error and nothing else is done with it. This can cause hidden bugs. Perhaps this is okay but most likely something needs to be added: either a corrective action or a re-throw of the error in the catch block.

Errors and User Experience

The primary reason we have error handling is to improve the robustness and reliability of the code which, in turn, improves the user experience. Applications that have errors are annoying and in today's marketplace users have no tolerance for shoddy apps. If an app does not appear reliable users will walk away from it never to come back. As a developer you have pride in your work and you will not be satisfied when users are not happy with your work. It also directly affects the bottom-line, so it is important you take error handling seriously.

There are two types of errors: *non-fatal* and *fatal*. Non-fatal errors are the ones you can recover from. These are mostly handled by try-catch blocks and in the ideal situation your code is able to recover to a stable state and the user never knew that something went wrong. As long as the user is able to continue their work, it is best to suppress any error notifications. If this is not possible then your error is probably of the fatal (or catastrophic) type.

Examples of fatal errors include: database not available, server not available, a web service that is offline, etc. The app simply cannot function without these and the user will not be able to continue their work. A good way to handle these errors is to immediately inform the user and attempt to reload the page.

Of course, your recovery methods and messaging to your user base also depends on the reliability and architecture of your servers, as well as that of the external web services. It

is important that your user is informed about the error and possible ways to resolve this. In the worst case you explain a server is down and ask them to come back in the few moments and then try again. What you should avoid is to let the browser handle the error and show something over which you have no control. Again, when dealing with errors always consider the user experience.

Testing JavaScript

Testing is a large topic and a lot has been written about it. Having experience with testing in other languages will be helpful when starting out with JavaScript testing because many of the same principles apply. In this section we will review two of the more popular testing frameworks and how to use these in your projects.

Before getting into the details of testing frameworks we will first introduce the concept of Debug Mode which is an often overlooked aspect of testing (and tracing) your code.

Debug Mode

You will recall that the log function discussed earlier supports 5 levels of severity: 1=fatal, 2=error, 3=warning, 4=info, 5=debug. The debug value (number 5) offers a powerful way to intersperse your code with debug statements that only get logged when the app is in 'debug mode' (also called 'trace mode').

We will use a numeric `logLevel` variable which indicates the severity levels that are saved to the database. For example a 3 would save level 1, 2, and 3 errors (fatal, error, and warning), a 1 would save fatal errors only, and a 5 would save 1, 2, 3, 4, and 5 levels (which is 'Debug Mode'). Using `logLevel` offers a simple way to throttle the number of error message being stored in the database. In production you would typically limit it to level 1 or 2 whereas in production you could leave it at 4 or 5.

Here is the implementation:

```
var logLevel = 3;

function log(level, message) {
    try {
```

```
        if (level <= logLevel) {
            $.get ("log/" + EncodeURIComponent(level) +
                      "/" + EncodeURIComponent(message) );
        }
    }
    catch(e) { /* no action */ }
}
```

This works, but a better way is to place it in its own module and namespace to reduce your global footprint. Alternatively, place it in your own namespace to leave no additional globals at all.

```
var logging = (function() {
    var logLevel = 0;
    var log = function (level, message) {
        try {
            if (level <= logLevel){
            $.get ("log/" + EncodeURIComponent(level) +
                      "/" + EncodeURIComponent(message) );
            }
        }
        catch (e) { /* no action */ }
    };
    return {
       logLevel: logLevel,
       log: log
    }
}());
```

Where necessary you can now intersperse your code with debug statements allowing you to trace the execution of your code. Here is an example of a complex multistep calculation:

```
logging.logLevel = 5;

function complexQuantumCalculation(start) {

    logging.log(5, "Just before Step1: value = " + start);
    var a = quantumCalculationStep1(start);

    logging.log(5, "Just before Step2: value = " + a);
```

```
    var b = quantumCalculationStep2(a);

    logging.log(5, "Just before Step3: value = " + b);
    var c = quantumCalculationStep3(b);

    logging.log(5, "Just before Step4: value = " + c);
    var d = quantumCalculationStep4(c);

    logging.log(5, "Final value = " + d);
    return d;
}
```

This code will log all intermediate calculation steps in the log database which can be double checked and validated. Once you are satisfied with the calculations you could possibly remove some of these level 5 log statements.

As an aside, it is important to have some utility or app that developers can use to quickly view, filter, and manage the log entries in the database without having to manually issue queries to the database.

Testing Frameworks

Determining which JavaScript testing framework to use in your project can be a daunting task: there are literally dozens to choose from. Many of these are good and at the end it comes down to personal preference. We will focus our discussion on a testing framework called *Jasmine*.

Jasmine

Jasmine is a BDD testing framework. What is BDD? BDD is an extension of TDD which stands for Test Driven Development. The basic idea of TDD is that your development cycle is test-driven, meaning you write your tests before you actually start writing code. Doing it this way forces you to think upfront about your program's design, structure, and API rather than just starting off coding. Experiments have shown that the resulting code is usually markedly different (and better structured) from what would have been produced otherwise.

There are some limitations with TDD which BDD tries to address. For example, TDD does not tell you *what* should be tested and *how*. The focus in BDD is testing the desired behavior of the software, that is, those elements that add business value by concentrating on the requirements and user stories that need to be implemented by the code. A full discussion of TDD is beyond the scope of this discussion but we will be able to show you how to setup some basic Jasmine JavaScript testing.

Jasmine does not require the DOM, so it can run without a browser. Many different environments are supported, such as .NET, Java, Ruby, node.js, etc. This allows Jasmine tests to easily integrate with a Continuous Integration (CI) environment. When using Jasmine with CI you always have access to the latest builds and you get regular feedback on the state of your automated tests.

For demonstration purposes we will keep it simple and run the Jasmine tests in the browser. Let's get started writing a simple test:

```
describe("Javascript built-in functionality", function () {
    it("concatenates two strings together" , function () {
        expect("Hello " + "Test").toBe("Hello Test");
    }
}
```

The syntax of Jasmine is clean and feels natural which makes it easy to use. Notice how the above code almost reads like a sentence: "*Describe JavaScript built-in functionality: it concatenates two strings together*". Next you state what your expectations are: in this case, you expect that concatenating "Hello " + "Test" to be equal to "Hello Test". Again, it reads like a sentence: Expect "*Hello " + "Test" to be "Hello Test*";

Let's examine the functions involved. With `describe` you start a test *suite*. It can have multiple `it` functions and all of these together are called a suite. The `it` function declares what is called a *spec* which consists of a short description of a test and a test function. Within the test function you are testing a value or expression that is passed as an argument into `expect`. The argument is called the *actual*. Then finally, the chained `toBe` method contains the expected outcome. The `toBe` method is called a *matcher* which matches the actual value against the expected value. Jasmine offers many different matchers, some of which we will see later in this section.

Using Jasmine in the browser requires that you download the "standalone release" of Jasmine. You need to include three files (one css and two js files) and an immediate function with Jasmine startup code in your test page. At that point you are ready to start building your tests. The code below shows where to place the code to be tested and the specs (tests). There is no need to put anything in the page body because Jasmine will fill that out for you.

```
<!doctype html>
<html>
<head>
  <title>Jasmine test page</title>
  <link href="/css/jasmine.css" rel="stylesheet" />
  <script src="/js/jasmine.js" type="text/javascript"></script>
  <script src="/js/jasmine-html.js" type="text/javascript"></script>

  <!-- Your code and specs go here -->

  <script type="text/javascript">
    (function () {
        var jasmineEnv = jasmine.getEnv();
        jasmineEnv.updateInterval = 1000;

        var htmlReporter = new jasmine.HtmlReporter();
        jasmineEnv.addReporter(htmlReporter);

        jasmineEnv.specFilter = function (spec) {
            return htmlReporter.specFilter(spec);
        };

        var currentWindowOnload = window.onload;

        window.onload = function () {
            if (currentWindowOnload) {
                currentWindowOnload();
            }
            execJasmine();
        };

        function execJasmine() {
            jasmineEnv.execute();
        }
    })();
   </script>
```

```
</head>
<body>
</body>
</html>
```

Now that we have a skeleton test page we are ready to write a test. In our case we are going to build a calculator. It is a simple, but fully functional calculator that supports chaining, like so:

```
var calculator = new Calculator();

alert(calculator.add(20).sub(5).mul(2).div(5).get());  // => 6
```

Since we are using BDD we will write our tests *before* implementing the calculator. Our calculator will support 4 basic operations: add, sub, mul, and div and two helper methods: get, which returns the current value, and reset, which resets the calculator back to zero. As mentioned before, all operations will be chainable.

Here are the Jasmine test suites and specs:

```
describe("Calculator", function () {
    describe("Simple method calls", function () {

        var calculator = new Calculator();
        it("adds a number", function () {
            expect(calculator.add(13).get()).toBe(13);
        });
        it("substracts a number", function () {
            expect(calculator.sub(5).get()).toBe(8);
        });
        it("multiplies a number", function () {
            expect(calculator.mul(4).get()).toBe(32);
        });
        it("divides a number", function () {
            expect(calculator.div(2).get()).toBe(16);
        });
    });

    describe("Chained method calls", function () {
        var calculator = new Calculator();
```

```
        beforeEach(function () {
            calculator.reset();
        });

        it("adds, substracts a number", function () {
            expect(calculator.add(13).sub(8).get()).toBe(5);
        });
        it("adds, subtracts, multiplies, divides a number", function () {
            expect(calculator.add(6).sub(1).mul(4).div(2).get()).toBe(10);
        });
    });
});
```

We have a test suite called 'Calculator' with two sub suites: one to perform single operations and another to perform chained operations. It is perfectly fine to have nested suites. In fact, this is common practice as it allows you to structure your test suites in a way that works best for your projects. Notice that the JavaScript function scope rules apply, that is, nested functions have access to their parent's variables and properties, so the `it` functions have access to all of `describe`'s variables.

The 'Simple method calls' suite has 4 specs, each one testing a different operation. Between the `it` functions the calculator is not being reset, so each calculation builds upon the prior test. This confirms that all nested it functions have access to a shared calculator maintained by their parent describe function.

The 'Chained method calls' suite has two 2 specs, each one testing a chaining operation. In this situation, we like to reset the calculator between the specs. Jasmine offers a `beforeEach` and `afterEach` method that gets called before and after each spec. We prefer to reset the calculator before each test, so this is what we use. The two specs test chaining of two and four levels respectively.

It is now time to write the calculator:

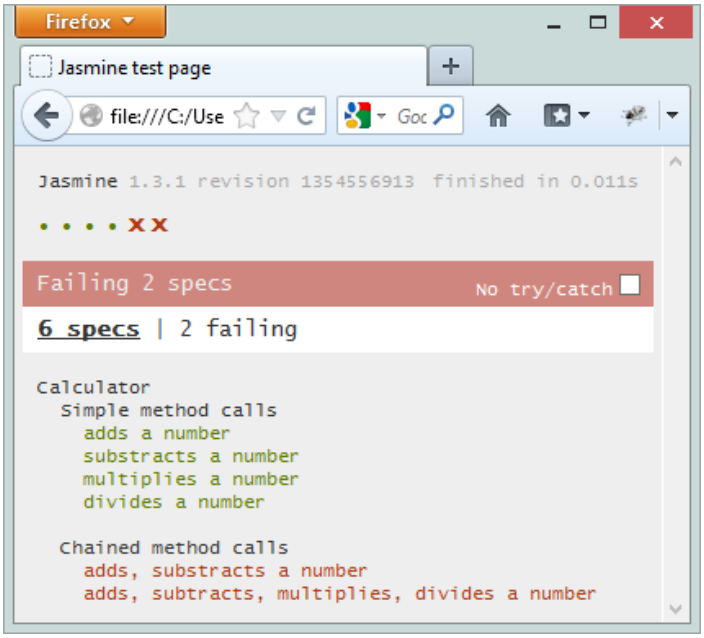
```
function Calculator() {

    var total = 0;

    this.add = function (x) { total += x; return this; };
    this.sub = function (x) { total -= x; return this; };
```

```
    this.mul = function (x) { total *= x; return this; };
    this.div = function (x) { total /= x; return this; };

    this.get = function () { return total; };
    this.reset = function () { total = 0; };
}
```

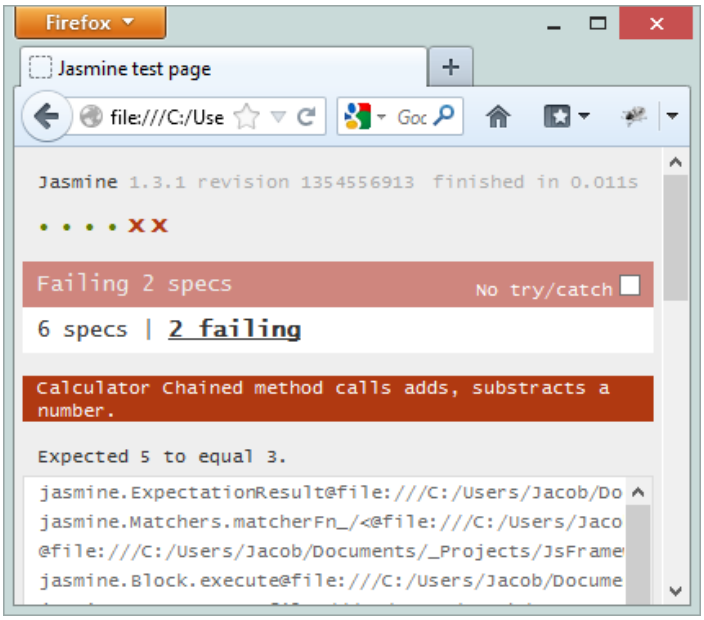
With the calculator and the specs in place we are now ready to run our test suite. Here are the results:

That looks good: all specs passed. Jasmine has a clean interface. Most items on the page are links which allow you to drill down and navigate through all suites and specs. By the way, when you click on the 'Simple methods calls' Jasmine will run the specs individually which will fail for the second, third, and fourth because they depend on prior computations (as explained earlier).

Let's create a couple of failing tests. Here is what a failed test result looks like:



This shows the summary of all 6 specs. Details are visible by clicking on the failing links or the '2 failing' link at the top. In one of the cases it expected a value of 5 but the actual value was 3.



Next we will look at some additional matchers. We have already seen `toBe`. The more common ones are: `toEqual` which compares literals, values, and objects, `toMatch` which finds a match by using regular expressions, `toBeUndefined` which compares against undefined, `toBeNull`, `toBeTruthy`, `toBeFalsy`, `toContain`, `toBeLessThan`, `toBeGreaterThan`, and `toThrow`. All of these are fairly self-explanatory. We will look at `toThrow` in some more detail as this is an important matcher for testing expected failures.

Suppose that we like to tighten our Calculator a little by ensuring that only numeric arguments are used in the operations. If the value is non-numeric then we raise a TypeError exception. Let's first write a Jasmine suite to test this new functionality:

```
describe("Non-numeric argument types", function () {
    var calculator = new Calculator();

    beforeEach(function () {
        calculator.reset();
    });

    it("raises a type error exception (string)", function () {
        var fn = function () {
            return calculator.add("Sixteen").get();
        };
        expect(fn).toThrow("non-numeric value");
```

```
    });

    it("raises a type error exception (object)", function () {
        var fn = function () {
            return calculator.add({ city: "Miami" }).get();
        };
        expect(fn).toThrow("non-numeric value");
    });
});
```

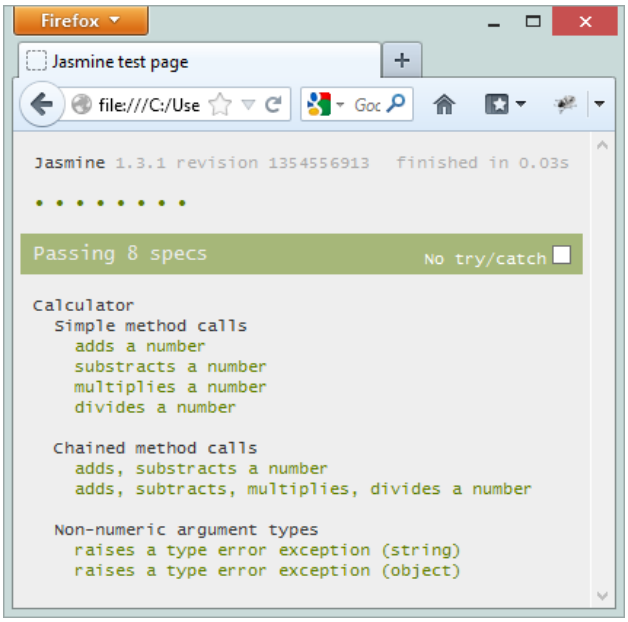
This is fairly similar to the prior suites, but notice that the calculator calls are wrapped in their own function. This function is then passed into `expect`. This is necessary or else `expect` won't be able to properly catch the exception.

Next we have to adjust the Calculator and include the argument check.

```
function Calculator() {

    var total = 0;

    this.add = function (x) { numeric(x); total += x; return this; };
    this.sub = function (x) { numeric(x); total -= x; return this; };
    this.mul = function (x) { numeric(x); total *= x; return this; };
    this.div = function (x) { numeric(x); total /= x; return this; };

    this.get = function () { return total; };
    this.reset = function () { total = 0; };

    function numeric(x) {
        if (!isNaN(parseFloat(x)) && isFinite(x)) return;

        throw new TypeError("non-numeric value");
    }
}
```

Each operation includes a call to `numeric` which is a local function that checks whether the incoming argument value is numeric. If not it raises a `TypeError` exception with a message string "non-numeric value". It is this specific error message that Jasmine will be looking for in the `toThrow` method.

When running this test we are getting the expected results:



Jasmine matchers also have the ability to test negative cases by chaining in a *not* between `expect` and the matcher, like so:

```
expect("Hello " + "Test").not.toBe("Greetings Spec");
```

Finally, we will look at custom matchers. They are easy to write and easy to add. Suppose we want a matcher to confirm that a number is positive. Let's call it `toBePositive`. You can add your custom matcher to a `beforeEach` with a call to `this.addMatchers` using `this.actual`. Here is an example:

```
beforeEach(function () {
    this.addMatchers({
        toBePositive: function() {
            return this.actual > 0;
        }
    });
});
```

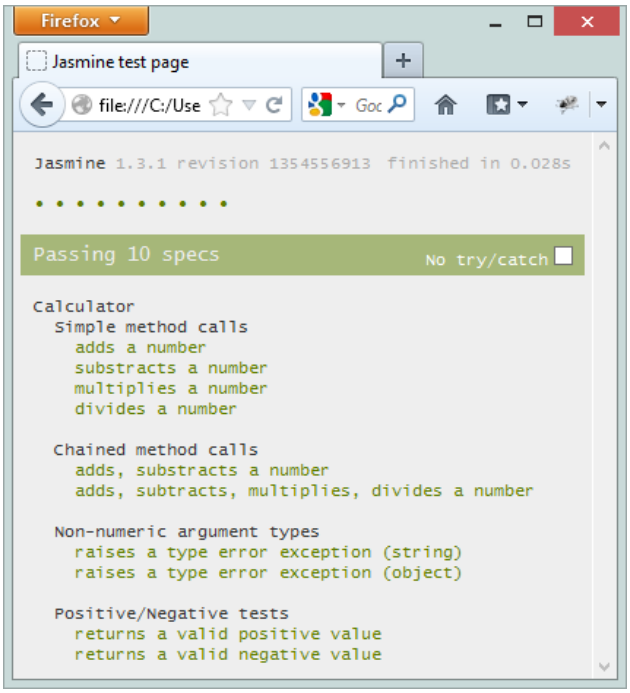
And here is how it is used in our spec:

```
describe("Positive/Negative tests", function() {
    var calculator = new Calculator();
```

```
    beforeEach(function() {
        this.addMatchers({
            toBePositive: function() {
                return this.actual > 0;
            }
        });
    });

    it("returns a valid positive value", function () {
        expect(calculator.add(3).sub(1).get()).toBePositive();
    });

    it("returns a valid negative value", function () {
        expect(calculator.add(3).sub(9).get()).not.toBePositive();
    });
});
```

Notice how we are using the new matcher to test for positive and negative numbers by chaining in the operator. This is very convenient and it prevents us from having to write a separate `toBeNegative` matcher.

These are the results:



Jasmine offers more than we have shown here, but hopefully these and the other examples have given you a sense of the maturity and richness of the JavaScript testing tools available.