UNDERSCORE TEMPLATE ,
JASMINE &
AMD

banuprakashc@yahoo.co.in

Templating libraries

- When you build a JavaScript application, you'll almost certainly use some JavaScript templates.
- Rather than use a library like jQuery (or vanilla JavaScript) to update your HTML when values update, you can use templates, which cleans up your code hugely

Popular templating libraries.

1. Mustache

Mustache is often considered the base for JavaScript templating

```
Mustache.render("Hello, {{name}}", { name: "Jack" });

// returns: Hello, Jack
```

02. Underscore Templates

- Underscore provides simple templates we can use.
- It uses a slightly differet syntax to Mustache. Here's a simple example:

```
_.template("Hello, <%= name %>", { name: "Jack" });
```

- In Underscore templates, you can embed arbitary JavaScript within '<% %>' tags.
- Note that we use '<%= %>' to output to the page, and `<% %>` to contain JavaScript.
- This means any form of loop or conditional you can do in JS, you can use in Underscore.

Popular templating libraries

03. Embedded JS Templates

EJS is different in that it expects your templates to be in individual files, and you then pass the filename into EJS. It loads the file in, and then gives you back HTML.

- // in template.ejs
- Hello, <%= name %>
- // in JS file

new EJS({ url: "template.ejs" }).render({ name: "Jack" }); // returns: Hello, Jack

04. HandlebarsJS

Handlebars is one of the most popular templating engines and builds on top of Mustache.

Anything that was valid in a Mustache template is valid in a Handlebars template.

Handlebars add lots of helpers to Mustache.

Popular templating libraries.

05. Jade templating

Jade templates are very different in that they depend hugely on indents and whitespace

Example:

```
each person in people
    li=person
```

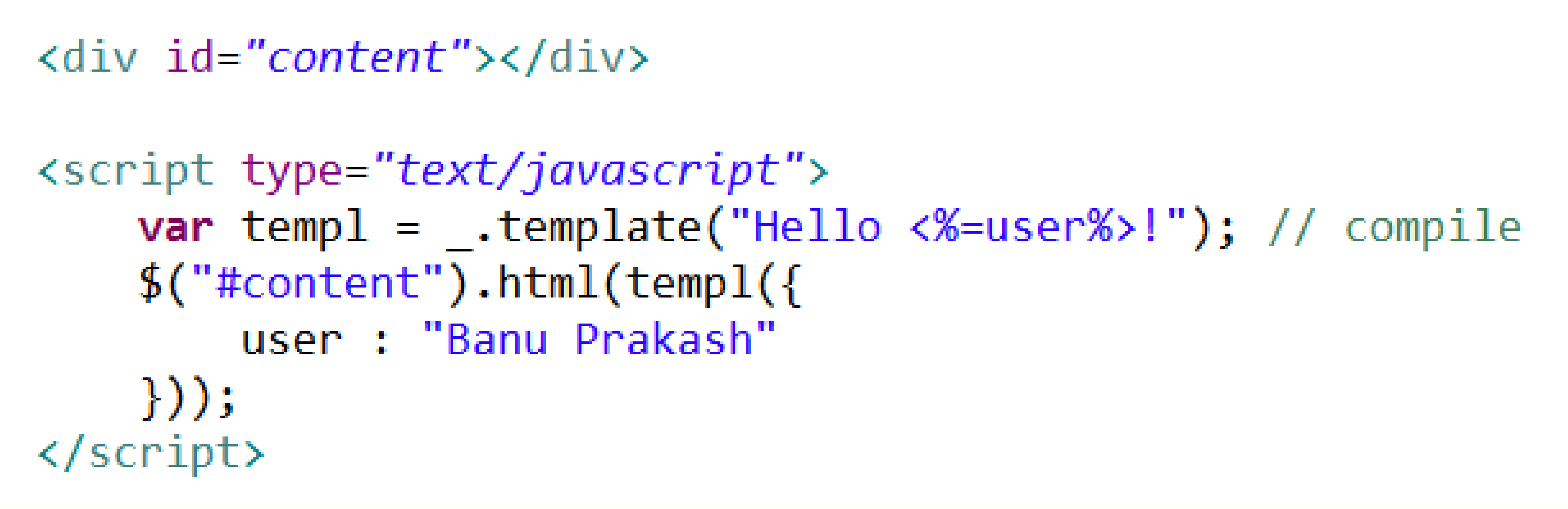
- Called with an array of names: '{ people: ["Jack", "Fred"]}', this will output:
- <li>Jack</li><li>Fred</li>

template _.template(templateString, [settings])

- Compiles JavaScript templates into functions that can be evaluated for rendering.
- Useful for rendering complicated bits of HTML from JSON data sources.
- Template functions can both interpolate values, using <%= … %>, as well as execute arbitrary JavaScript code, with <% … %>.
- If you wish to interpolate a value, and have it be HTML-escaped, use <%- … %>.
- When you evaluate a template function, pass in a data object that has properties corresponding to the template's free variables.
- The settings argument should be a hash containing any _.templateSettings that should be overridden.

Working with Underscore templates

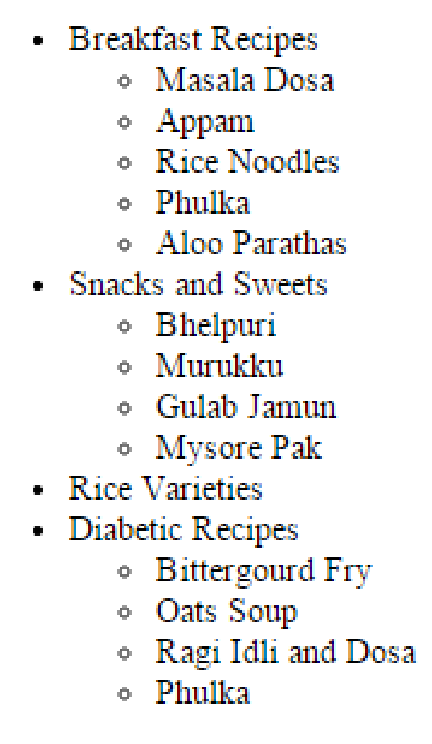
- The template function has the following signature:
- _.template(templateString, data?, settings?)



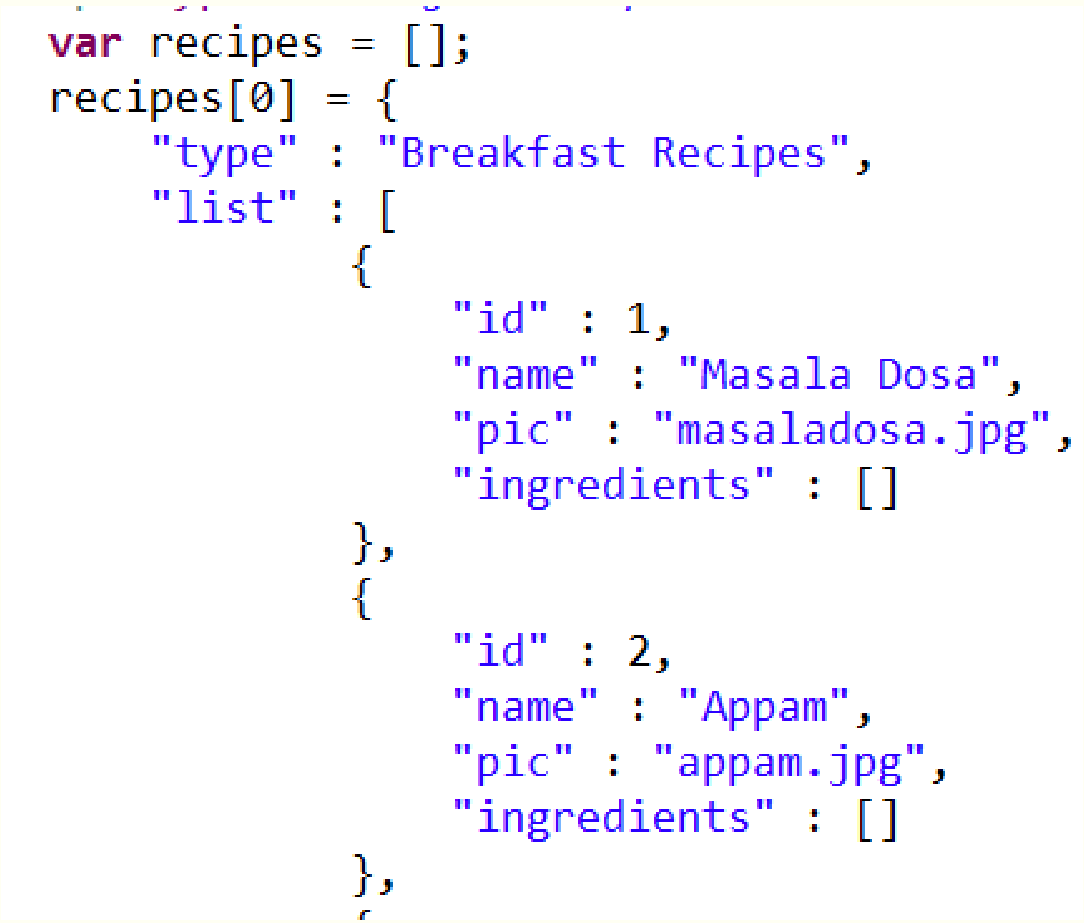
```
<div id="content"></div>

<script type="text/javascript">
    var templ = _.template("Hello <%=user%>!"); // compile
    $("#content").html(templ({
        user : "Banu Prakash"
    }));
</script>
```

Using loops

- Template to print recipes and list of recipes items

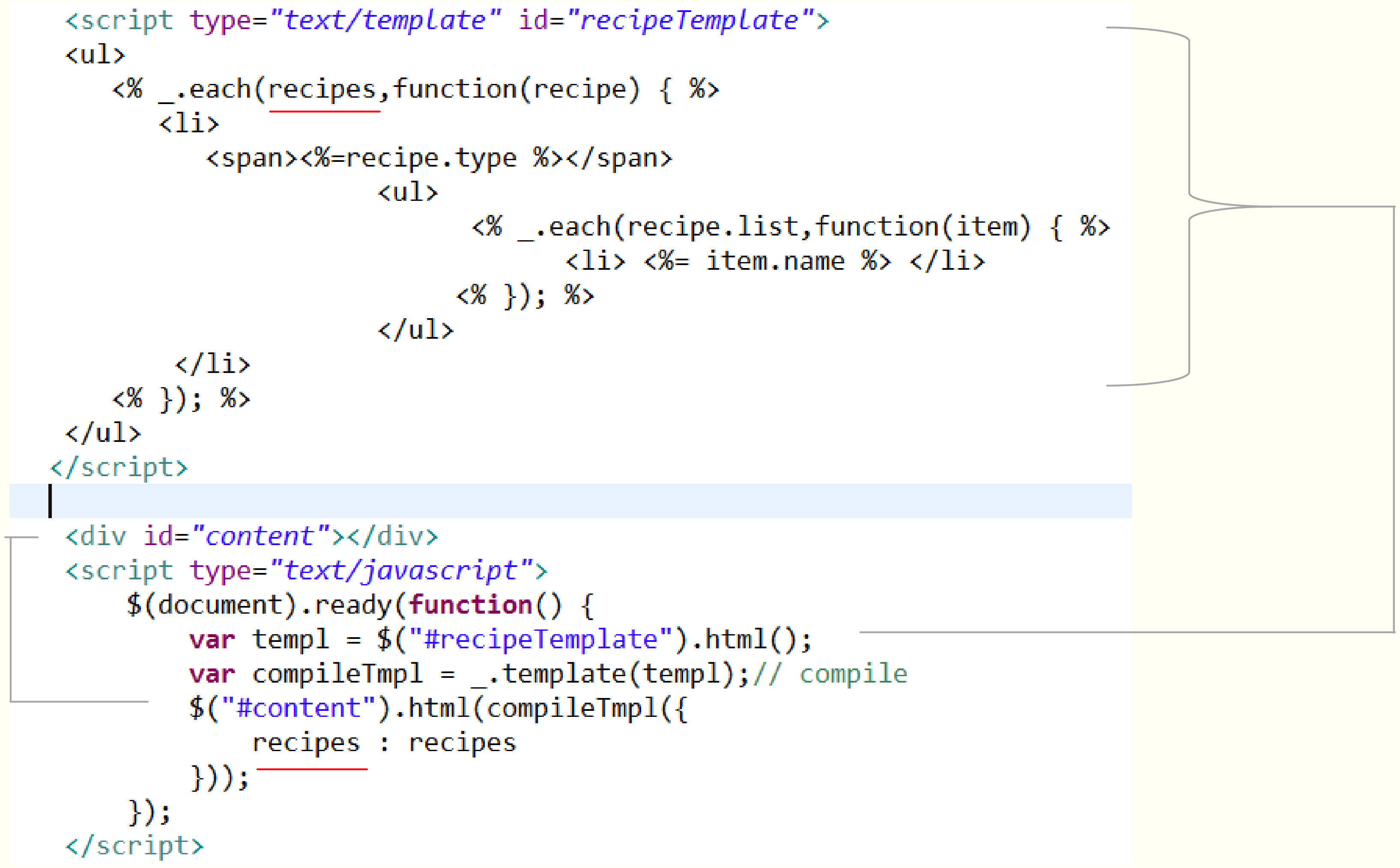


- Breakfast Recipes
  - Masala Dosa
  - Appam
  - Rice Noodles
  - Phulka
  - Aloo Parathas
- Snacks and Sweets
  - Bhelpuri
  - Murukku
  - Gulab Jamun
  - Mysore Pak
- Rice Varieties
- Diabetic Recipes
  - Bittergourd Fry
  - Oats Soup
  - Ragi Idli and Dosa
  - Phulka



```
var recipes = [];
recipes[0] = {
    "type" : "Breakfast Recipes",
    "list" : [
            {
                "id" : 1,
                "name" : "Masala Dosa",
                "pic" : "masaladosa.jpg",
                "ingredients" : []
            },
            {
                "id" : 2,
                "name" : "Appam",
                "pic" : "appam.jpg",
                "ingredients" : []
            },
```

Using loops



```
 <script type="text/template" id="recipeTemplate">
 <ul>
    <% _.each(recipes,function(recipe) { %>
       <li>
          <span><%=recipe.type %></span>
                     <ul>
                           <% _.each(recipe.list,function(item) { %>
                                 <li> <%= item.name %> </li>
                          <% }); %>
                     </ul>
        </li>
    <% }); %>
 </ul>
</script>

 <div id="content"></div>
 <script type="text/javascript">
     $(document).ready(function() {
         var templ = $("#recipeTemplate").html();
         var compileTmpl = _.template(templ);// compile
         $("#content").html(compileTmpl({
             recipes : recipes
         }));
     });
 </script>
```

Handlebars

- **Create a template**
- **In a script tag with invalid type attribute**

```
<script id="post-template" type="text/x-handlebars-template">
<div class='post'>
<h1 class="post-title">{{title}}</h1>
```

- **<!-- If the value should not be escaped use triple curly brackets - ->**

```
<p class="post-content">{{{content}}}</p>
</div>
</script>
```

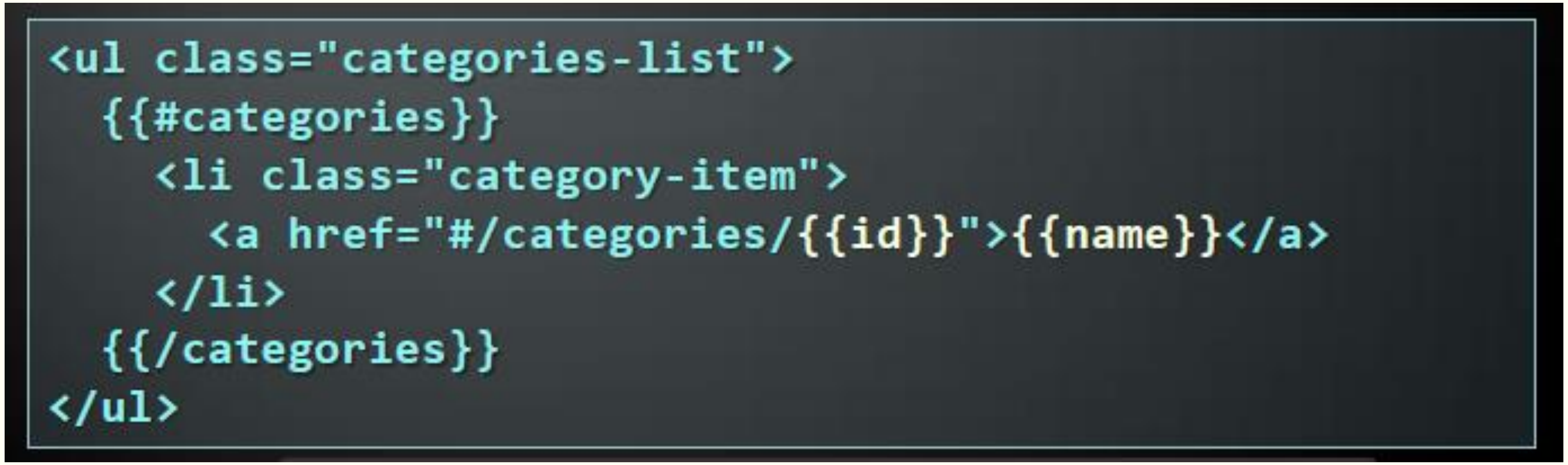
Handlebars

- **Render the template**

```
var post = {title: '…', content: '…'},
htmlTemplate= postTemplateNode.innerHTML,
postTemplate= Handlebars.compile(htmlTemplate),
postNode.innerHTML= postTemplate(post);
```

Handlebars

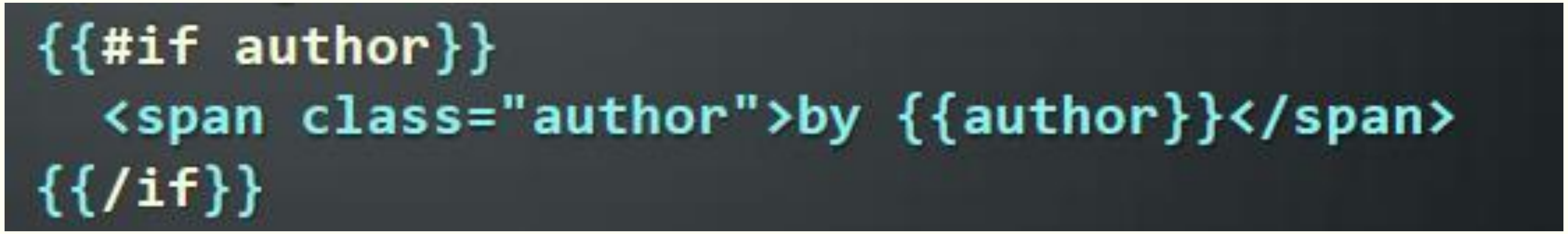
- **Block expressions**
- **Created using {{#collection}}and {{/collection}}**
  - **Everything between will be evaluated for each object in the collection**



```
<ul class="categories-list">
  {{#categories}}
    <li class="category-item">
      <a href="#/categories/{{id}}">{{name}}</a>
    </li>
  {{/categories}}
</ul>
```

Handlebars

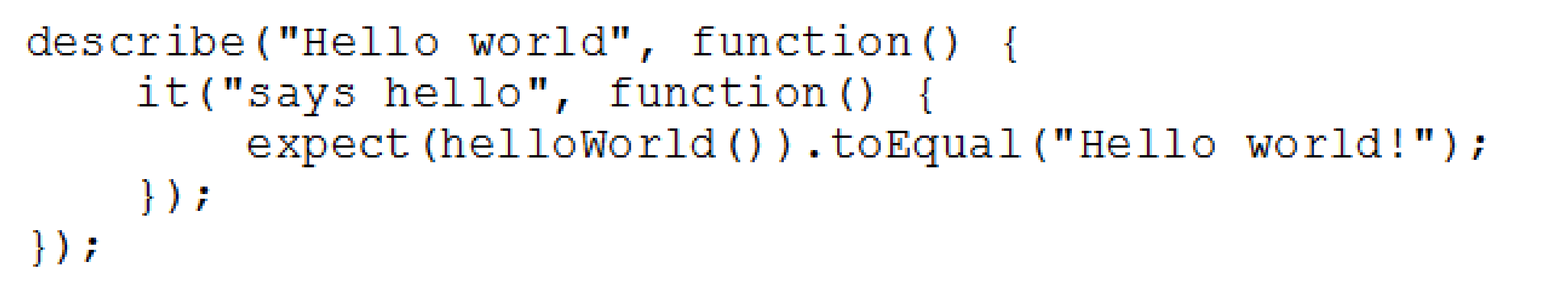
- **Conditional Expressions**



```
{{#if author}}
  <span class="author">by {{author}}</span>
{{/if}}
```

Jasmine Unit testing framework

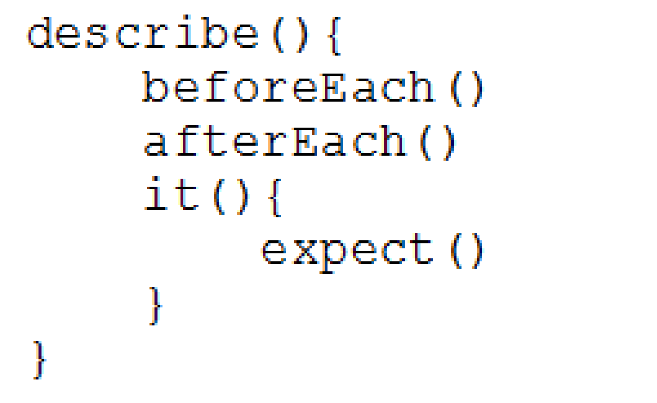
- Jasmine is a framework for unit testing JavaScript code.
- Suites: describe your tests.
- A test suite begins with a call to the global function describe with two parameters: a string and a function.
- The string is the name or title for what is being tested.
- The function is a block of code that implements the suite.



```
describe("Hello world", function() {
    it("says hello", function() {
        expect(helloWorld()).toEqual("Hello world!");
    });
});
```

Structure of test suite

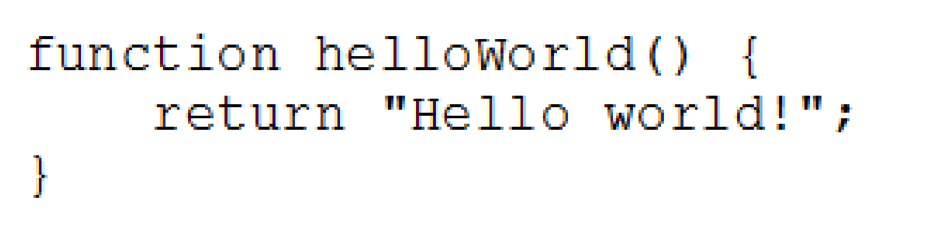
- The beforeEach() and afterEach() are optional utility methods for setup and teardown of your tests.
- Test specifications – it( )
- Test specifications are where tests are defined . A test specification is called it( )and takes two arguments. A string description of the purpose of the tests and a function closure that contains the tests.



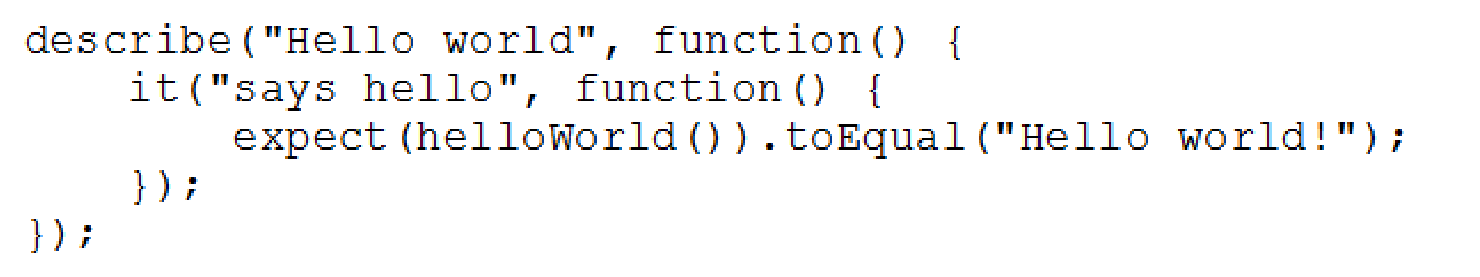
```
describe(){
    beforeEach()
    afterEach()
    it(){
        expect()
    }
}
```

Expectations

- Expectations are built with a function “expect” which takes a value, called the actual. It is chained with a Matcher function, which takes the expected value



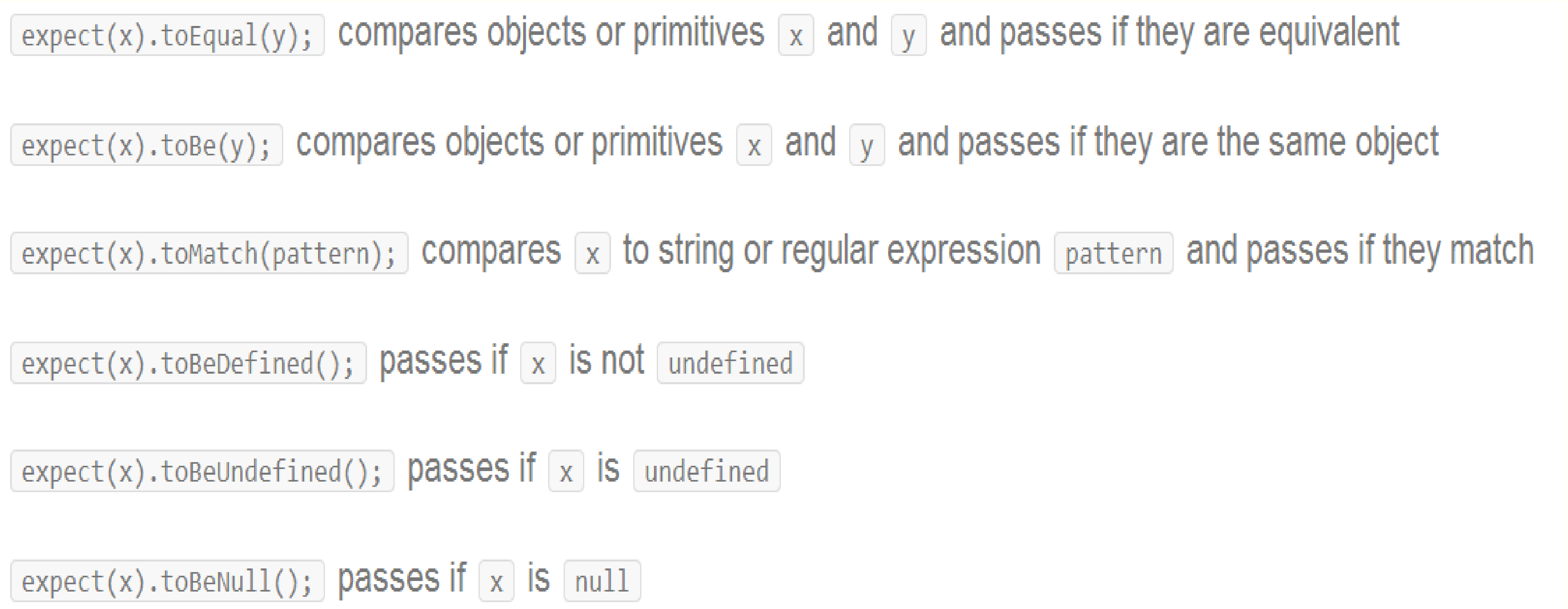
```
function helloWorld() {
    return "Hello world!";
}
```



```
describe("Hello world", function() {
    it("says hello", function() {
        expect(helloWorld()).toEqual("Hello world!");
    });
});
```

More Matchers

- Jasmine has a rich set of matchers.



`expect(x).toEqual(y);` compares objects or primitives `x` and `y` and passes if they are equivalent

`expect(x).toBe(y);` compares objects or primitives `x` and `y` and passes if they are the same object

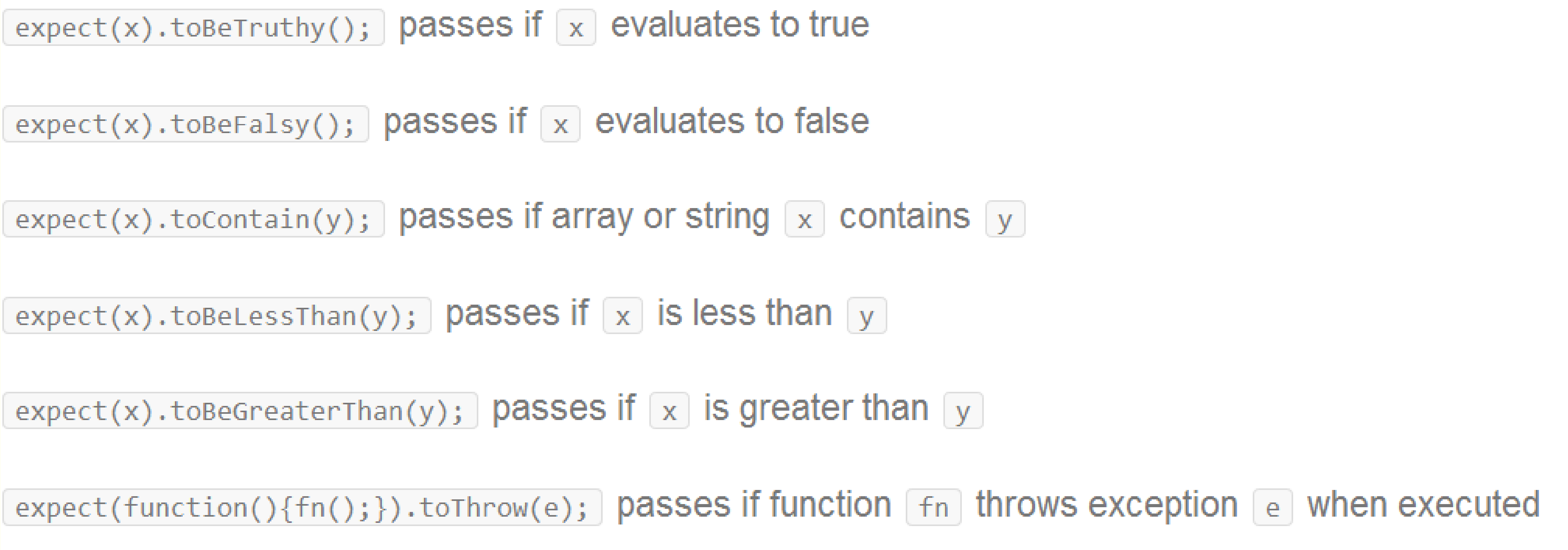
`expect(x).toMatch(pattern);` compares `x` to string or regular expression `pattern` and passes if they match

`expect(x).toBeDefined();` passes if `x` is not `undefined`

`expect(x).toBeUndefined();` passes if `x` is `undefined`

`expect(x).toBeNull();` passes if `x` is `null`

More Matchers…



`expect(x).toBeTruthy();` passes if `x` evaluates to true

`expect(x).toBeFalsy();` passes if `x` evaluates to false

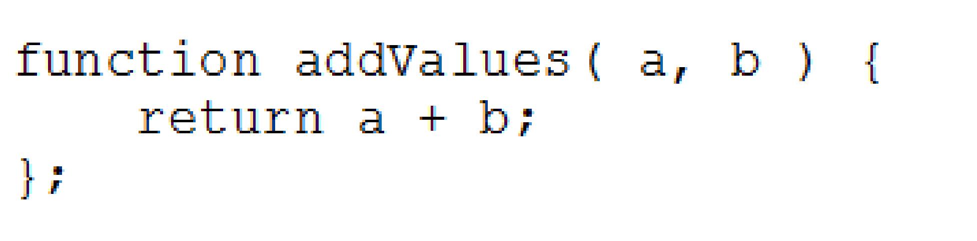
`expect(x).toContain(y);` passes if array or string `x` contains `y`

`expect(x).toBeLessThan(y);` passes if `x` is less than `y`

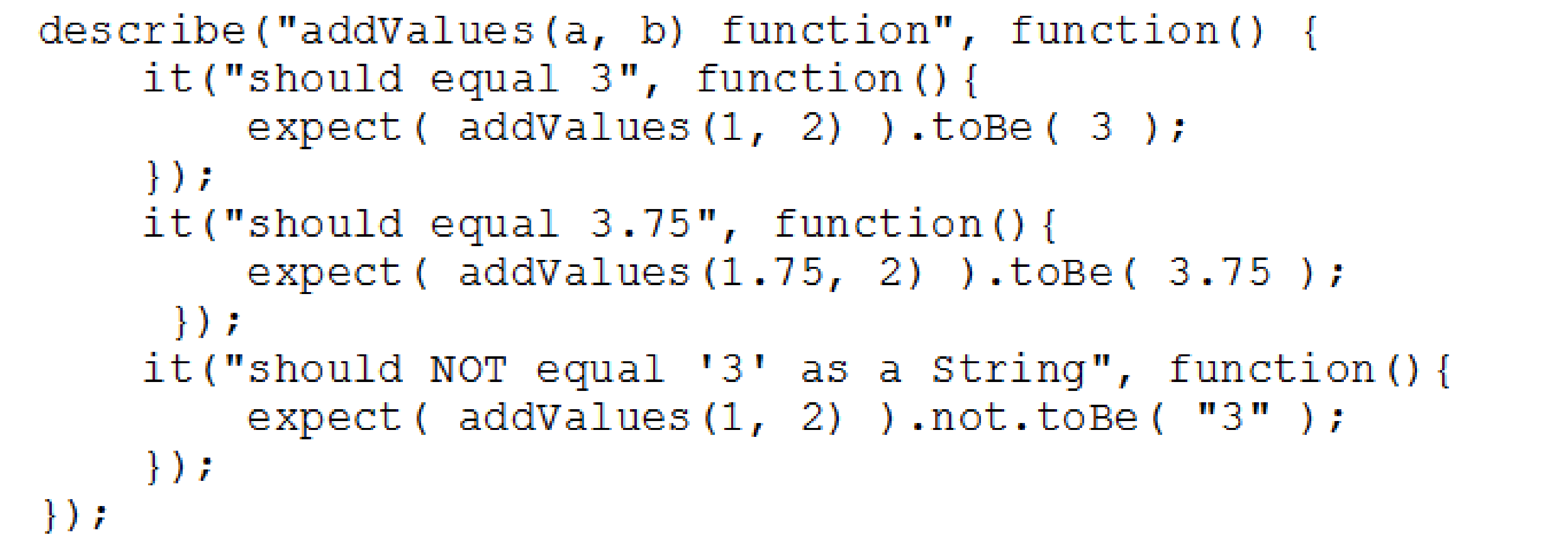
`expect(x).toBeGreaterThan(y);` passes if `x` is greater than `y`

`expect(function(){fn();}).toThrow(e);` passes if function `fn` throws exception `e` when executed

Examples

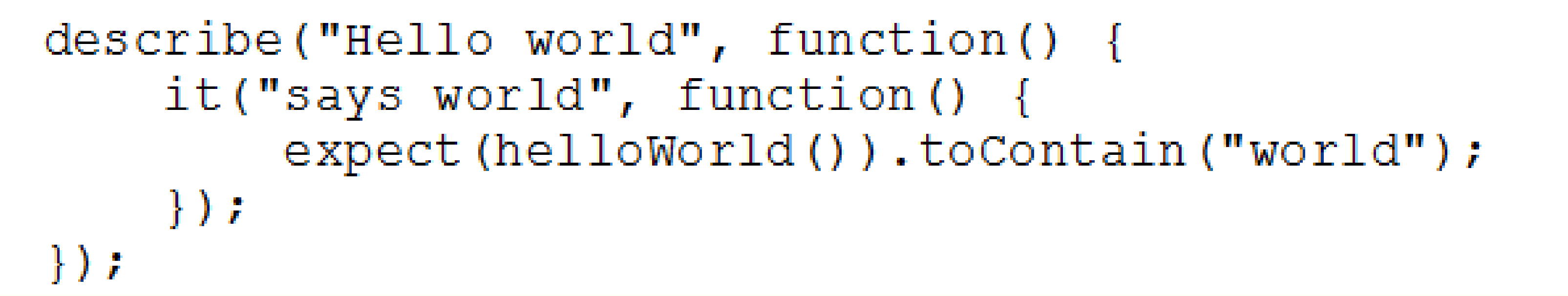


```
function addValues( a, b ) {
    return a + b;
};
```

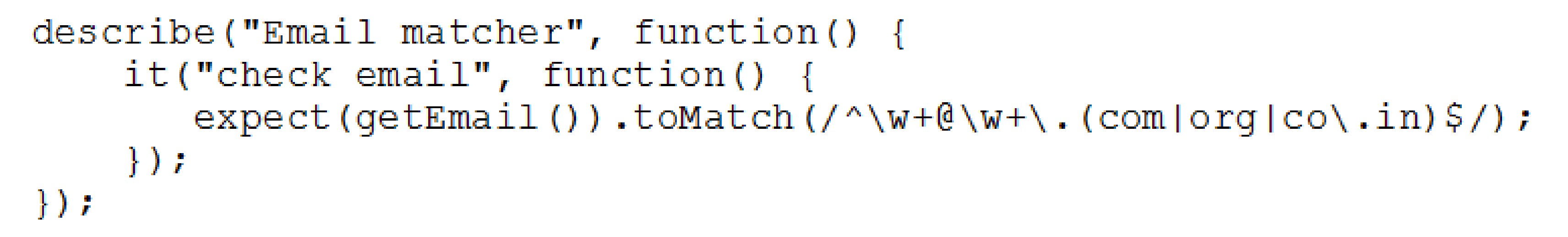


```
describe("addValues(a, b) function", function() {
    it("should equal 3", function(){
        expect( addValues(1, 2) ).toBe( 3 );
    });
    it("should equal 3.75", function(){
        expect( addValues(1.75, 2) ).toBe( 3.75 );
     });
    it("should NOT equal '3' as a String", function(){
        expect( addValues(1, 2) ).not.toBe( "3" );
    });
});
```

Examples

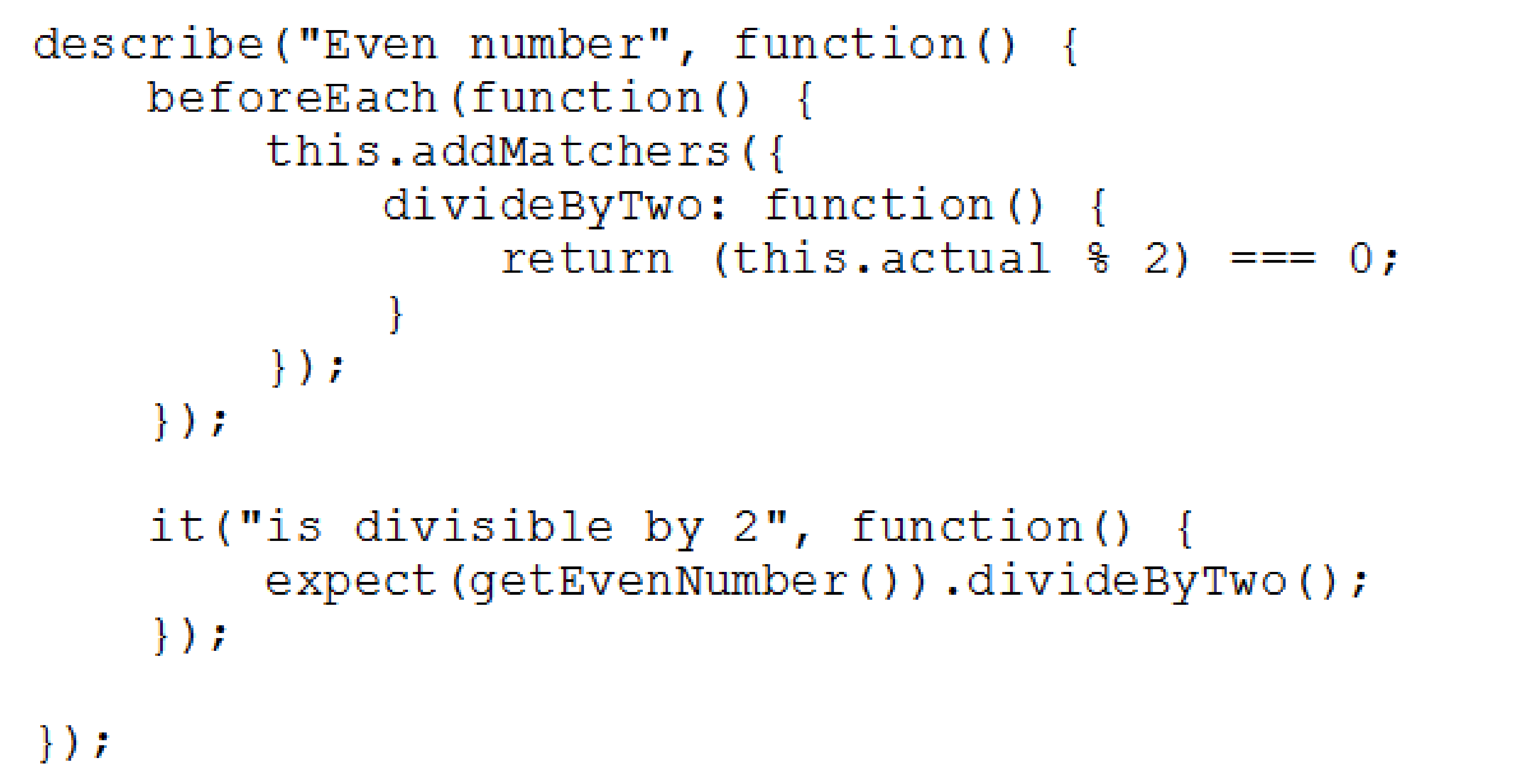


```
describe("Hello world", function() {
    it("says world", function() {
        expect(helloWorld()).toContain("world");
    });
});
```



```
describe("Email matcher", function() {
    it("check email", function() {
       expect(getEmail()).toMatch(/^\w+@\w+\.(com|org|co\.in)$/);
    });
});
```

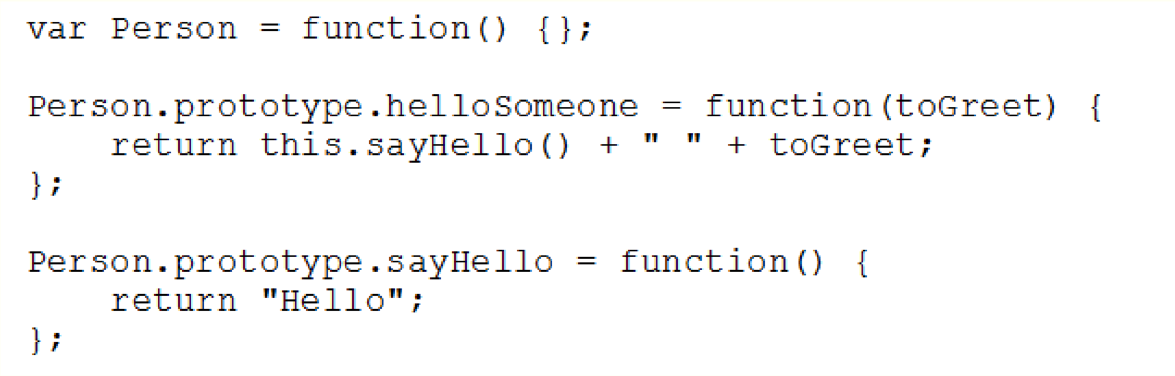
Custom Matchers



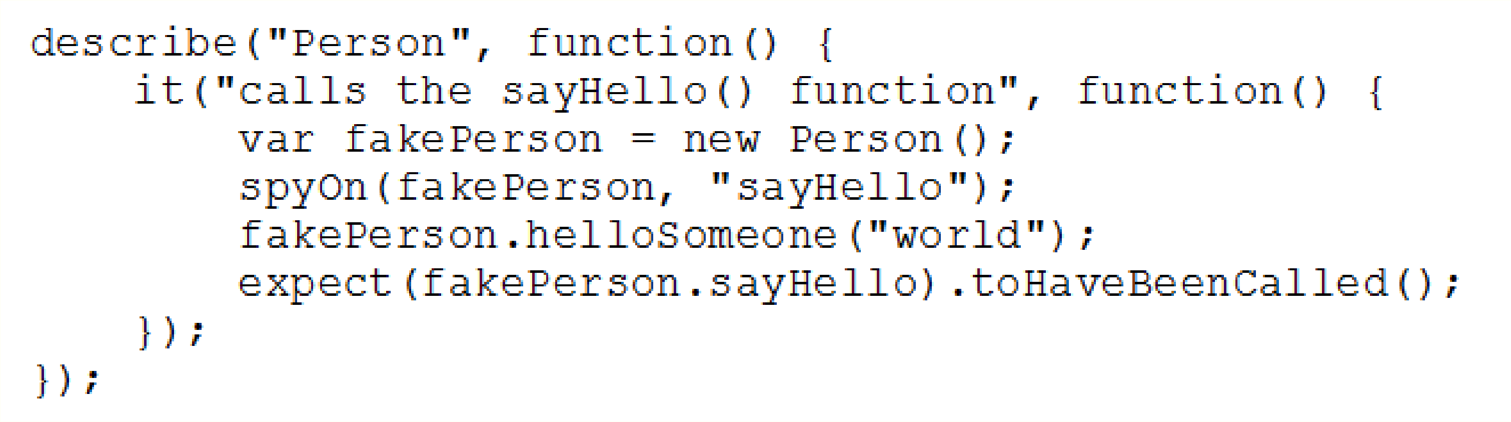
```
describe("Even number", function() {
    beforeEach(function() {
        this.addMatchers({
            divideByTwo: function() {
                return (this.actual % 2) === 0;
            }
        });
    });

    it("is divisible by 2", function() {
        expect(getEvenNumber()).divideByTwo();
    });

});
```

Spies

- In Jasmine, a spy does pretty much what it says: it lets you spy on pieces of your program (and in general, the pieces that aren't just variable checks)



```
var Person = function() {};

Person.prototype.helloSomeone = function(toGreet) {
    return this.sayHello() + " " + toGreet;
};

Person.prototype.sayHello = function() {
    return "Hello";
};
```



```
describe("Person", function() {
    it("calls the sayHello() function", function() {
        var fakePerson = new Person();
        spyOn(fakePerson, "sayHello");
        fakePerson.helloSomeone("world");
        expect(fakePerson.sayHello).toHaveBeenCalled();
    });
});
```

AMD

- Intentionally left blank

Modules

- **Why Web Modules?**
- **The Problem**
  - Web sites are turning into Web apps
  - Code complexity grows as the site gets bigger
  - Assembly gets harder
  - Developer wants discrete JS files/modules
  - Deployment wants optimized code in just one or a few HTTP calls

Modules

- **Solution**
  - Front-end developers need a solution with:
    - Some sort of #include/import/require
    - ability to load nested dependencies
    - ease of use for developer but then backed by an optimization tool that helps deployment

Script Loading APIs

- First thing to sort out is a script loading API. Here are some candidates:
  - Dojo: dojo.require("some.module")
  - LABjs: $LAB.script("some/module.js")
  - CommonJS: require("some/module")

- All of them map to loading some/path/some/module.js.
  - Ideally we could choose the **CommonJS** syntax, since it is likely to get more common over time, and we want to reuse code.
- We also want some sort of syntax that will allow loading plain JavaScript files that exist today.
  - a developer should not have to rewrite all of their JavaScript to get the benefits of script loading.
- However, we need something that works well in the browser.
  - The CommonJS require() is a synchronous call, it is expected to return the module immediately. This does not work well in the browser.

Async vs Sync

- Suppose we have an Employee object and we want a Manager object to derive from the Employee object

```
var Employee = require("types/Employee");
function Manager () {
    this.reports = [];
}
//Error if require call is async
Manager.prototype = new Employee();
```

As the comment indicates above, if require() is async, this code will not work. However, loading scripts synchronously in the browser kills performance

Script Loading: XHR

- Using eval() to evaluate the modules is bad
- While debugging, the line number you get for an error does not map to the original source file.
- XHR also has issues with cross-domain requests. Some browsers now have cross-domain XHR support

- Note: Dojo has used an XHR-based loader with eval() and, while it works, it has been a source of frustration for developers

Script Loading: Web Workers

- Web Workers might be another way to load scripts, but:
  - It does not have strong cross browser support
  - It is a message-passing API, and the scripts likely want to interact with the DOM, so it means just using the worker to fetch the script text, but pass the text back to the main window then use eval/script with text body to execute the script.
  - This has all of the problems as XHR mentioned above.

Script Loading: document.write()

- document.write() can be used to load scripts -- it can load scripts from other domains and it maps to how browsers normally consume scripts, so it allows for easy debugging
  - Problems:
    - Ideally we could know the require() dependencies before we execute the script, and make sure those dependencies are loaded first. But we do not have access to the script before it is executed
    - document.write() does not work after page load
    - scripts loaded via document.write() will block page rendering

Script Loading: head.appendChild(script)

```
var head = document.getElementsByTagName('head')[0],

script = document.createElement('script');

script.src = url;

head.appendChild(script);
```

- This approach has the advantage over document.write in that it will not block page rendering and it works after page load.
- However, it still has the Async vs Sync problem

AMD

- AMD stands for “Asynchronous Module Definition” and is a proposed API for loading modules (i.e. scripts) asynchronously
- Why AMD?
  - Performance (loading scripts asynchronously improves the loading speed of your web page).
  - Modular (which means code that is easier to maintain and re-use).
  - Best practice (helps to reduce global variables, maintain dependencies, and your code will follow guidelines that will in the future become the standards for the next iteration of JavaScript)

AMD

- AMD addresses these issues by:
  - Register the factory function by calling define(), instead of immediately executing it.
  - Pass dependencies as an array of string values, do not grab globals.
  - Only execute the factory function once all the dependencies have been loaded and executed.
  - Pass the dependent modules as arguments to the factory function.

```
//Calling define with a dependency array and a factory function
define(['dep1', 'dep2'], function (dep1, dep2) {
    //Define the module value by returning a value.
    return function () {};
});
```

Named Modules

- Named Modules allows a developer to place the module in a different path to give it a different ID/name.
- The AMD loader will give the module an ID based on how it is referenced by other scripts

```
//Calling define with module ID, dependency array, and factory function
define('myModule', ['dep1', 'dep2'], function (dep1, dep2) {

    //Define the module value by returning a value.
    return function () {};
});
```

Note: You should avoid naming modules yourself, and only place one module in a file while developing.

AMD with many dependencies

```
define(function (require) {
            var dependency1 = require('dependency1'),
            dependency2 = require('dependency2');
            return function () { };
});
```

The AMD loader will parse out the require('') calls by using Function.prototype.toString(),

then internally convert the above define call into this:

```
define(['require', 'dependency1', 'dependency2'], function (require) {
             var dependency1 = require('dependency1'),
             dependency2 = require('dependency2');
           return function ()  { };
});
```

- Note: This allows the loader to load dependency1 and dependency2 asynchronously, execute those dependencies, then execute this function

AMD advantages

- AMD modules require less tooling, there are fewer edge case issues, and better debugging support.
- **Returning a function as the module value**, particularly a constructor function, leads to better API design
- **Dynamic code loading** (done in AMD systems via

  require([ ], function () { })) is a basic requirement
- **Selectively mapping one module** to load from another location makes it easy to provide mock objects for testing.

REQUIREJS

- REQUIREJS IS A JAVASCRIPT FILE AND MODULE LOADER.
- WHY SHOULD I USE REQUIREJS ?
    - Code organization
    - Code saleability
    - Code optimization
    - Multi-developer teams
    - Nested dependencies
    - Ease of use vs. compilation
    - Asynchronous
    - Implementation of the AMD specification.

WHAT DO I NEED TO USE REQUIREJS ?

- RequireJS Library
- Reference it in the HTML head
  - <!-- data-main attribute tells require.js to load
  - scripts/main.js after require.js loads. -->

```
<script data-main="scripts/main" src="scripts/require.js"></script>
```

- WHAT IS REQUIREJS DOING ?
  - It defines two global functions - define() & require()
  - It loads the main.js file (async) after require.js has been loaded

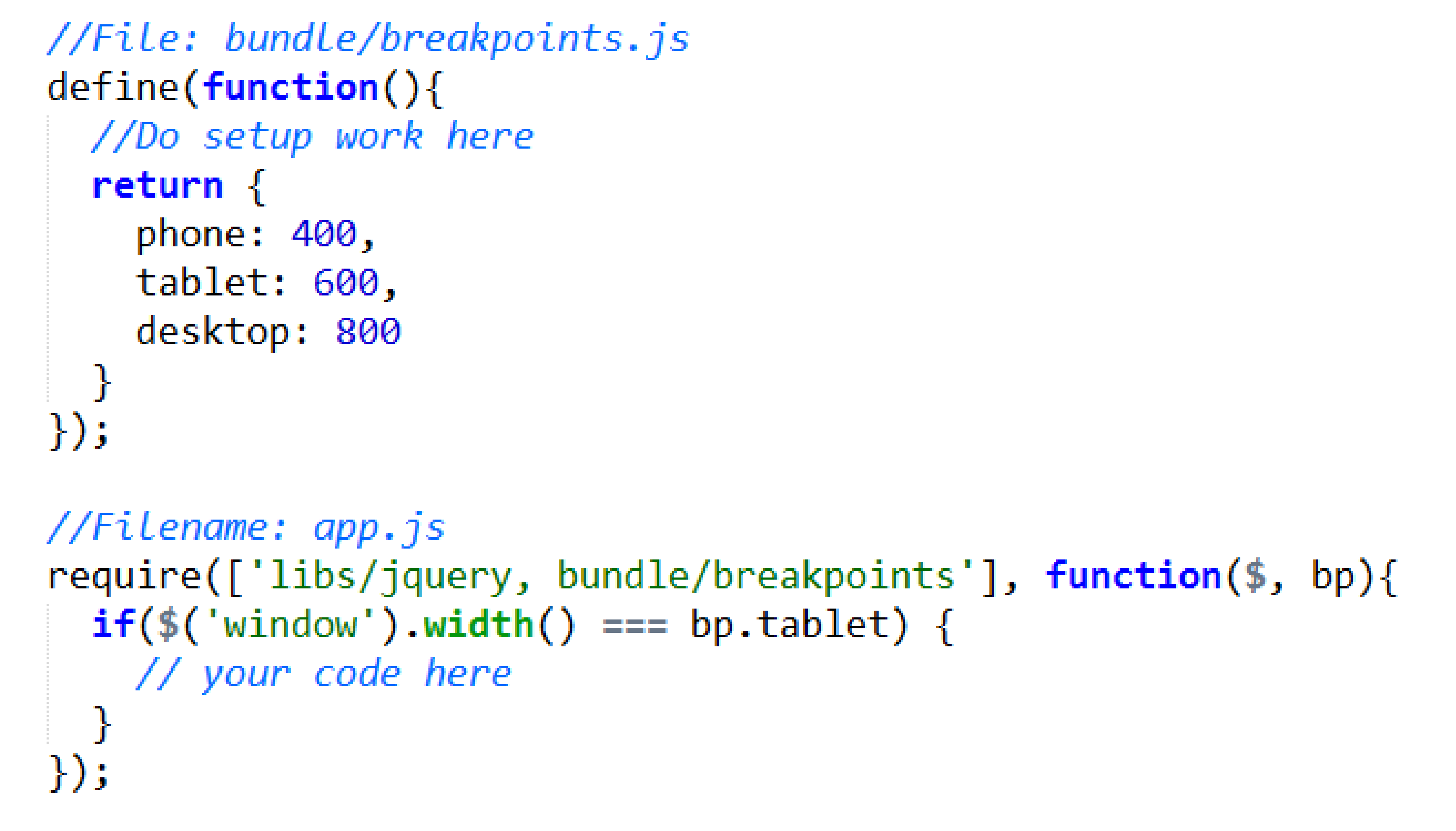
MAIN.JS FILE

- The main.js file is the entry point of the whole app
- It requires the core modules for the website
- It can define aliases for common used libs
- It can be used to compile (minimize)

The define() and require() methods

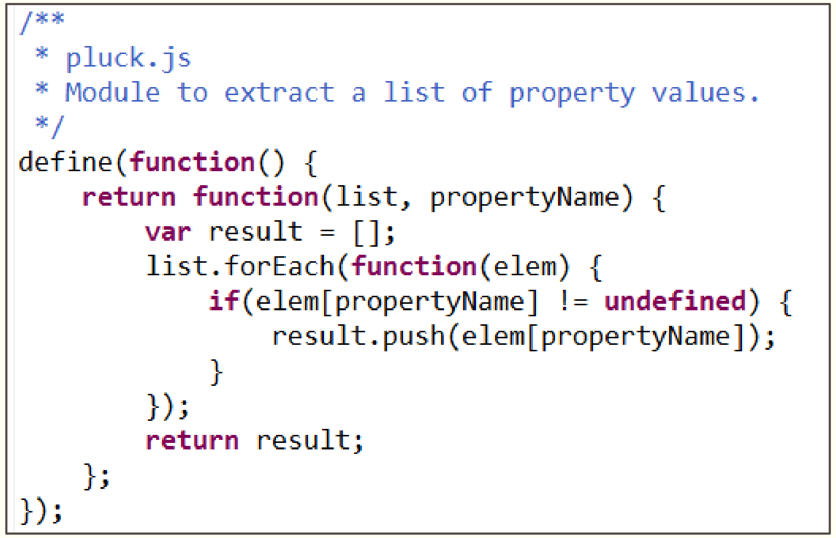
- The two key concepts you need to be aware of here are the idea of a define method for facilitating module definition and a require method for handling dependency loading.
  - define is used to define named or unnamed modules
  - require on the other hand is typically used to load code in a top-level JavaScript file or within a module should you wish to dynamically fetch dependencies

Modules without dependencies

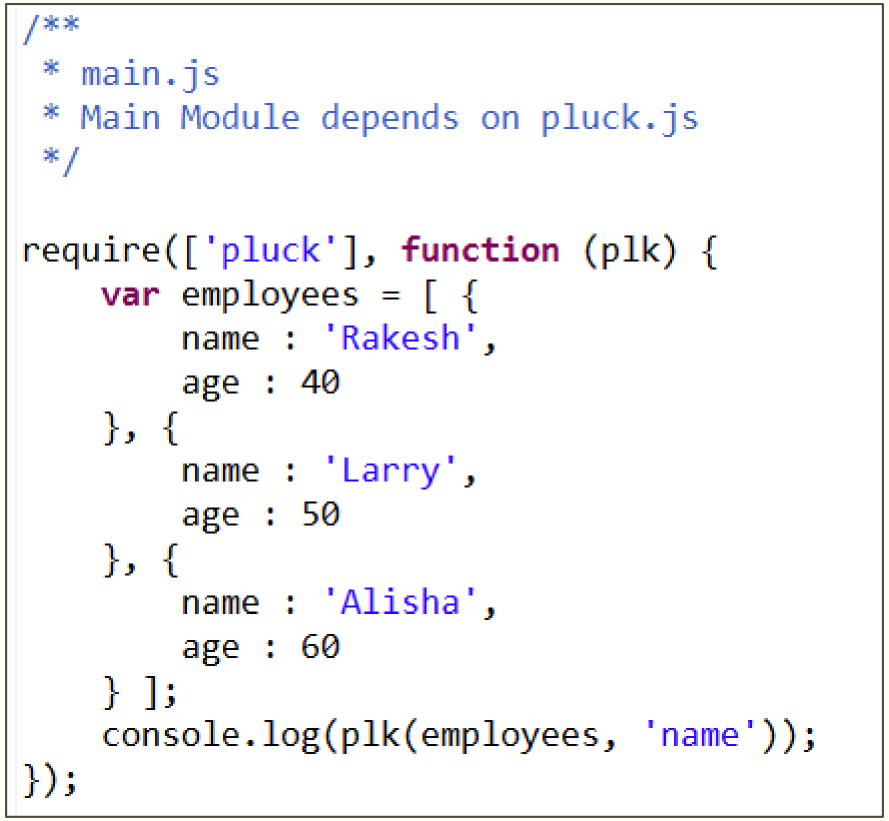


```
//File: bundle/breakpoints.js
define(function(){
  //Do setup work here
  return {
    phone: 400,
    tablet: 600,
    desktop: 800
  }
});

//Filename: app.js
require(['libs/jquery, bundle/breakpoints'], function($, bp){
  if($('window').width() === bp.tablet) {
    // your code here
  }
});
```

Modules example

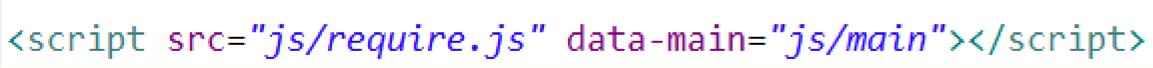


```
/**
 * pluck.js
 * Module to extract a list of property values.
 */
define(function() {
    return function(list, propertyName) {
        var result = [];
        list.forEach(function(elem) {
            if(elem[propertyName] != undefined) {
                result.push(elem[propertyName]);
            }
        });
        return result;
    };
});
```



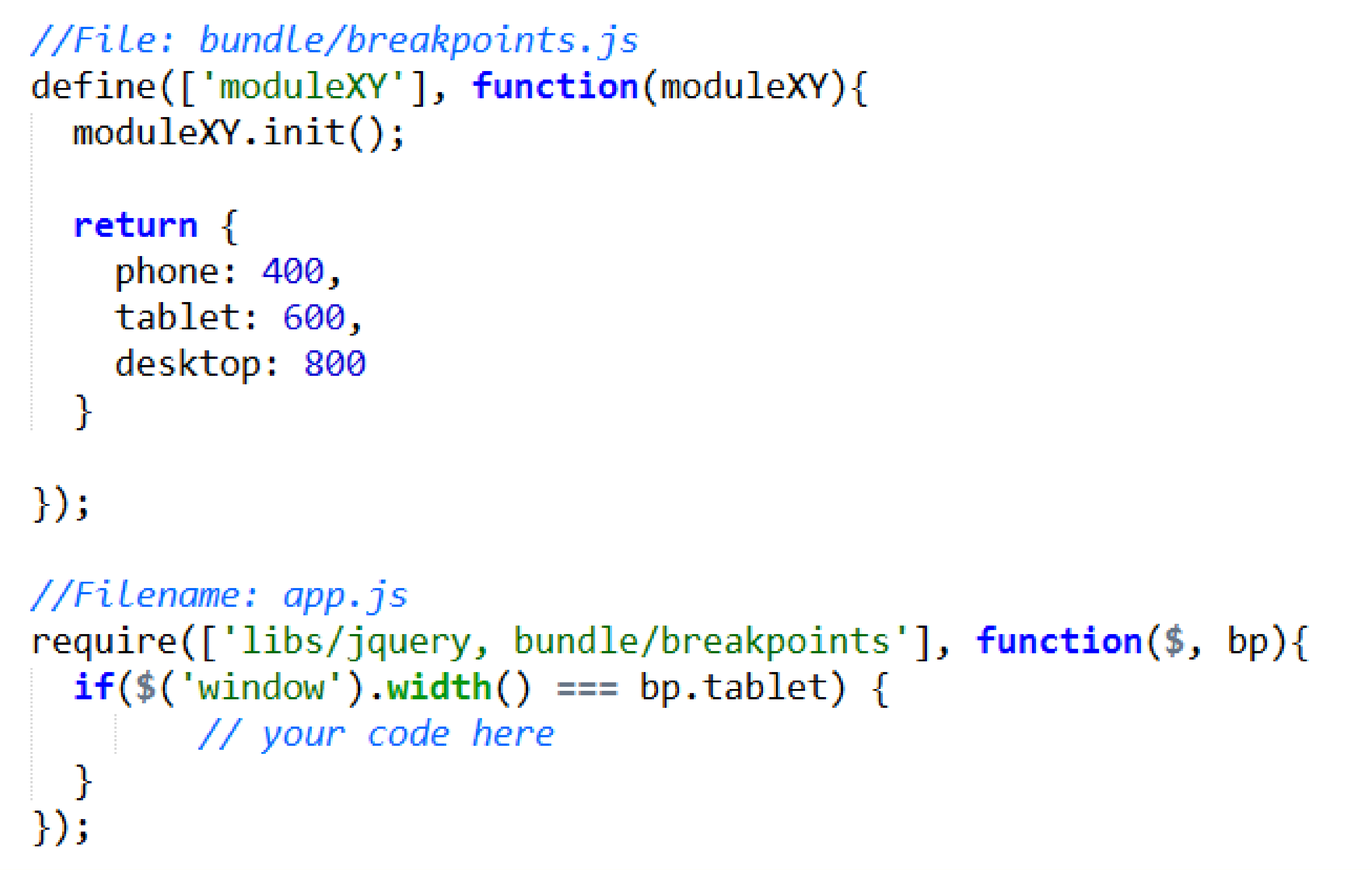
```
/**
 * main.js
 * Main Module depends on pluck.js
 */

require(['pluck'], function (plk) {
    var employees = [ {
        name : 'Rakesh',
        age : 40
    }, {
        name : 'Larry',
        age : 50
    }, {
        name : 'Alisha',
        age : 60
    } ];
    console.log(plk(employees, 'name'));
});
```

HTML file



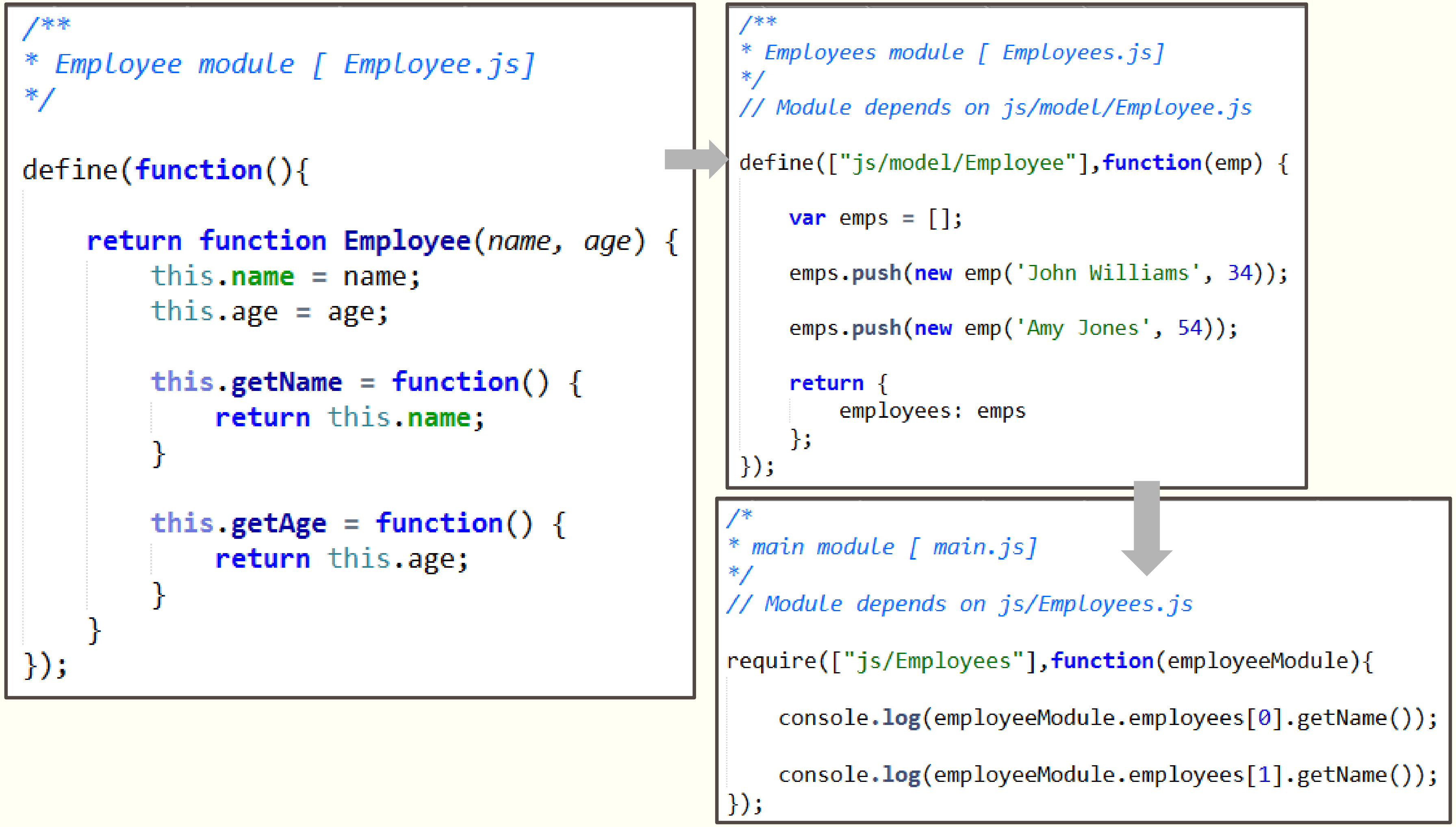
```
<script src="js/require.js" data-main="js/main"></script>
```

Modules with dependencies



```
//File: bundle/breakpoints.js
define(['moduleXY'], function(moduleXY){
  moduleXY.init();

  return {
    phone: 400,
    tablet: 600,
    desktop: 800
  }

});

//Filename: app.js
require(['libs/jquery, bundle/breakpoints'], function($, bp){
  if($('window').width() === bp.tablet) {
        // your code here
  }
});
```

Modules with dependencies example

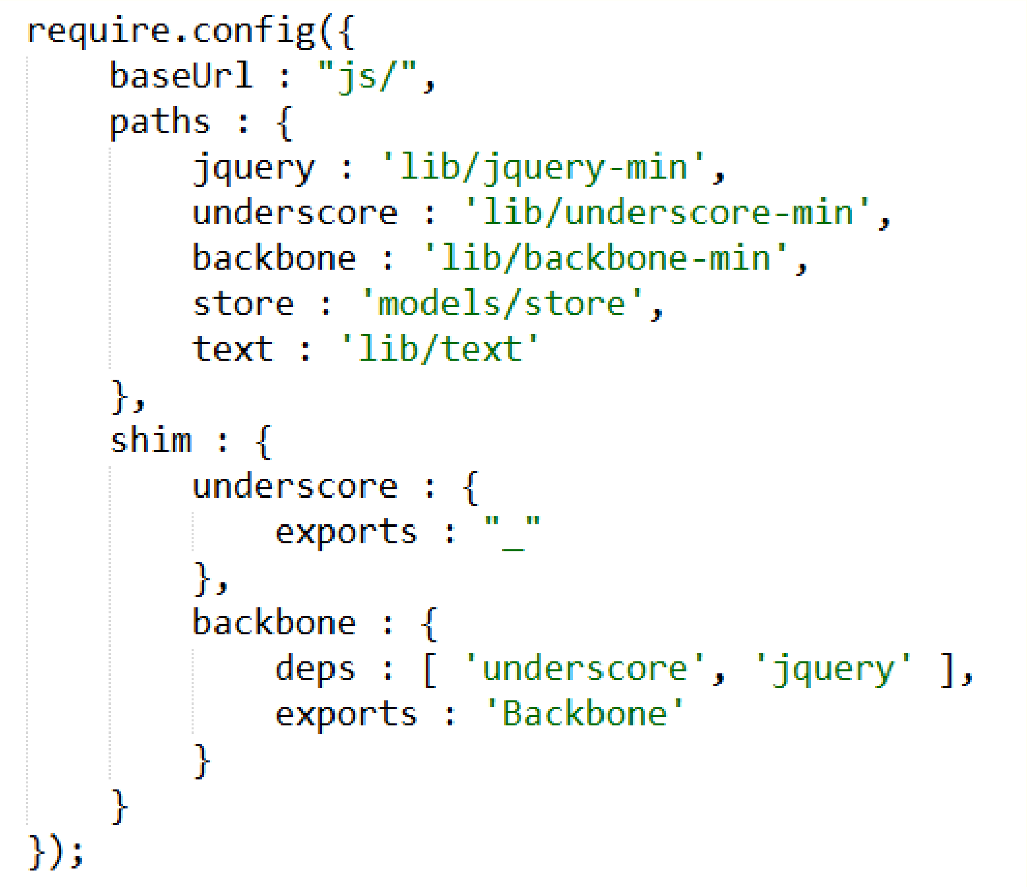


```
/**
* Employee module [ Employee.js]
*/

define(function(){

    return function Employee(name, age) {
        this.name = name;
        this.age = age;

        this.getName = function() {
            return this.name;
        }

        this.getAge = function() {
            return this.age;
        }
    }
});
```

```
/**
* Employees module [ Employees.js]
*/
// Module depends on js/model/Employee.js

define(["js/model/Employee"],function(emp) {

    var emps = [];

    emps.push(new emp('John Williams', 34));

    emps.push(new emp('Amy Jones', 54));

    return {
        employees: emps
    };
});
```

```
/*
* main module [ main.js]
*/
// Module depends on js/Employees.js

require(["js/Employees"],function(employeeModule){

    console.log(employeeModule.employees[0].getName());

    console.log(employeeModule.employees[1].getName());
});
```

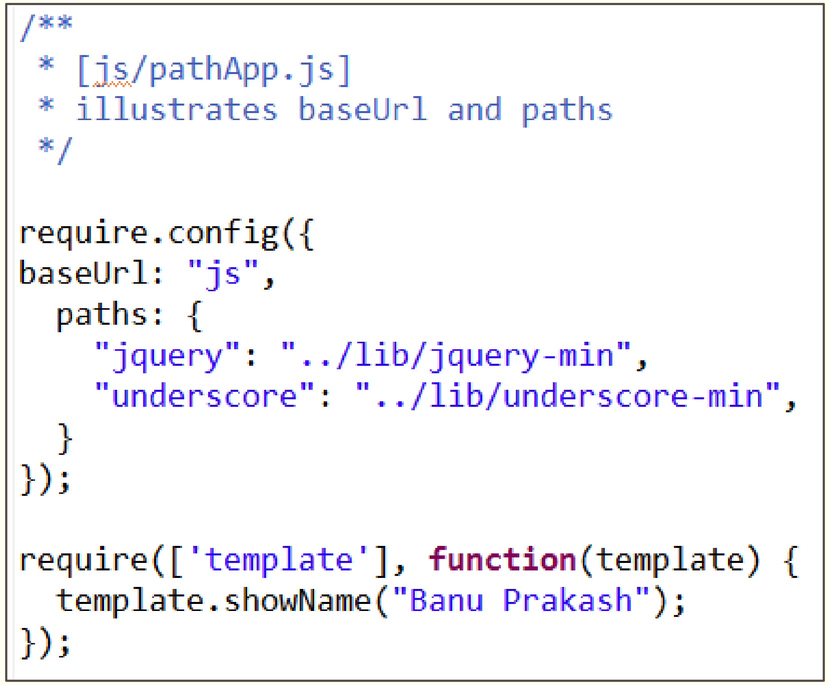
Configuration Options



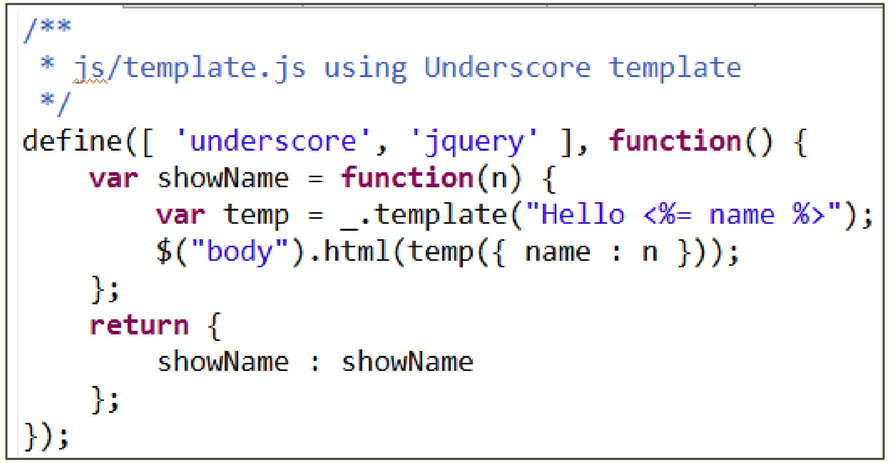
```
require.config({
    baseUrl : "js/",
    paths : {
        jquery : 'lib/jquery-min',
        underscore : 'lib/underscore-min',
        backbone : 'lib/backbone-min',
        store : 'models/store',
        text : 'lib/text'
    },
    shim : {
        underscore : {
            exports : "_"
        },
        backbone : {
            deps : [ 'underscore', 'jquery' ],
            exports : 'Backbone'
        }
    }
});
```

- **baseUrl**
  - the root path to use for all module lookups.
  - If no baseUrl is explicitly set in the configuration, the default value will be the location of the HTML page that loads require.js
- **Paths**
  - path mappings for module names not found directly under baseUrl.
  - The path settings are assumed to be relative to baseUrl, unless the paths setting starts with a "/" or has a URL protocol in it ("like http:").
- **Shim** :
  - gives a hint to RequireJs about the required module dependencies such that RequireJs will then know to load them in the correct order

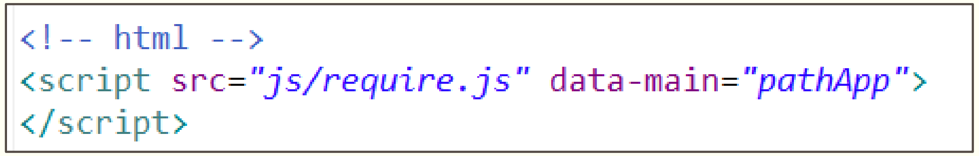
Configuration Options example



```
/**
 * [js/pathApp.js]
 * illustrates baseUrl and paths
 */

require.config({
baseUrl: "js",
  paths: {
    "jquery": "../lib/jquery-min",
    "underscore": "../lib/underscore-min",
  }
});

require(['template'], function(template) {
  template.showName("Banu Prakash");
});
```



```
/**
 * js/template.js using Underscore template
 */
define([ 'underscore', 'jquery' ], function() {
    var showName = function(n) {
        var temp = _.template("Hello <%= name %>");
        $("body").html(temp({ name : n }));
    };
    return {
        showName : showName
    };
});
```

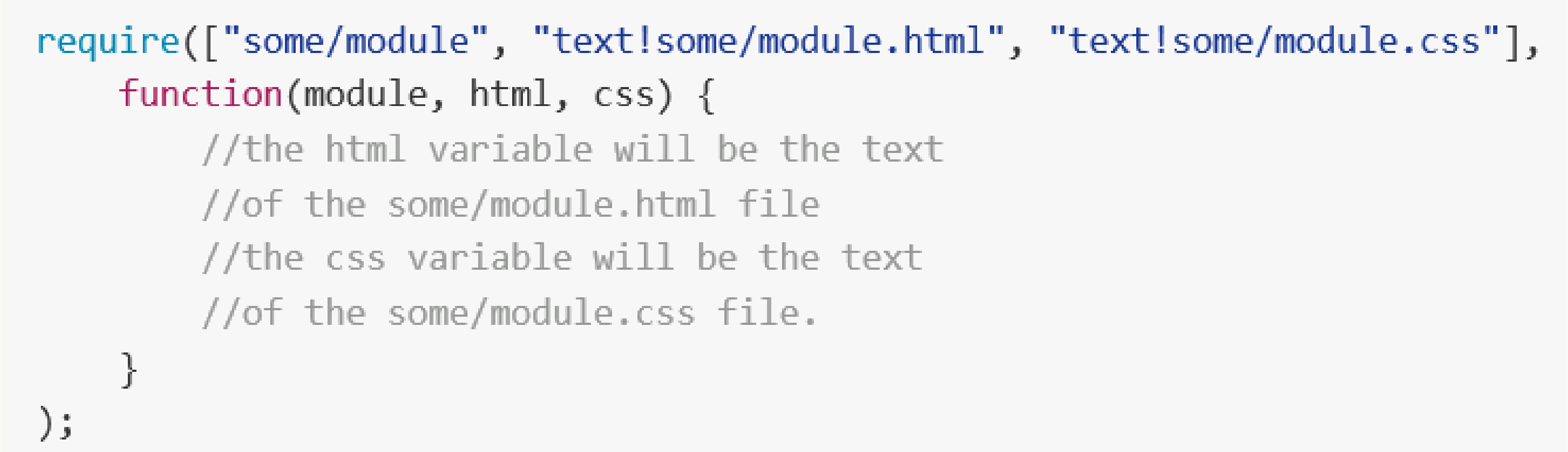


```
<!-- html -->
<script src="js/require.js" data-main="pathApp">
</script>
```

AMD loader plugin for loading text resources

- **text**
  - A RequireJS/AMD loader plugin for loading text resources
  - It is nice to build HTML using regular HTML tags, instead of building up DOM structures in script.
  - However, there is no good way to embed HTML in a JavaScript file.
  - The best that can be done is using a string of HTML, but that can be hard to manage, particularly for multi-line HTML.
  - The text.js AMD loader plugin can help with this issue.
  - It will automatically be loaded if the **text!** prefix is used for a dependency

AMD loader plugin for loading text resources

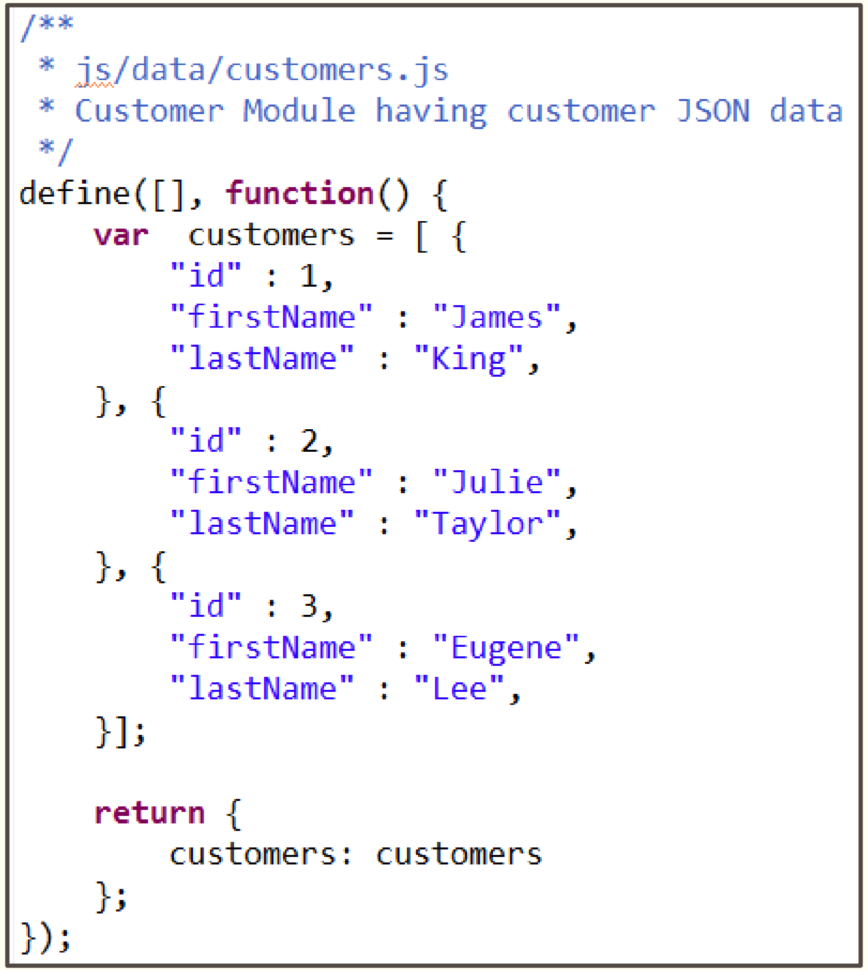


```
require(["some/module", "text!some/module.html", "text!some/module.css"],
    function(module, html, css) {
        //the html variable will be the text
        //of the some/module.html file
        //the css variable will be the text
        //of the some/module.css file.
    }
);
```

- Notice the .html and .css suffixes to specify the extension of the file.
- The "some/module" part of the path will be resolved according to normal module name resolution: it will use the **baseUrl** and **paths** configuration options to map that name to a path

Example using !text

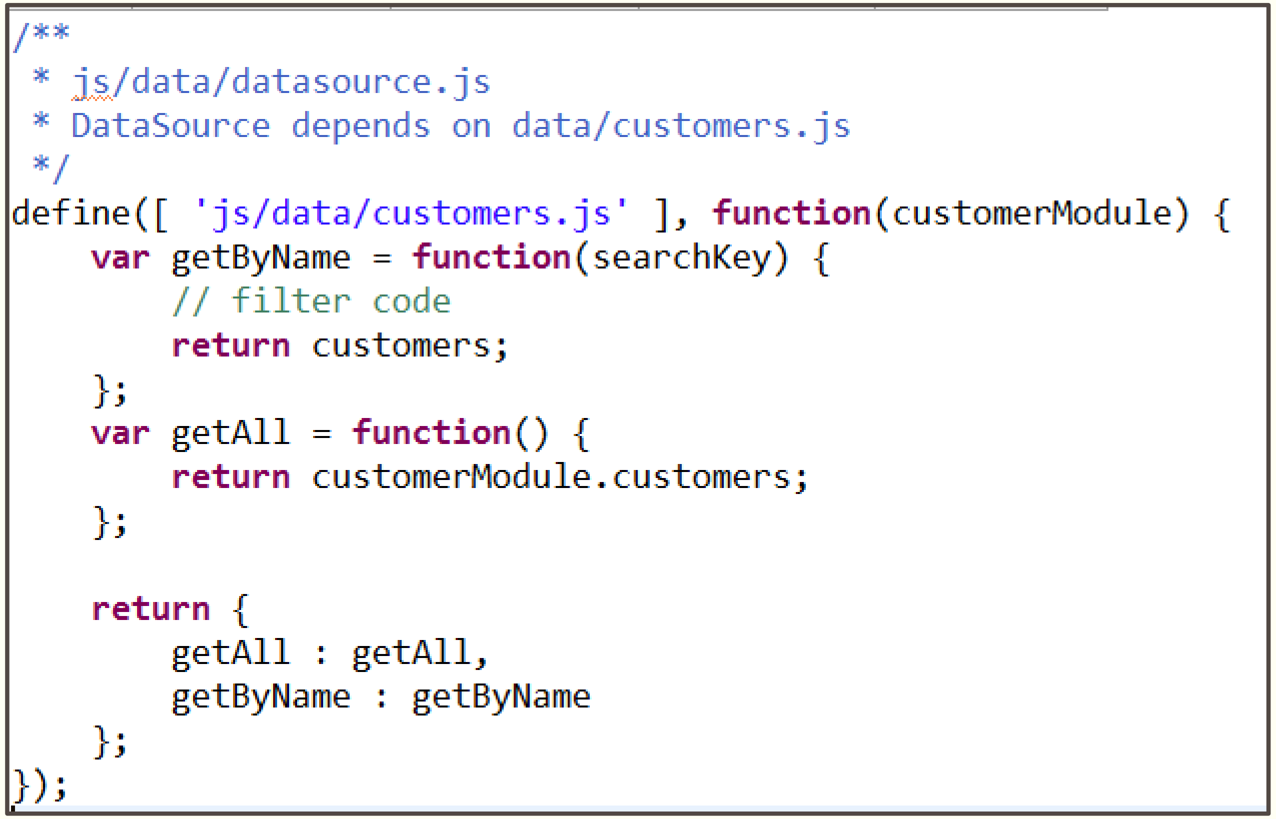
- Customer module having json data



```
/**
 * js/data/customers.js
 * Customer Module having customer JSON data
 */
define([], function() {
    var  customers = [ {
        "id" : 1,
        "firstName" : "James",
        "lastName" : "King",
    }, {
        "id" : 2,
        "firstName" : "Julie",
        "lastName" : "Taylor",
    }, {
        "id" : 3,
        "firstName" : "Eugene",
        "lastName" : "Lee",
    }];

    return {
        customers: customers
    };
});
```

Example using !text

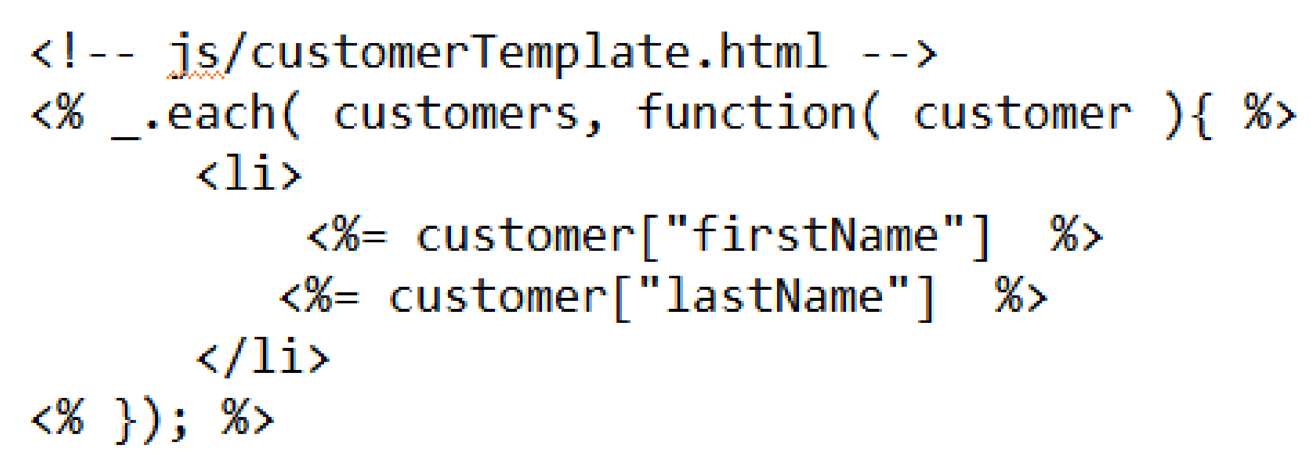
- DataSource module for CRUD operations



```
/**
 * js/data/datasource.js
 * DataSource depends on data/customers.js
 */
define([ 'js/data/customers.js' ], function(customerModule) {
    var getByName = function(searchKey) {
        // filter code
        return customers;
    };
    var getAll = function() {
        return customerModule.customers;
    };

    return {
        getAll : getAll,
        getByName : getByName
    };
});
```

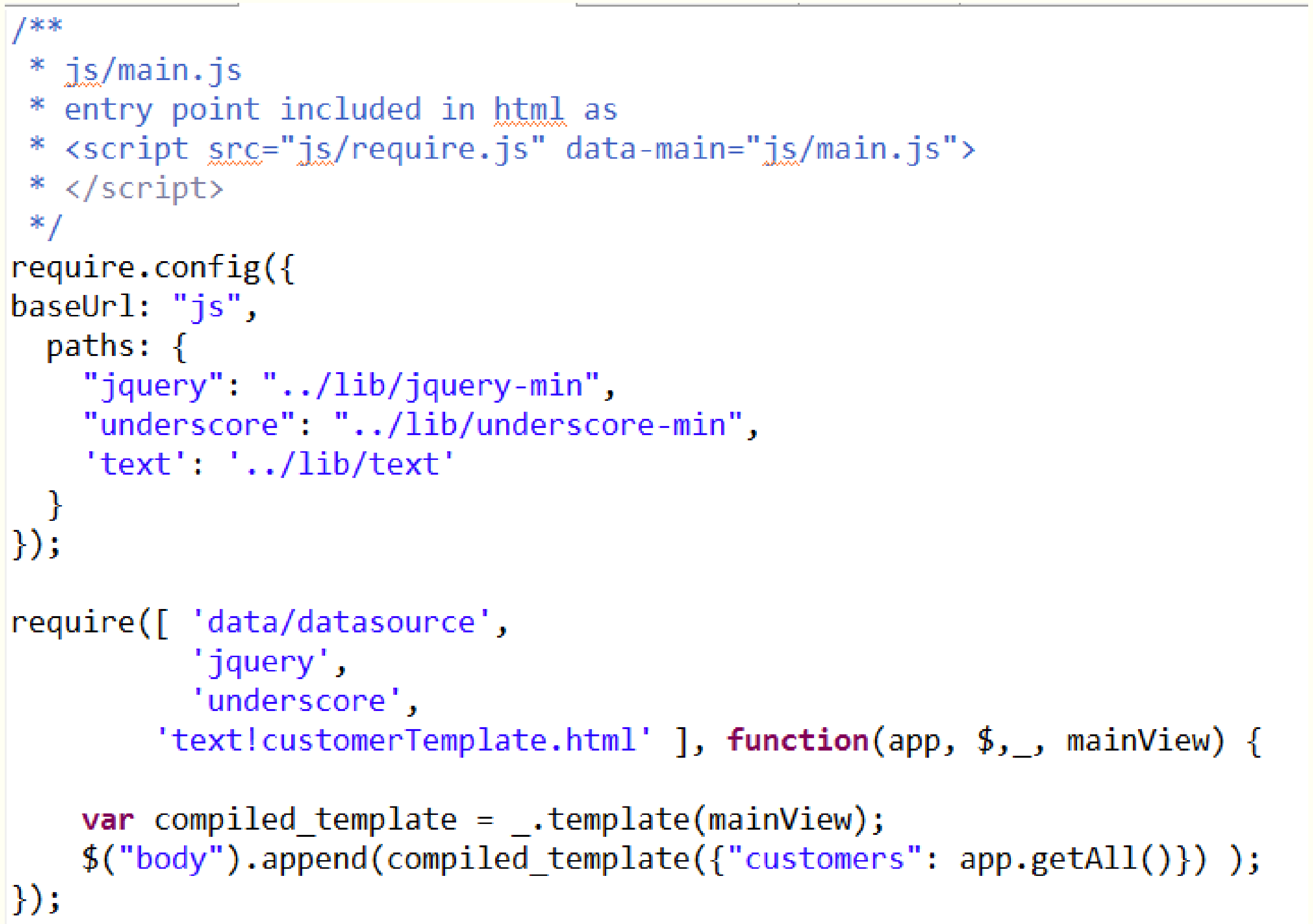
Example using !text

- HTML template which will be loaded using !text plugin



```
<!-- js/customerTemplate.html -->
<% _.each( customers, function( customer ){ %>
      <li>
          <%= customer["firstName"]  %>
         <%= customer["lastName"]  %>
      </li>
<% }); %>
```

Example using !text



```
/**
 * js/main.js
 * entry point included in html as
 * <script src="js/require.js" data-main="js/main.js">
 * </script>
 */
require.config({
baseUrl: "js",
  paths: {
    "jquery": "../lib/jquery-min",
    "underscore": "../lib/underscore-min",
    'text': '../lib/text'
  }
});

require([ 'data/datasource',
          'jquery',
          'underscore',
        'text!customerTemplate.html' ], function(app, $,_, mainView) {

    var compiled_template = _.template(mainView);
    $("body").append(compiled_template({"customers": app.getAll()}) );
});
```

JavaScript coding standards

- Page intentionally left blank

JavaScript Coding Conventions

- Coding conventions and rules for use in JavaScript programming is inspired by the Sun document Code Conventions for the Java Programming Language.
- It is heavily modified of course because JavaScript is not Java.
- The long-term value of software to an organization is in direct proportion to the quality of the codebase.
- Over its lifetime, a program will be handled by many pairs of hands and eyes. If a program is able to clearly communicate its structure and characteristics, it is less likely that it will break when modified in the never-too-distant future.
- Code conventions can help in reducing the brittleness of programs.
- **Neatness counts.**

JavaScript Coding Conventions

- Naming conventions
  - Variable and function names should be full words, using camel case with a lowercase first letter.
  - Constructors for use with new should have a capital first letter.
  - In general:
    - Function names: [example: computeSum() , getEmployeeId(), setName(name), etc]
    - Variable names: [example: employeeName, bookTitle, etc]
    - Class names/Constructors: [example: LineItem, Order, Person, etc]
    - Constant names: [example: MAX_VALUE, MIN_VALUE, etc]

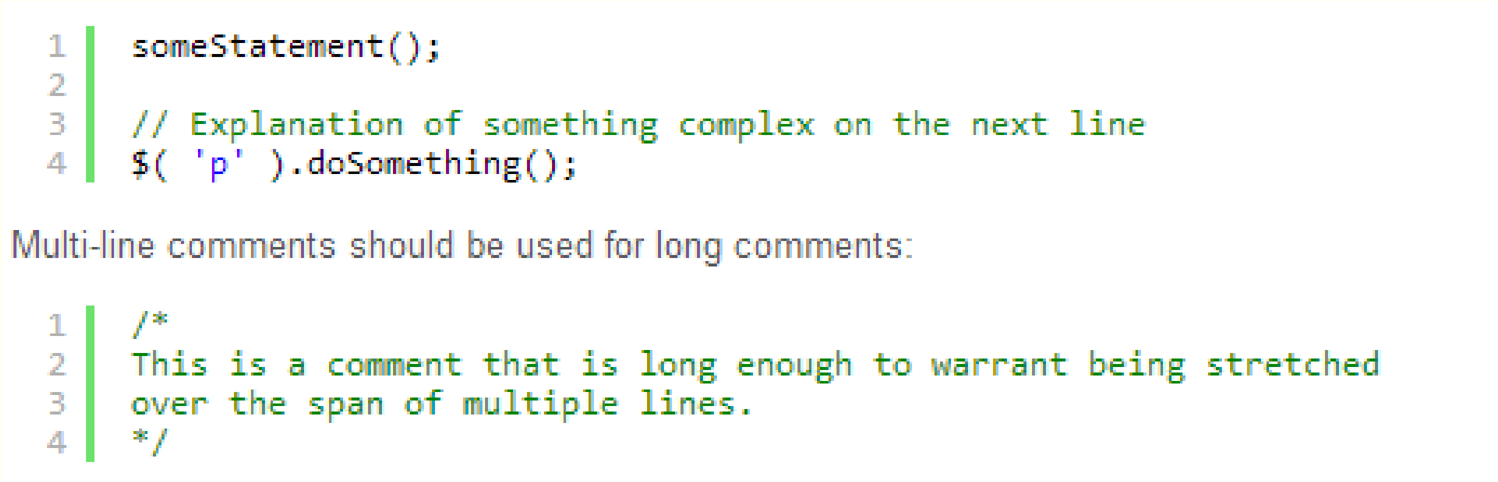
JavaScript Coding Conventions

- **Variable Declarations**
  - All variables should be declared before used. JavaScript does not require this, but doing so makes the program easier to read and makes it easier to detect undeclared variables that may become implied globals.
  - Implied global variables should never be used. Use of global variables should be minimized.
  - The var statement should be the first statement in the function body.
  - It is preferred that each variable be given its own line and comment. They should be listed in alphabetical order if possible.

```
var currentEntry, // currently selected table entry
level,      // indentation level
size;       // size of table
```

JavaScript Coding Conventions

- Comments:
  - Comments come before the code to which they refer and should always be preceded by a blank line.
  - Capitalize the first letter of the comment and include a period at the end when writing full sentences.



```
someStatement();

// Explanation of something complex on the next line
$( 'p' ).doSomething();
```

Multi-line comments should be used for long comments:

```
/*
This is a comment that is long enough to warrant being stretched
over the span of multiple lines.
*/
```

JavaScript Coding Conventions

- **Function Declarations**
- There should be no space between the name of a function and the ((left parenthesis) of its parameter list.
- There should be one space between the) (right parenthesis) and the { (left curly brace) that begins the statement body.
- The body itself is indented four spaces. The } (right curly brace) is aligned with the line containing the beginning of the declaration of the function.

```
function computeSum(first, second) {

    return first + second;

}
```

JavaScript Coding Conventions

- **Spaces Around Operators**
    - Always put spaces around operators (= + / *), and after commas:
    - Examples:

```
var x = y + z;
var values = ["Volvo", "Saab", "Fiat"];
```

- **Code Indentation**
    - Always use 4 spaces for indentation of code blocks:

```
function toCelsius(fahrenheit) {
    return (5/9) * (fahrenheit-32);
}
```

JavaScript Coding Conventions

- Statement Rules
  - General rules for simple statements:
    - Always end simple statement with a semicolon.
  - General rules for complex (compound) statements:
    - Put the opening bracket at the end of the first line.
    - Use one space before the opening bracket.
    - Put the closing bracket on a new line, without leading spaces.

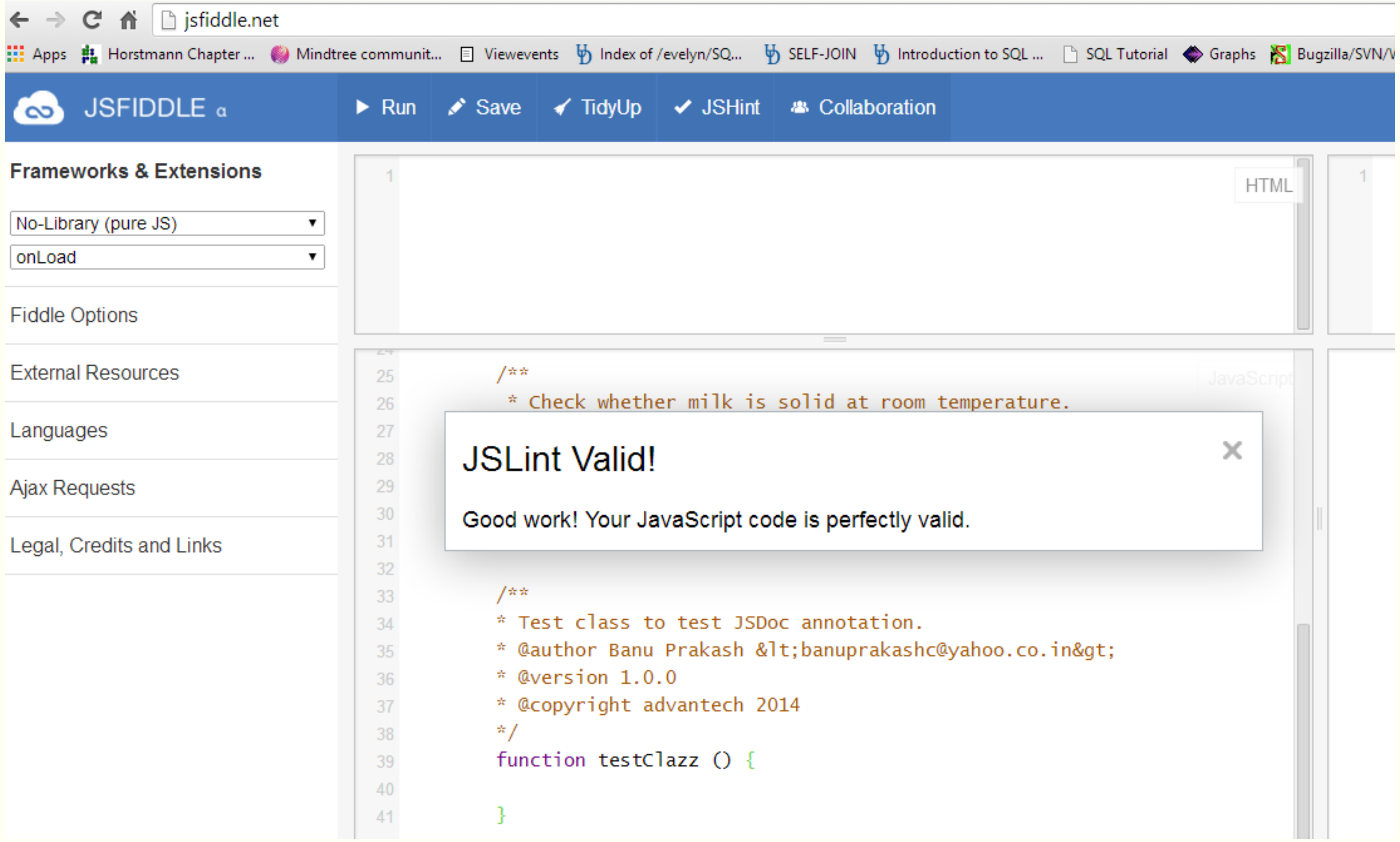
JavaScript Coding Conventions

- Line Length < 80
  - For readability, avoid lines longer than 80 characters.
  - If a JavaScript statement does not fit on one line, the best place to break it, is after an operator or a comma.
  - Example
    - document.getElementById("demo").innerHTML =

      "Hello Dolly.";

Coding Standards

- Check coding standards using JSFIDDLE



No-Library (pure JS)

onLoad

```
        /**
         * Check whether milk is solid at room temperature.
```

JSLint Valid!

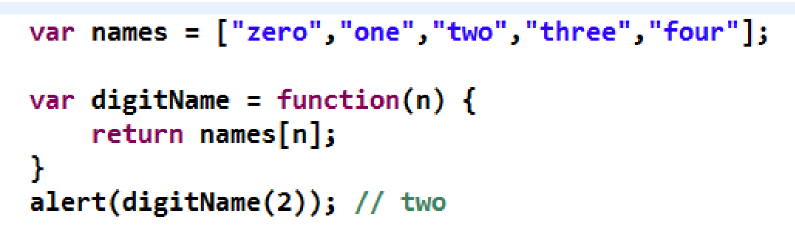
Good work! Your JavaScript code is perfectly valid.

```
        /**
        * Test class to test JSDoc annotation.
        * @author Banu Prakash &lt;banuprakashc@yahoo.co.in&gt;
        * @version 1.0.0
        * @copyright advantech 2014
        */
        function testClazz () {

        }
```

JavaScript best practices

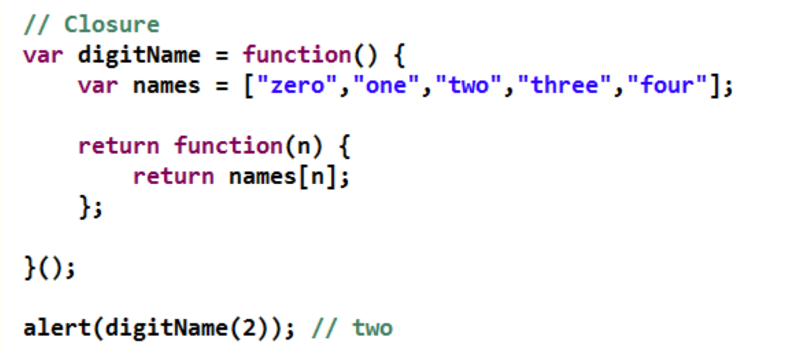
- Avoid Global Variables
  - Minimize the use of global variables.
  - This includes all data types, objects, and functions.
  - Global variables and functions can be overwritten by other scripts.
  - Use local variables instead, and learn how to use closures.

Global variables



```
var names = ["zero","one","two","three","four"];

var digitName = function(n) {
    return names[n];
}
alert(digitName(2)); // two
```

Closure



```
// Closure
var digitName = function() {
    var names = ["zero","one","two","three","four"];

    return function(n) {
        return names[n];
    };

}();

alert(digitName(2)); // two
```

JavaScript best practices

- **Initialize Variables**
  - It is a good coding practice to initialize variables when you declare them.
  - This will:
    - Give cleaner code
    - Provide a single place to initialize variables
    - Avoid undefined values
- Example:
    - var firstName = "",
    - lastName = "",
    - price = 0,
    - discount = 0,
    - myArray = [],
    - myObject = {};

JavaScript best practices

- **Never Declare Number, String, or Boolean Objects**
  - Always treat numbers, strings, or booleans as primitive values. Not as objects.
  - Declaring these types as objects, slows down execution speed, and produces nasty side effects:
  - Example
      - var x = “Mindtree";
      - var y = new String("Mindtree");
      - (x === y) // is false because x is a string and y is an object.
  - Or even worse:
    - var x = new String("Mindtree");
    - var y = new String("Mindtree");
    - (x == y) // is false because you cannot compare objects.

JavaScript best practices

- **Don't Use new Object()**
- Use { } instead of new Object()
- Use "" instead of new String()
- Use 0 instead of new Number()
- Use false instead of new Boolean()
- Use [] instead of new Array()
- Use /()/ instead of new RegExp()
- Use function (){ } instead of new function()
- **Example**
  - var x1 = {}; // new object
  - var x2 = ""; // new primitive string
  - var x3 = 0; // new primitive number
  - var x4 = false; // new primitive boolean
  - var x5 = []; // new array object
  - var x6 = /()/; // new regexp object
  - var x7 = function(){}; // new function object

JavaScript best practices

- **Use === Comparison**
  - The == comparison operator always converts (to matching types) before comparison.
  - The === operator forces comparison of values and type:

```
0 == "";        // true
1 == "1";       // true
1 == true;      // true

0 === "";       // false
1 === "1";      // false
1 === true;     // false
```

JavaScript best practices

- **Use Parameter Defaults**
  - If a function is called with a missing argument, the value of the missing argument is set to **undefined**.
  - Undefined values can break your code. It is a good habit to assign default values to arguments.

```
function setData(opt) {
    opt = options || [ ];
}
```

JavaScript best practices

- **End Your Switches with Defaults**
  - Always end your switch statements with a default. Even if you think there is no need for it.

```
switch (new Date().getDay()) {
    case 0:
        day = "Sunday";
        break;
    case 1:
        day = "Monday";
        break;
    case 2:
        day = "Tuesday";
        break;
    case 3:
        day = "Wednesday";
        break;
    case 4:
        day = "Thursday";
        break;
    case 5:
        day = "Friday";
        break;
    case 6:
        day = "Saturday";
        break;
    default:
        day = "Unknown";
}
```

Java best practices

- **Caching selectors for long periods of time**

```
var initMenu = function()
{
    var menu = document.getElementById('menu').   // Cache the selector
    menu.show();
    menu.addClass('enabled');
    menu.click(handleClick);
};
```

Debugging JavaScript

- As the complexity of JavaScript applications increase, developers need powerful debugging tools to help quickly discover the cause of an issue and fix it efficiently.
- Firefox has Firebug Dev tools and chrome comes with the Chrome DevTools which include a number of useful tools to help make debugging JavaScript less painful

What is Firebug?

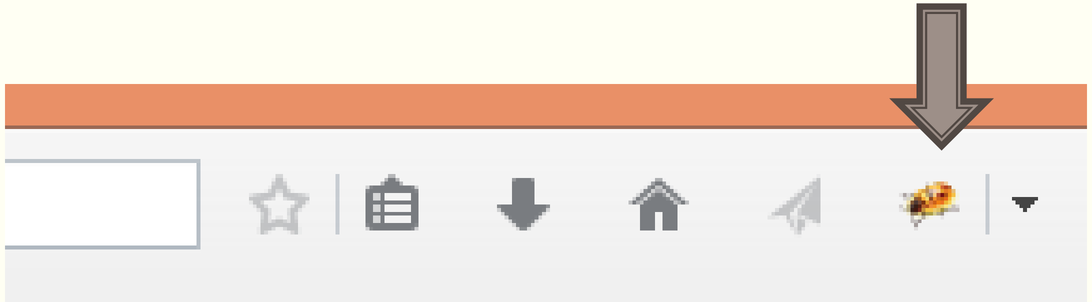
- Firebug is an extension for the Mozilla Firefox browser that allows you to debug and inspect HTML, CSS, the Document Object Model (DOM) and JavaScript

Why do Web Developers use Firebug?

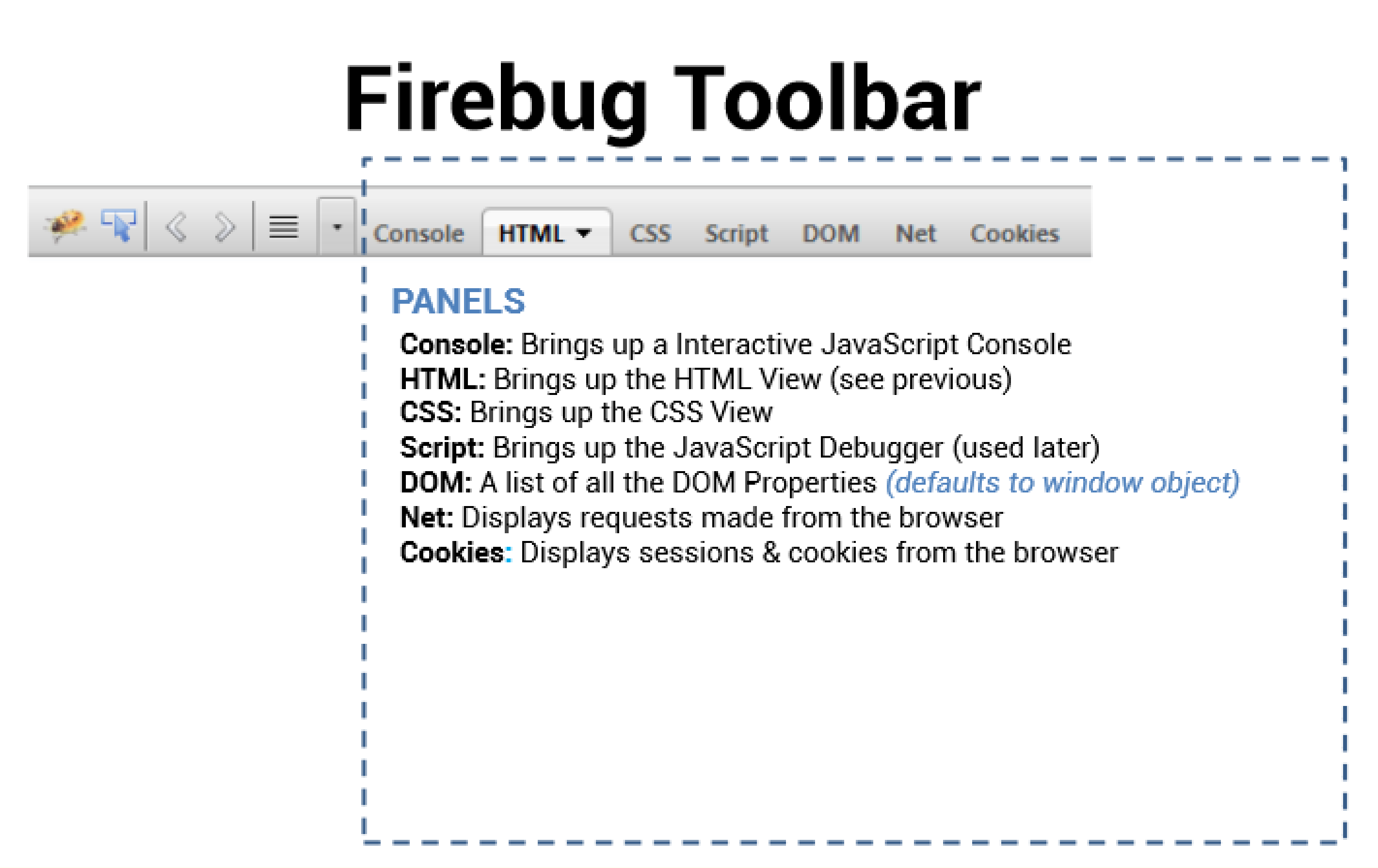
- Inspect the behaviour of HTML/CSS, and modify style & layout with true WYSIWYG
- Debug JavaScript
- Detect performance of website
- Track Cookies & Sessions

Launch Firebug

- With the Mozilla Firefox browser open...
  - Press F12 on the keyboard
  - OR
  - Press the Firebug button on the toolbar



Firebug Toolbar



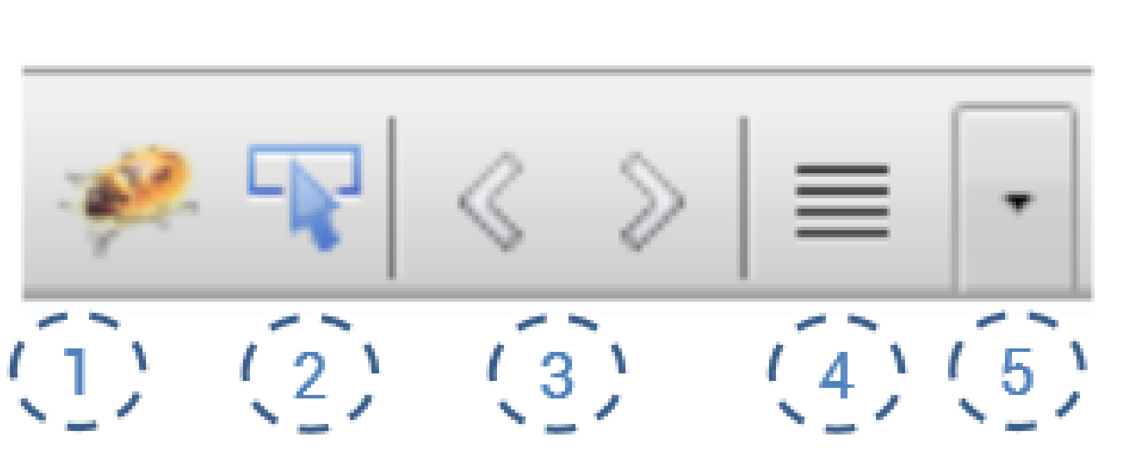
Firebug Toolbar

PANELS

Console: Brings up a Interactive JavaScript Console
HTML: Brings up the HTML View (see previous)
CSS: Brings up the CSS View
Script: Brings up the JavaScript Debugger (used later)
DOM: A list of all the DOM Properties *(defaults to window object)*
Net: Displays requests made from the browser
Cookies: Displays sessions & cookies from the browser

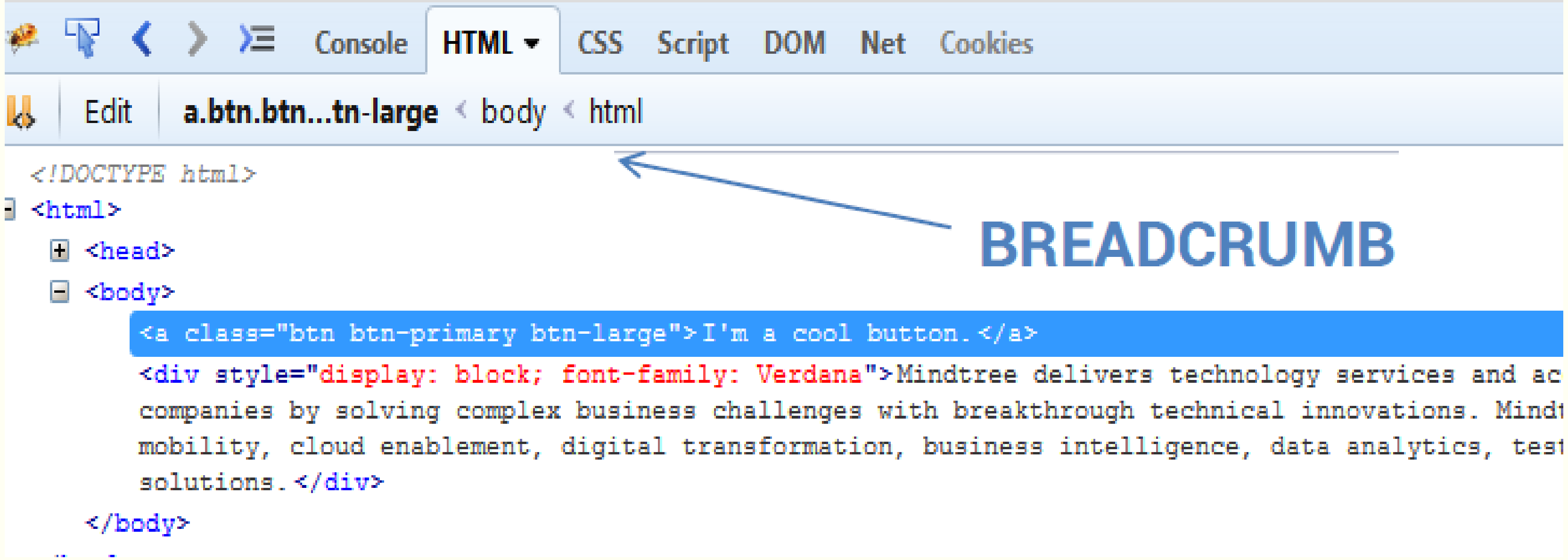
Firebug Toolbar – Firebug Button

- 1. The Firebug Button
- 2. Inspect Element
- 3. Back/Forward – Switches between Panels
- 4. Quick Console – Interactive JavaScript console
- 5. Show or H



Finding the Attributes & DOM properties of the Element

- The HTML Panel is displayed with the element selected

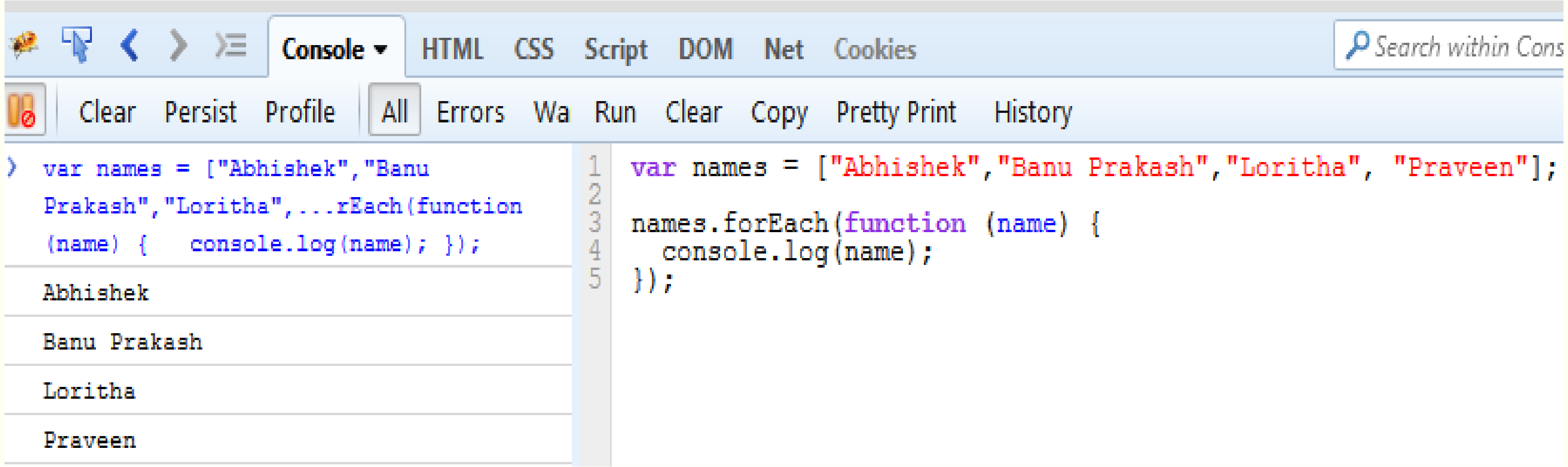


BREADCRUMB

```
<!DOCTYPE html>
<html>
  <head>
  <body>
      <a class="btn btn-primary btn-large">I'm a cool button.</a>
      <div style="display: block; font-family: Verdana">Mindtree delivers technology services and ac
      companies by solving complex business challenges with breakthrough technical innovations. Mindt
      mobility, cloud enablement, digital transformation, business intelligence, data analytics, test
      solutions.</div>
  </body>
```

JavaScript Console

- Using the Interactive JavaScript Console
  - Execute the following JavaScript line:
  - (To execute JavaScript code, type the line and click on Run)
  - To write good JavaScript code, you have to test it frequently. We can write functions, test it, and ensure validity through the console.

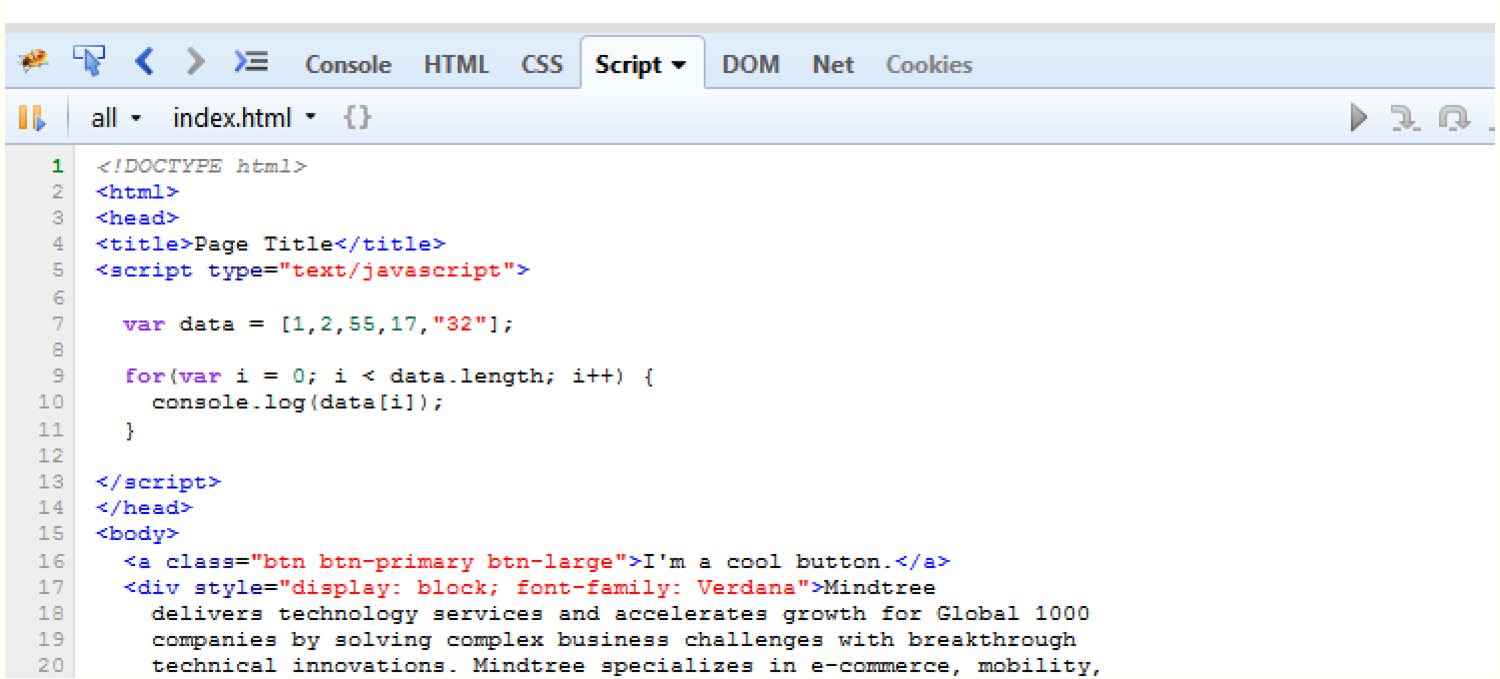


```
› var names = ["Abhishek","Banu
  Prakash","Loritha",...rEach(function
  (name) {   console.log(name); });
  Abhishek
  Banu Prakash
  Loritha
  Praveen
```

```
var names = ["Abhishek","Banu Prakash","Loritha", "Praveen"];

names.forEach(function (name) {
  console.log(name);
});
```

JavaScript Code & Detecting Errors with Firebug

- **Firebug - SCRIPT PANEL**

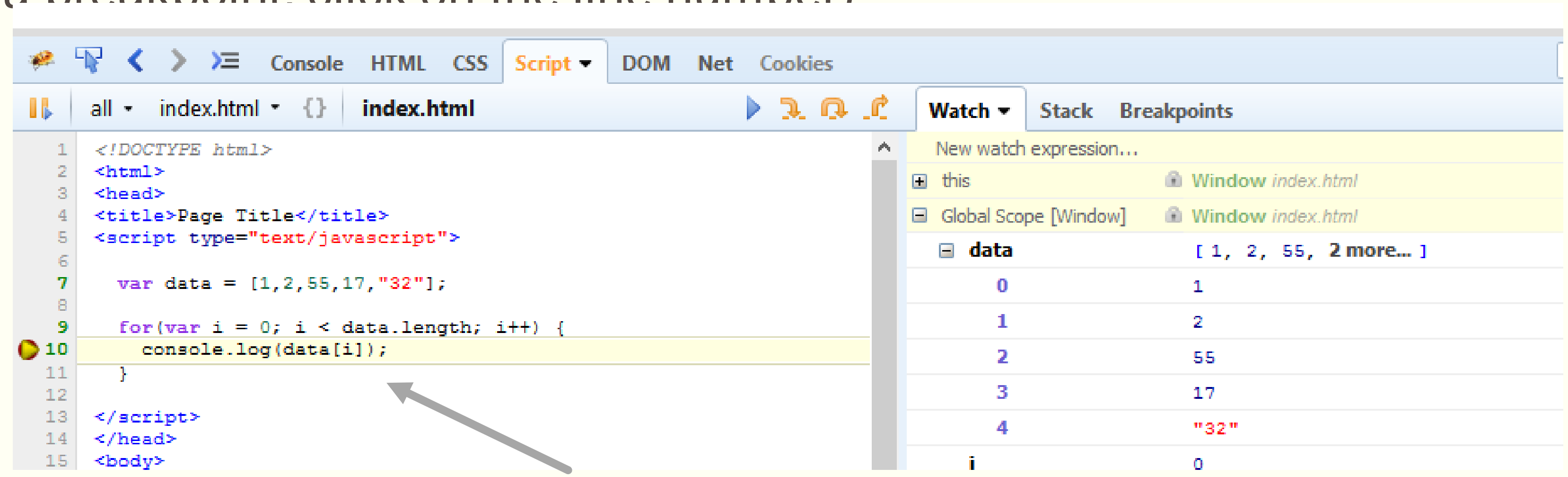


```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
<script type="text/javascript">

  var data = [1,2,55,17,"32"];

  for(var i = 0; i < data.length; i++) {
    console.log(data[i]);
  }

</script>
</head>
<body>
  <a class="btn btn-primary btn-large">I'm a cool button.</a>
  <div style="display: block; font-family: Verdana">Mindtree
    delivers technology services and accelerates growth for Global 1000
    companies by solving complex business challenges with breakthrough
    technical innovations. Mindtree specializes in e-commerce, mobility,
```

Stepping Through JavaScript

- UNDERSTANDING BEHAVIOR OF JAVASCRIPT CODE & DETECTING ERRORS WITH FIREBUG

- Create a breakpoint on line number 10 to analyse the problem. Refresh the page. (To create a breakpoint, click on the line number)



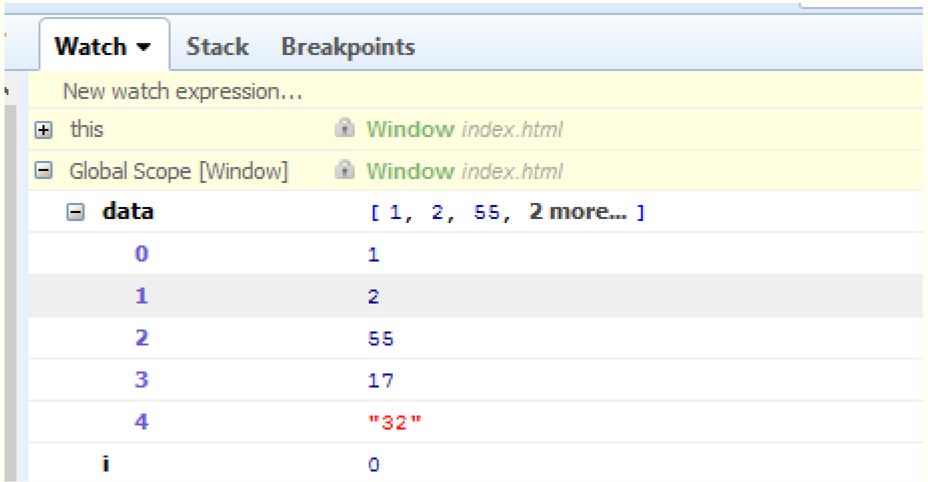
```
 1  <!DOCTYPE html>
 2  <html>
 3  <head>
 4  <title>Page Title</title>
 5  <script type="text/javascript">
 6
 7    var data = [1,2,55,17,"32"];
 8
 9    for(var i = 0; i < data.length; i++) {
10      console.log(data[i]);
11    }
12
13  </script>
14  </head>
15  <body>
```

```
⊞ this                    Window index.html
⊟ Global Scope [Window]   Window index.html
  ⊟ data                  [ 1, 2, 55, 2 more... ]
        0                 1
        1                 2
        2                 55
        3                 17
        4                 "32"
    i                     0
```

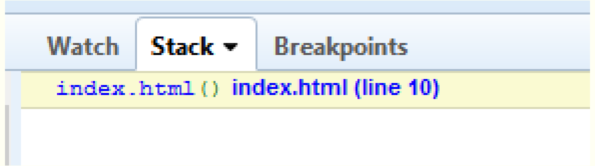
JavaScript has stopped on the breakpoint.

When Firebug Hits a Breakpoint, what can we see?

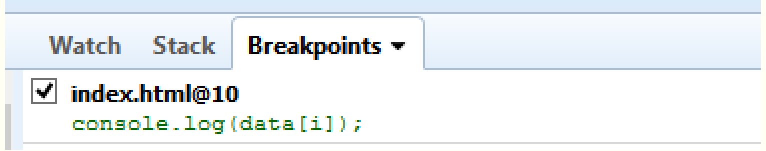
- Watch: Global elements & local elements
- Stack: Display the Stack trace
- Breakpoi



New watch expression...	
⊞ this	Window index.html
⊟ Global Scope [Window]	Window index.html
⊟ data	[1, 2, 55, 2 more...]
0	1
1	2
2	55
3	17
4	"32"
i	0



index.html() index.html (line 10)

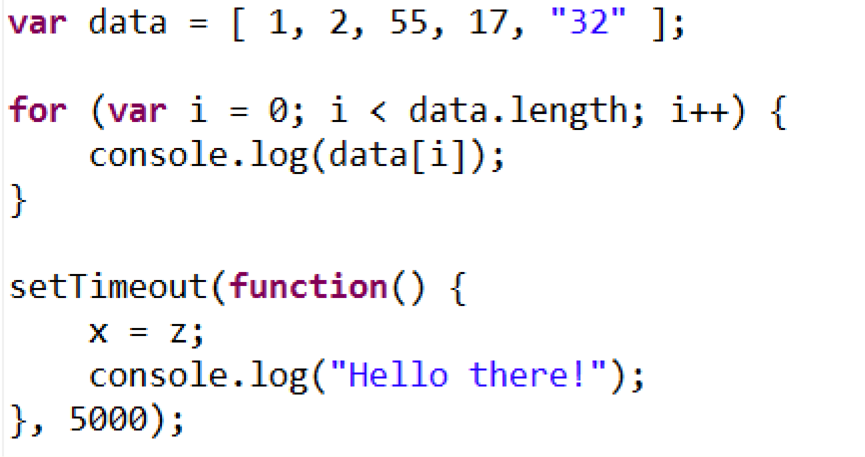


☑ index.html@10

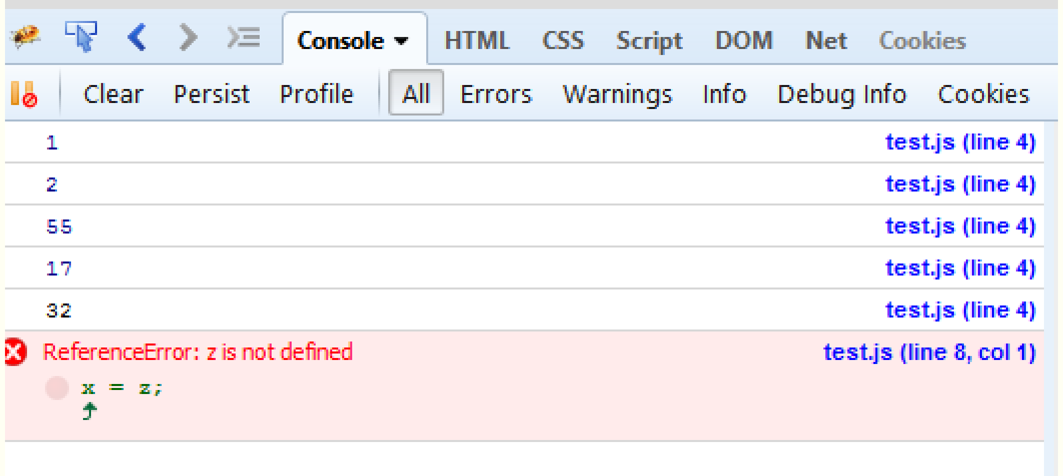
```
console.log(data[i]);
```

Errors in the Console

- By default, JavaScript code stops executing from the line an error occurs.
- •If we didn’t have Firebug, we would expect Hello there! to appear in the console, but it didn’t, and we would debug manually by using alert() or document.write()



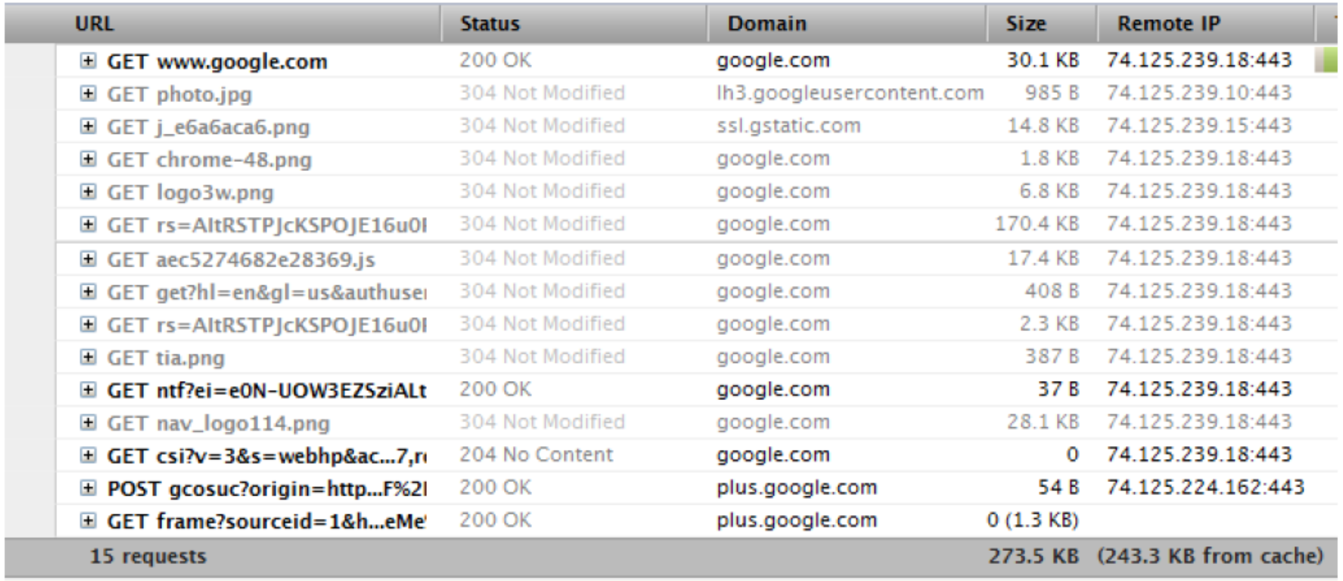
```
var data = [ 1, 2, 55, 17, "32" ];

for (var i = 0; i < data.length; i++) {
    console.log(data[i]);
}

setTimeout(function() {
    x = z;
    console.log("Hello there!");
}, 5000);
```



```
1                                          test.js (line 4)
2                                          test.js (line 4)
55                                         test.js (line 4)
17                                         test.js (line 4)
32                                         test.js (line 4)
ReferenceError: z is not defined           test.js (line 8, col 1)
x = z;
```

Detecting Web Performance Using Firebug

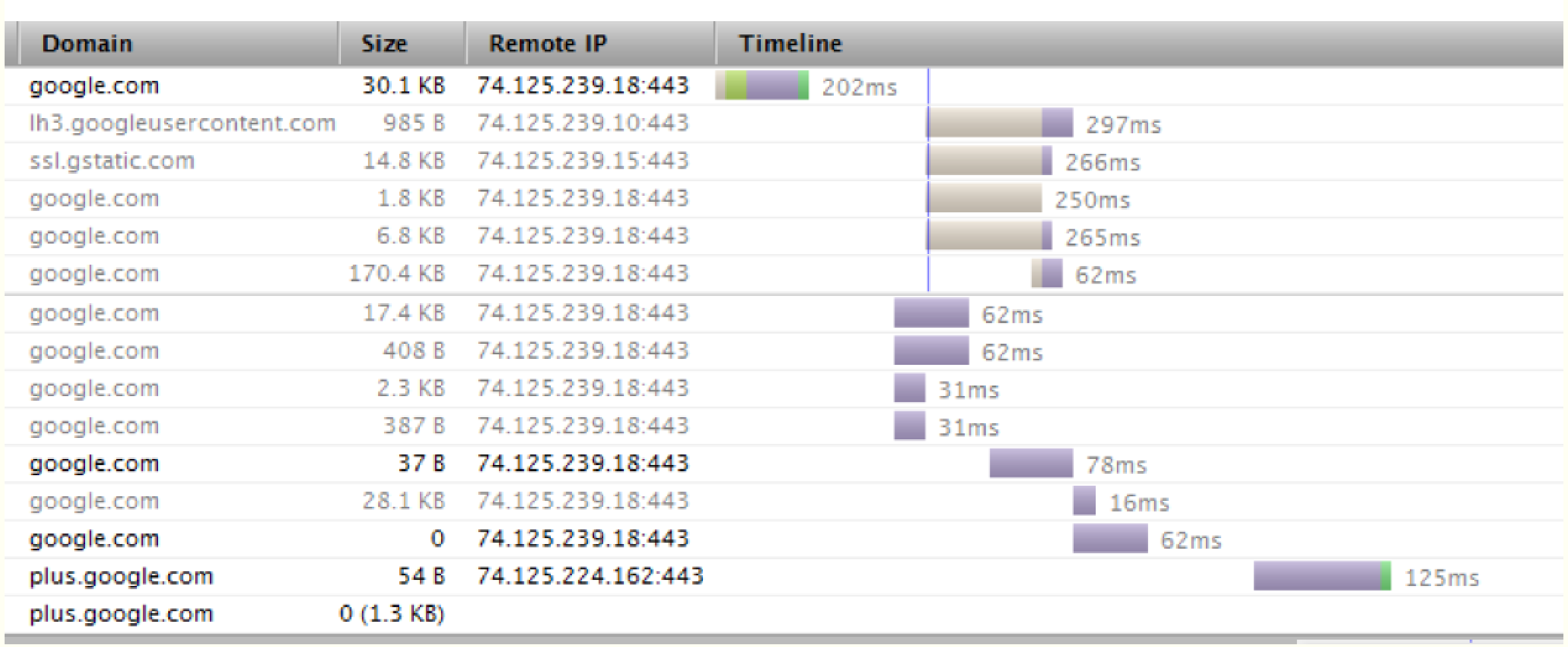
- After refreshing, something like the above should appear.
- There were 15 requests, 273.5 KB in total size, 243.3 KB from cache.
- The remote IP is 74.125.239.18: 443 and 74.125.224.162:443 for one req.
- The status of each request, and what type it was
- Here's what we can say about the performance of http://www.google.com



URL	Status	Domain	Size	Remote IP
⊞ GET www.google.com	200 OK	google.com	30.1 KB	74.125.239.18:443
⊞ GET photo.jpg	304 Not Modified	lh3.googleusercontent.com	985 B	74.125.239.10:443
⊞ GET j_e6a6aca6.png	304 Not Modified	ssl.gstatic.com	14.8 KB	74.125.239.15:443
⊞ GET chrome-48.png	304 Not Modified	google.com	1.8 KB	74.125.239.18:443
⊞ GET logo3w.png	304 Not Modified	google.com	6.8 KB	74.125.239.18:443
⊞ GET rs=AItRSTPJcKSPOJE16u0I	304 Not Modified	google.com	170.4 KB	74.125.239.18:443
⊞ GET aec5274682e28369.js	304 Not Modified	google.com	17.4 KB	74.125.239.18:443
⊞ GET get?hl=en&gl=us&authuse	304 Not Modified	google.com	408 B	74.125.239.18:443
⊞ GET rs=AItRSTPJcKSPOJE16u0I	304 Not Modified	google.com	2.3 KB	74.125.239.18:443
⊞ GET tia.png	304 Not Modified	google.com	387 B	74.125.239.18:443
⊞ GET ntf?ei=e0N-UOW3EZSziALt	200 OK	google.com	37 B	74.125.239.18:443
⊞ GET nav_logo114.png	304 Not Modified	google.com	28.1 KB	74.125.239.18:443
⊞ GET csi?v=3&s=webhp&ac...7,r	204 No Content	google.com	0	74.125.239.18:443
⊞ POST gcosuc?origin=http...F%2I	200 OK	plus.google.com	54 B	74.125.224.162:443
⊞ GET frame?sourceid=1&h...eMe	200 OK	plus.google.com	0 (1.3 KB)	
15 requests			273.5 KB	(243.3 KB from cache)

Detecting Web Performance Using Firebug

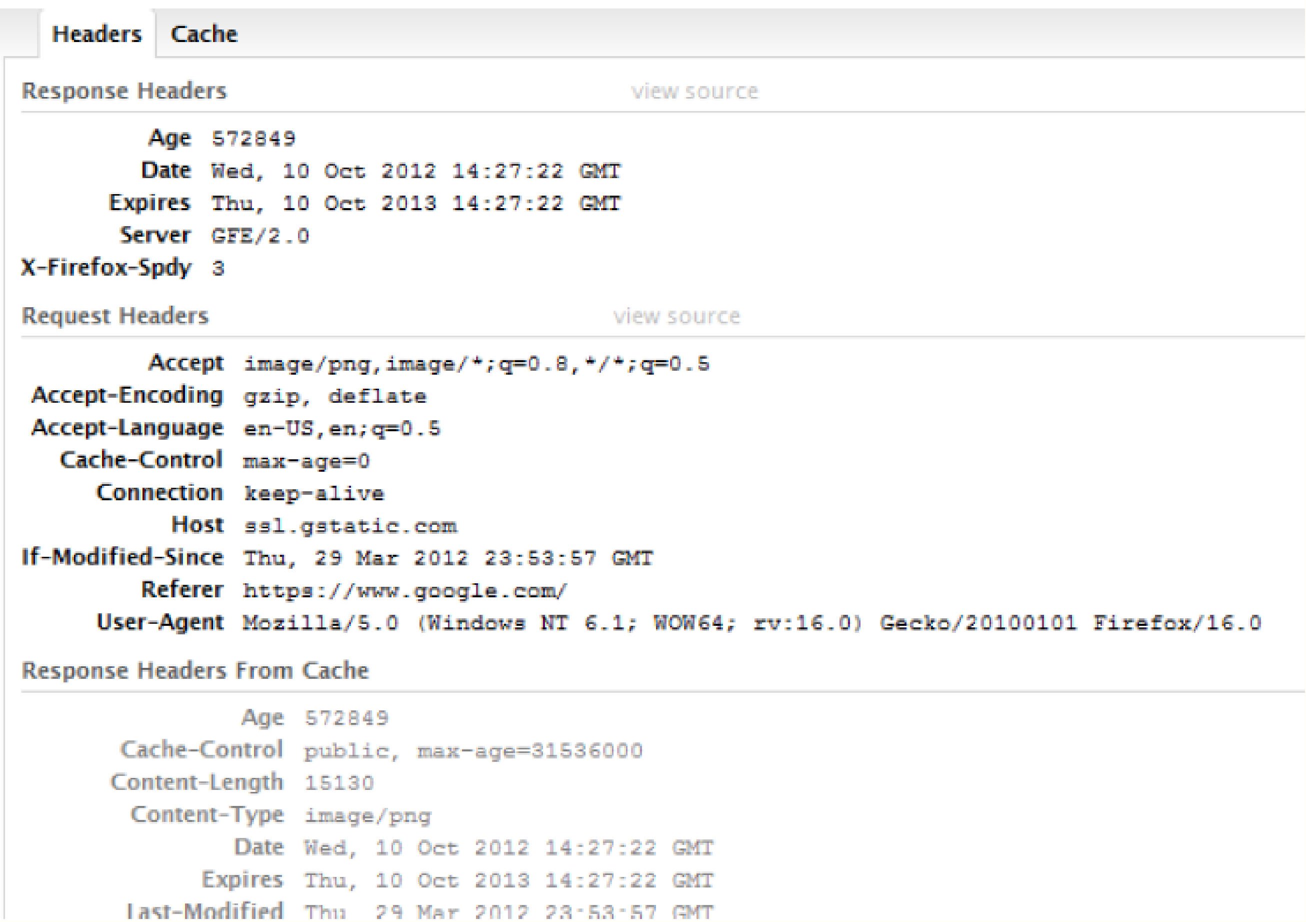
- The longest request is 297 ms.
- We also know the timeline of each request 6. The legend on the right indicates what each request was doing



Domain	Size	Remote IP	Timeline
google.com	30.1 KB	74.125.239.18:443	202ms
lh3.googleusercontent.com	985 B	74.125.239.10:443	297ms
ssl.gstatic.com	14.8 KB	74.125.239.15:443	266ms
google.com	1.8 KB	74.125.239.18:443	250ms
google.com	6.8 KB	74.125.239.18:443	265ms
google.com	170.4 KB	74.125.239.18:443	62ms
google.com	17.4 KB	74.125.239.18:443	62ms
google.com	408 B	74.125.239.18:443	62ms
google.com	2.3 KB	74.125.239.18:443	31ms
google.com	387 B	74.125.239.18:443	31ms
google.com	37 B	74.125.239.18:443	78ms
google.com	28.1 KB	74.125.239.18:443	16ms
google.com	0	74.125.239.18:443	62ms
plus.google.com	54 B	74.125.224.162:443	125ms
plus.google.com	0 (1.3 KB)		

Analyzing a Request using Firebug

- We can see the details of the request in the Headers Tab



Response Headers

```
          Age  572849
         Date  Wed, 10 Oct 2012 14:27:22 GMT
      Expires  Thu, 10 Oct 2013 14:27:22 GMT
       Server  GFE/2.0
X-Firefox-Spdy  3
```

Request Headers

```
          Accept  image/png,image/*;q=0.8,*/*;q=0.5
 Accept-Encoding  gzip, deflate
 Accept-Language  en-US,en;q=0.5
   Cache-Control  max-age=0
      Connection  keep-alive
            Host  ssl.gstatic.com
If-Modified-Since  Thu, 29 Mar 2012 23:53:57 GMT
         Referer  https://www.google.com/
      User-Agent  Mozilla/5.0 (Windows NT 6.1; WOW64; rv:16.0) Gecko/20100101 Firefox/16.0
```

Response Headers From Cache

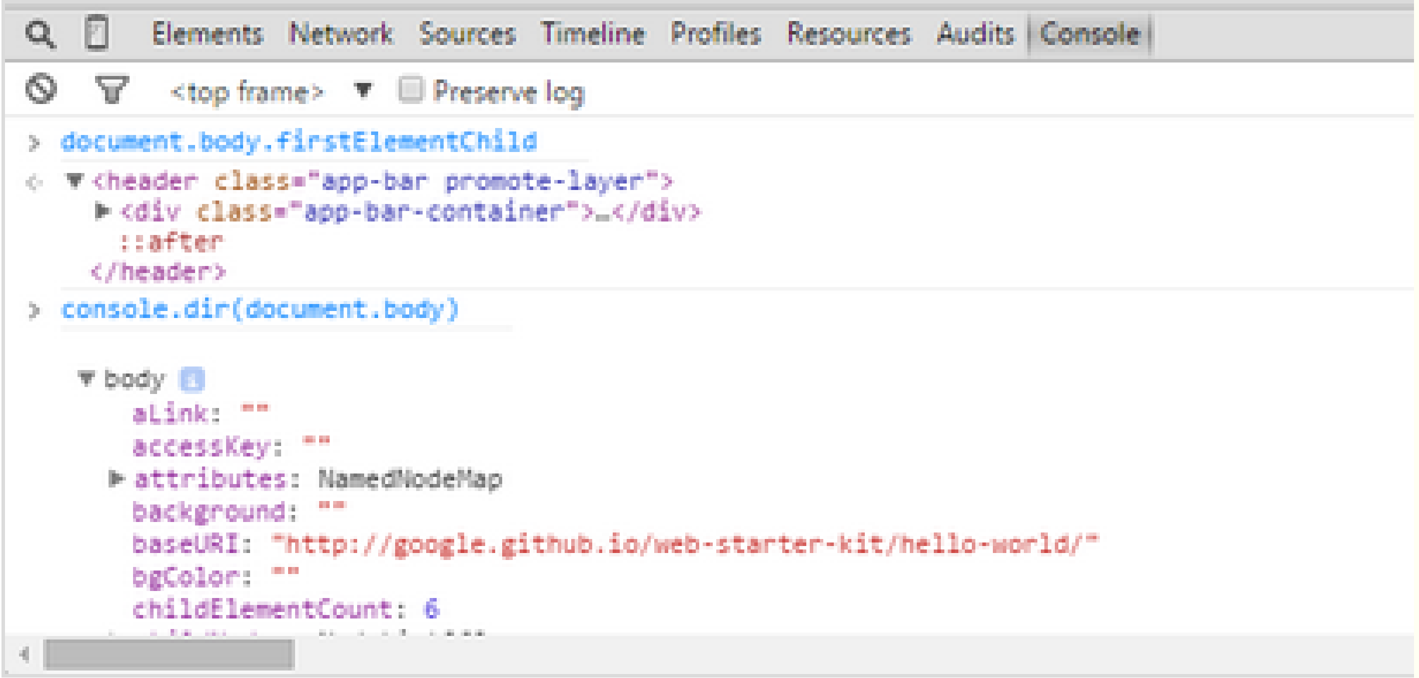
```
           Age  572849
 Cache-Control  public, max-age=31536000
Content-Length  15130
  Content-Type  image/png
          Date  Wed, 10 Oct 2012 14:27:22 GMT
       Expires  Thu, 10 Oct 2013 14:27:22 GMT
 Last-Modified  Thu, 29 Mar 2012 23:53:57 GMT
```

Chrome DevTools

- Similar to FireBug the Chrome Developer Tools (DevTools for short), are a set of web authoring and debugging tools built into Google Chrome.
- The DevTools provide web developers deep access into the internals of the browser and their web application.
- Use the DevTools to efficiently track down layout issues, set JavaScript breakpoints, and get insights for code optimization.

Working with the Console

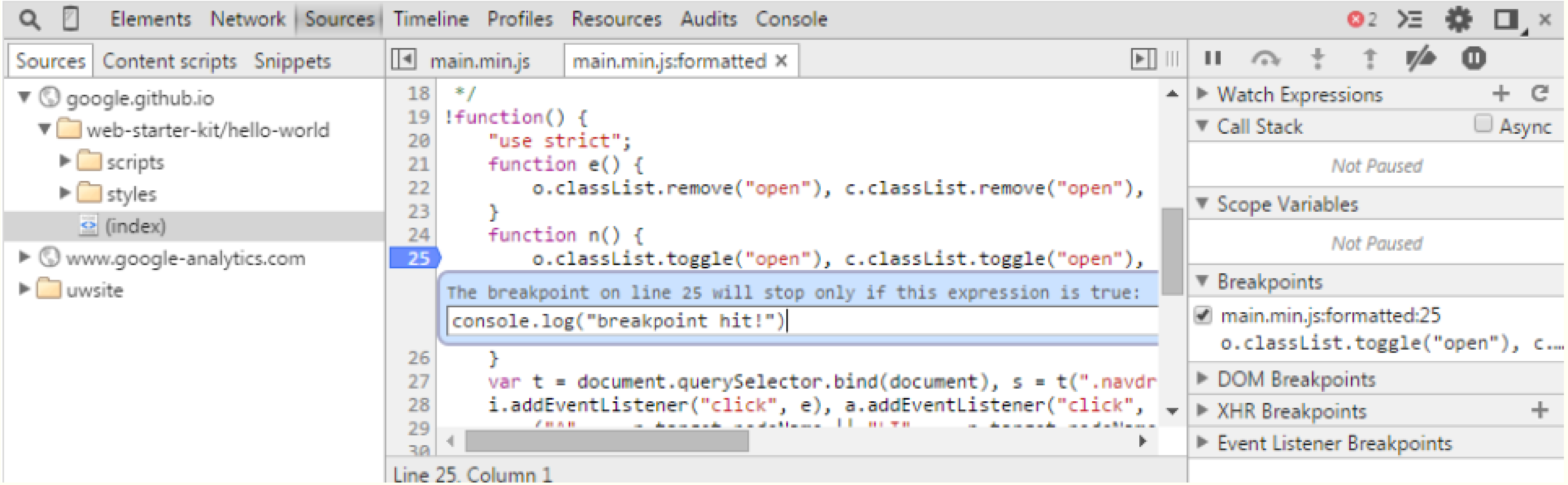
- The JavaScript Console provides two primary functions for developers testing web pages and applications. It is a place to:
  - Log diagnostic information in the development process.
  - A shell prompt which can be used to interact with the document and DevTools.



```
> document.body.firstElementChild
<· ▼<header class="app-bar promote-layer">
    ▶<div class="app-bar-container">…</div>
     ::after
   </header>
> console.dir(document.body)

   ▼ body
       aLink: ""
       accessKey: ""
     ▶ attributes: NamedNodeMap
       background: ""
       baseURI: "http://google.github.io/web-starter-kit/hello-world/"
       bgColor: ""
       childElementCount: 6
```

Debugging JavaScript

- As the complexity of JavaScript applications increase, developers need powerful debugging tools to help quickly discover the cause of an issue and fix it efficiently. The Chrome DevTools include a number of useful tools to help make debugging JavaScript less painful



```
18  */
19 !function() {
20     "use strict";
21     function e() {
22         o.classList.remove("open"), c.classList.remove("open"),
23     }
24     function n() {
25         o.classList.toggle("open"), c.classList.toggle("open"),
```

The breakpoint on line 25 will stop only if this expression is true:

```
console.log("breakpoint hit!")
```

```
26     }
27     var t = document.querySelector.bind(document), s = t(".navdr
28     i.addEventListener("click", e), a.addEventListener("click",
29
30
```

Line 25, Column 1

Call Stack: Not Paused

Scope Variables: Not Paused

Breakpoints:

☑ main.min.js:formatted:25
o.classList.toggle("open"), c.…