

WA2425 Introduction to AngularJS Programming



Web Age Solutions Inc.
USA: 1-877-517-6540
Canada: 1-866-206-4644
Web: <http://www.webagesolutions.com>

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

IBM, WebSphere, DB2 and Tivoli are trademarks of the International Business Machines Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

For customizations of this book or other sales inquiries, please contact us at:

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com

Canada: 1-866-206-4644 toll free, email: getinfo@webagesolutions.com

Copyright © 2014 Web Age Solutions Inc.

This publication is protected by the copyright laws of Canada, United States and any other country where this book is sold. Unauthorized use of this material, including but not limited to, reproduction of the whole or part of the content, re-sale or transmission through fax, photocopy or e-mail is prohibited. To obtain authorization for any such activities, please write to:

Web Age Solutions Inc.
439 University Ave
Suite 820
Toronto
Ontario, M5G 1Y8

Table of Contents

Chapter 1 - Introduction to JavaScript.....	10
1.1 What JavaScript Is.....	11
1.2 What JavaScript Is.....	13
1.3 What JavaScript Is Not.....	15
1.4 Not All JavaScripts are Created Equal	16
1.5 ECMAScript Language and Specification.....	17
1.6 What JavaScript Can Do.....	19
1.7 What JavaScript Can't Do.....	20
1.8 JavaScript on the Server-side.....	21
1.9 Elements of JavaScript.....	22
1.10 Values, Variables and Functions.....	23
1.11 Embedded Scripts.....	25
1.12 External Scripts.....	27
1.13 Browser Dialog Boxes.....	29
1.14 What is AJAX?.....	31
1.15 Summary.....	32
Chapter 2 - JavaScript Fundamentals.....	33
2.1 Variables.....	34
2.2 JavaScript Reserved Words	36
2.3 Dynamic Types.....	37
2.4 JavaScript Strings.....	39
2.5 Escaping Control Characters.....	40
2.6 What is False in JavaScript?.....	41
2.7 Numbers.....	42
2.8 The Number Object	44
2.9 Not A Number (NaN) Reserved Keyword.....	45
2.10 JavaScript Objects.....	46
2.11 Operators.....	47
2.12 Primitive Values vs Objects.....	49
2.13 Primitive Values vs Objects.....	50
2.14 Flow Control.....	51
2.15 'if' Statement.....	52
2.16 'if...else' Statement.....	55
2.17 'switch' Statement.....	57
2.18 'for' Loop.....	59
2.19 'for / in' Loop.....	61
2.20 'while' Loop.....	62
2.21 'do...while' Loop.....	64
2.22 Break and Continue.....	66
2.23 Labeled Statements.....	68
2.24 The undefined and null Keywords.....	70
2.25 Checking for undefined and null.....	71
2.26 Checking Types with typeof Operator	72
2.27 Date Object.....	73

2.28 Document Object.....	75
2.29 Other Useful Objects.....	77
2.30 Browser Object Detection.....	78
2.31 The eval Function	80
2.32 Enforcing the Strict Mode	81
2.33 Summary.....	82
Chapter 3 - JavaScript DOM API.....	83
3.1 What is DOM?.....	84
3.2 Element Hierarchy.....	86
3.3 DOM Standardization.....	88
3.4 The Document Object.....	90
3.5 The Document Object.....	92
3.6 The Document Object.....	94
3.7 Nodes and Elements.....	96
3.8 The Element Object.....	98
3.9 The Element Object.....	100
3.10 The Element Object.....	102
3.11 The Element Object.....	104
3.12 The Element Object.....	106
3.13 Element Event Handlers.....	107
3.14 The window Object.....	109
3.15 The window Object.....	110
3.16 The window Object.....	112
3.17 The Frame Object.....	113
3.18 The History Object.....	114
3.19 Summary.....	115
Chapter 4 - JavaScript Functions.....	116
4.1 Functions Defined.....	117
4.2 Declaring Functions.....	118
4.3 Function Arguments.....	120
4.4 More on Function Arguments.....	121
4.5 Return Values.....	123
4.6 Multiple Return Values in ECMAScript 6	125
4.7 Optional Default Parameter Values.....	126
4.8 Emulating Optional Default Parameter Values.....	127
4.9 Anonymous Function Expressions.....	128
4.10 Functions as a Way to Create Private Scope.....	129
4.11 Linking Functions to Page Elements.....	130
4.12 Local and Global Variables.....	132
4.13 Local and Global Variables.....	134
4.14 Declaring Object Methods.....	135
4.15 The arguments Parameter	136
4.16 Example of Using arguments Parameter.....	137
4.17 Summary.....	138
Chapter 5 - JavaScript Arrays.....	139
5.1 Arrays Defined.....	140

5.2 Creating an Array.....	141
5.3 The length Array Member.....	143
5.4 Traversing an Array.....	144
5.5 Appending to an Array.....	146
5.6 Deleting Elements.....	148
5.7 Inserting Elements.....	149
5.8 Other Array Methods.....	150
5.9 Other Array Methods.....	151
5.10 Accessing Objects as Arrays.....	152
5.11 Summary.....	153
Chapter 6 - Advanced Objects and Functionality in JavaScript.....	154
6.1 Basic Objects.....	155
6.2 Constructor Function.....	157
6.3 More on the Constructor Function.....	159
6.4 Object Properties.....	160
6.5 Deleting a Property.....	162
6.6 The instanceof Operator.....	163
6.7 Object Properties.....	164
6.8 Constructor and Instance Objects.....	166
6.9 Constructor Level Properties.....	168
6.10 Namespace.....	169
6.11 Functions are First-Class Objects.....	171
6.12 Closures.....	173
6.13 Closure Examples.....	174
6.14 Closure Examples.....	175
6.15 Closure Examples.....	176
6.16 Private Variables with Closures.....	177
6.17 Immediately Invoked Function Expression (IIFE).....	178
6.18 Prototype.....	180
6.19 Inheritance in JavaScript.....	182
6.20 The Prototype Chain	184
6.21 Traversing Prototype Property Hierarchy.....	186
6.22 Prototype Chain.....	188
6.23 Inheritance Using Prototype.....	190
6.24 Extending Inherited Behavior.....	192
6.25 Extending Inherited Behavior.....	194
6.26 Enhancing Constructors.....	196
6.27 Improving Constructor Performance.....	198
6.28 Inheritance with Object.create.....	200
6.29 The hasOwnProperty Method	201
6.30 Summary.....	202
Chapter 7 - Introduction to AngularJS.....	203
7.1 What is AngularJS?.....	204
7.2 Why AngularJS?.....	206
7.3 Scope and Goal of AngularJS.....	208
7.4 Using AngularJS.....	209

7.5 A Very Simple AngularJS Application.....	210
7.6 Building Blocks of an AngularJS Application.....	212
7.7 Use of Model View Controller (MVC) Pattern.....	213
7.8 A Simple MVC Application.....	214
7.9 The View.....	215
7.10 The Controller.....	217
7.11 Data Binding.....	219
7.12 Basics of Dependency Injection (DI).....	220
7.13 Other Client Side MVC Frameworks.....	222
7.14 Summary.....	223
Chapter 8 - AngularJS Module.....	224
8.1 What is a Module?.....	225
8.2 Benefits of Having Modules.....	226
8.3 Life Cycle of a Module.....	227
8.4 The Configuration Phase.....	228
8.5 The Run Phase.....	229
8.6 Module Wide Data Using Value.....	230
8.7 Module Wide Data Using Constant.....	231
8.8 Module Dependency.....	232
8.9 Using Multiple Modules in a Page.....	233
8.10 Summary.....	235
Chapter 9 - AngularJS Controllers.....	236
9.1 Controller Main Responsibilities.....	237
9.2 About Constructor and Factory Functions.....	238
9.3 Defining a Controller.....	240
9.4 Using the Controller.....	241
9.5 Controller Constructor Function.....	242
9.6 More About Scope.....	243
9.7 Example Scope Hierarchy.....	245
9.8 Using Scope Hierarchy.....	247
9.9 Modifying Objects in Parent Scope.....	248
9.10 Modified Parent Scope in DOM.....	250
9.11 Handling Events.....	251
9.12 Another Example for Event Handling.....	252
9.13 Storing Model in Instance Property.....	255
9.14 Summary.....	256
Chapter 10 - AngularJS Expressions.....	257
10.1 Expressions.....	258
10.2 Operations Supported in Expressions.....	260
10.3 Operations Supported in Expressions.....	261
10.4 AngularJS Expressions vs JavaScript Expressions.....	262
10.5 AngularJS Expressions are Safe to Use!	263
10.6 What is Missing in Expressions	264
10.7 Considerations for Using src and href Attributes in Angular.....	265
10.8 Examples of ng-src and ng-href Directives.....	266
10.9 Summary.....	267

Chapter 11 - Basic View Directives.....	268
11.1 Introduction to AngularJS Directives.....	269
11.2 Controlling Element Visibility.....	270
11.3 Adding and Removing an Element.....	271
11.4 Dynamically Changing Style Class.....	272
11.5 The ng-class Directive.....	273
11.6 Example Use of ng-class.....	274
11.7 Setting Image Source.....	275
11.8 Setting Hyperlink Dynamically.....	276
11.9 Preventing Initial Flash.....	277
11.10 Summary.....	278
Chapter 12 - Advanced View Directives.....	279
12.1 The ng-repeat Directive	280
12.2 Example Use of ng-repeat.....	281
12.3 Dynamically Adding Items.....	282
12.4 Special Properties.....	284
12.5 Example: Using the \$index Property.....	285
12.6 Scope and Iteration.....	286
12.7 Event Handling in Iterated Elements.....	287
12.8 The ng-switch Directive.....	289
12.9 Example Use of ng-switch.....	290
12.10 Inserting External Template using ng-include.....	292
12.11 Summary.....	294
Chapter 13 - Working with Forms.....	295
13.1 Forms and AngularJS.....	296
13.2 Scope and Data Binding.....	297
13.3 Role of a Form.....	299
13.4 Using Input Text Box.....	300
13.5 Using Radio Buttons.....	301
13.6 Using Checkbox.....	302
13.7 Using Checkbox - Advanced.....	303
13.8 Using Select.....	305
13.9 Using Select – Advanced.....	306
13.10 Reacting to Model Changes in a Declarative Way	308
13.11 Example of Using the on-change Directive	309
13.12 Summary.....	310
Chapter 14 - Formatting Data with Filters in AngularJS.....	311
14.1 What are AngularJS Filters?	312
14.2 The Filter Syntax.....	313
14.3 Angular Filters.....	314
14.4 More Angular Filters	315
14.5 Using Filters in JavaScript.....	316
14.6 Using Filters.....	317
14.7 A More Complex Example	318
14.8 The date Filter	319
14.9 The date's format Parameter.....	320

14.10 Examples of Using the date Filter	321
14.11 The limitTo Filter	322
14.12 Using limitTo Filter	323
14.13 Filter Performance Considerations.....	324
14.14 Summary.....	325
Chapter 15 - AngularJS \$watch Scope Function.....	326
15.1 The \$watch Function	327
15.2 The \$watch Function Signature.....	328
15.3 The \$watch Function Details.....	329
15.4 Canceling the Watch Action	330
15.5 Example of Using \$watch.....	331
15.6 Things to be Aware Of.....	332
15.7 More Things to Be Aware Of.....	333
15.8 Performance Considerations.....	334
15.9 Speeding Things Up.....	335
15.10 Summary.....	336
Chapter 16 - Communicating with Web Servers.....	337
16.1 The \$http AngularJS Service	338
16.2 The Promise interface.....	340
16.3 \$http Set Up.....	342
16.4 \$http Function Invocation.....	344
16.5 Callback Parameters.....	345
16.6 Request Configuration Properties.....	346
16.7 Shortcut Methods.....	347
16.8 Complete List of Shortcut Methods.....	348
16.9 Passing Parameters to \$http GET Requests.....	349
16.10 Working with JSON Response.....	350
16.11 Making a POST Request.....	351
16.12 Combining \$http POST Request Data with URL Parameters.....	352
16.13 The then() Method of the Promise.....	353
16.14 The Response Object.....	354
16.15 Setting Up HTTP Request Headers.....	355
16.16 Caching Responses.....	356
16.17 Setting the Request Timeout.....	357
16.18 Unit Testing with ngMock.....	358
16.19 Writing Unit Tests.....	359
16.20 Summary.....	360
Chapter 17 - Custom Directives.....	361
17.1 What are Directives?.....	362
17.2 Directive Usage Types.....	363
17.3 Directive Naming Convention.....	364
17.4 Defining a Custom Directive.....	365
17.5 Using the Directive.....	366
17.6 Scope of a Directive.....	367
17.7 Isolating Scope.....	369
17.8 Example Scope Isolation.....	370

17.9 Using External Template File.....	372
17.10 Manipulating a DOM Element.....	373
17.11 The Link Function.....	374
17.12 Event Handling from a Link Function.....	375
17.13 Wrapping Other Elements.....	376
17.14 Summary.....	379
Chapter 18 - AngularJS Services.....	380
18.1 Introduction to Services.....	381
18.2 Defining a Service.....	382
18.3 The factory() Method Approach.....	383
18.4 The service() Method Approach.....	384
18.5 The provider() Method Approach.....	385
18.6 Using a Service.....	387
18.7 Configuring a Service using its Provider.....	388
18.8 Summary.....	390
Chapter 19 - Unit Test using Jasmine.....	391
19.1 Introduction to Node JS.....	392
19.2 What is Jasmine.....	393
19.3 Running Jasmine.....	395
19.4 Running Tests.....	397
19.5 Example Explained.....	399
19.6 End-to-End Testing with Protractor.....	402
19.7 Writing E2E Test with Protractor.....	404
19.8 Summary.....	406

Chapter 1 - Introduction to JavaScript

Objectives

Key objectives of this chapter

- JavaScript in context
- Major elements of JavaScript
- Adding JavaScript code to a web page



1.1 What JavaScript Is

- JavaScript is a programming language supported by web browsers that can be used to add interactivity to web pages
- It is often called a “scripting language” as opposed to a “programming language” but this is more because you do not need to do any kind of compilation or building of your JavaScript source code before it can execute
- Code for JavaScript is either provided directly in the HTML code of web pages or pages link to external JavaScript files with the code that are also downloaded from the server
 - ◇ This is done with a **<script>** tag that either has embedded JavaScript source code or links to another file that has the code



1.2 What JavaScript Is

- JavaScript executes as a “client-side” language which means it runs inside the browser after the relevant HTML and JavaScript code has been downloaded from the server
- JavaScript is consistently one of the most “popular” programming languages in use
- JavaScript was created by Brendan Eich for Netscape over a very short period of time (time, as usual, was money) with certain design flaws that are still being fixed



1.3 What JavaScript Is Not

- JavaScript is NOT related to Java
 - ◇ JavaScript was originally designed by Netscape to add scripting to their browser. It was called 'LiveScript'.
 - ◇ The Java language was getting a ton of press coverage at the time so Netscape wanted some of that attention to rub off on its scripting language, so they renamed it JavaScript



1.4 Not All JavaScripts are Created Equal ...

- JavaScript is not exactly the same in all browsers
 - ◇ Besides the original JavaScript from Netscape, Microsoft came up with a scripting language called JScript for Internet Explorer
 - ◇ With much older browsers this was more of an issue to write code that would function no matter what browser a user was using
 - ◇ In 1997 the ECMAScript specification was released which looked to standardize JavaScript code so writing ECMAScript-compatible code will help ensure compatibility with even some fairly old browsers
 - ◇ Sometimes advanced JavaScript frameworks will provide support for ensuring your code is compatible with various browsers



1.5 ECMAScript Language and Specification

- ECMAScript is the scripting language and specification standardized by Ecma International in the ECMA-262 specification and ISO/IEC 16262
- The ECMAScript spec has been implemented in such languages (as we know them) as JavaScript, JScript and ActionScript
 - ◇ You can see JavaScript, JScript, etc., as ECMAScript dialects
- Any references to the versions of JavaScript are, in fact, made to the version of the ECMAScript spec that JavaScript (more specifically, the hosting browser) implements
- The most widely implemented version of JavaScript in browsers is ECMAScript 5
 - ◇ The next release of ECMAScript specification, version 6 (a.k.a. “ES6 Harmony”), is scheduled for publishing in mid-2015



1.6 What JavaScript Can Do

- JavaScript can draw the attention of the user to certain elements of a page to highlight how the user can interact with the page
 - ◇ This could include changing the appearance of links or buttons when the mouse rolls over them to indicate they could be clicked
- JavaScript code can provide a client-side validation function to ensure that data submitted by forms on web pages is valid BEFORE the data is sent to the server
 - ◇ This can include immediate feedback of highlighting a field in red, for example, if the user moves away from that field with invalid data entered
- JavaScript could dynamically modify the elements of a page (like a form) based on data or actions the user has taken on the page
 - ◇ This could be something like changing the boxes and labels of a form to collect address or telephone information based on the user selection in a country field
- JavaScript can open new windows or popup dialogs to interact with the user instead of functioning only within a single browser window (or tab)



1.7 What JavaScript Can't Do

- JavaScript can't write data directly on a server machine
 - ◇ JavaScript can send a web request that might cause the server to store or update data but you would need a “server-side” technology to implement handling that request and modifying data on a server and this would not involve JavaScript
- The JavaScript code from a particular web site can't close a browser window or tab it hasn't opened or read data from a page that comes from another site
 - ◇ Otherwise malicious JavaScript code could close a legitimate site and open a fake site hoping you would enter sensitive information or directly read data from the page of the other site



1.8 JavaScript on the Server-side

- JavaScript is used primarily as a client-side scripting language to add interactivity to Web pages rendered in Web browsers, and is, increasingly, used on the server-side
- For example, **Node.js** is a platform for building fast, event-driven, scalable and data-intensive network applications written in JavaScript; it uses Chrome's JavaScript runtime called V8
- JavaScript can also run in the Java VM (via the Rhino technology)



1.9 Elements of JavaScript

- JavaScript is an "object-oriented" language which includes various elements
 - ◇ **Objects** – Unique instances in memory (of the browser) that contain state and behavior
 - ◇ **Properties** – Contain values for the state of an object and can refer to simple data types or other objects
 - ◇ **Methods** – The behavior an object can perform when the method is called on the object
- Another important aspect of JavaScript is the "Document Object Model" or "DOM"
 - ◇ This represents the elements of a web page in a hierarchy of objects and allows the JavaScript code to interact with the elements of the page



1.10 Values, Variables and Functions

- A **value** is a piece of information which can be of different types:
 - ◇ Number – any numeric value
 - ◇ String – characters inside quotation marks
 - ◇ Boolean – true or false
 - ◇ Null – empty value
 - ◇ Object – reference to an object
- A variable is a named reference to a stored value
 - ◇ Unlike some other programming languages, the type of a variable is not declared and it can reference values of different types within the script without errors
- A **function** is a set of JavaScript statements that performs a task
 - ◇ The function can be invoked, or called from other parts of the script
 - ◇ The function can declare parameters that are passed to it and return a value



1.11 Embedded Scripts

- JavaScript code can be embedded directly on a web page by using the **<script>** HTML tag
 - ◊ This tag can be used within the **<head>** section or **<body>** section

```
<body>  
  <script>  
    document.write("Hello, world!");  
  </script>
```

```
</body>
```

- You can also have a **<noscript>** tag for what the browser should display if JavaScript is disabled

```
<noscript>  
  <h2>This page requires JavaScript.</h2>  
</noscript>
```



1.12 External Scripts

- Embedding lots of JavaScript code directly in a page can make it tough to read and is not reusable on multiple pages
- Linking to external files that contain JavaScript code can be a more efficient way to separate HTML and JavaScript code and allow reuse from multiple pages
- First you create a text file with a **'.js'** file extension that contains the JavaScript code
 - ◇ You do not include any `<script>` tags within the external file

```
(Saved in file 'external.js')
```

```
alert("Hello, world from external script!");
```

- Then you link to the source of the external JavaScript file using the **'src'** attribute of the `<script>` tag on the page and the path to the location of the external file

```
<script src="external.js" />
```

- Although you can place the reference to the external script source in the `<head>` or `<body>` sections of the page, it is generally suggested to use the `<head>` section to ensure the external file is loaded before the page begins to display



1.13 Browser Dialog Boxes

- There are some basic functions built-in to JavaScript that allow you to display pop-up dialog boxes from the browser

- ◊ **'alert'** allows you to display a message with an 'OK' button

```
alert("Hello from JavaScript!");
```

- ◊ **'confirm'** displays a message with 'OK' and 'Cancel' buttons and returns a boolean depending on which button the user clicks ('OK' is true, 'Cancel' is false)

```
var continue = confirm("Do you want to continue?");  
if (continue) { ...
```

- ◊ **'prompt'** displays a message along with a text box for the user to fill in a value

- The user's value will be returned by the function if the user clicks the 'OK' button
- A 'null' is returned if the user clicks the 'Cancel' button
- This function takes two parameters for the message and the default value

```
var yourName = prompt("What is your name?", "");  
if (yourName != null) { alert("Hello " + yourName); }
```



1.14 What is AJAX?

- AJAX stands for Asynchronous JavaScript and XML (JSON)
- AJAX is often implemented with a combination of technologies including HTML, CSS, and JavaScript
- AJAX is an approach to developing web applications that differs from the traditional way of developing applications
 - ◇ In a traditional application, the user submits a form or clicks on a link, the browser sends a HTTP request to the server, the server replies with a fresh new HTML document that the browser renders as a new page
- AJAX differs in two main ways:
 - ◇ The browser makes a HTTP request that may or may not be due to a user action. For example, a clock application may automatically make a HTTP request every second
 - ◇ The reply from such a request does not contain a full new page. It contains information that is used to update portions of the existing page. For example, the clock application receives the current time at the server as a part of the HTTP reply and updates the time displayed in the page. The rest of the page remains as it is



1.15 Summary

- JavaScript is a very popular programming language
- Although the name implies some relationship with Java the two are actually quite different
- JavaScript contains objects with properties and methods, functions and variables
- JavaScript code can be embedded directly in a web page or linked from external source files

Chapter 2 - JavaScript Fundamentals

Objectives

Key objectives of this chapter

- JavaScript variables
- Operators and flow control statements
- Useful built-in JavaScript objects



2.1 Variables

- JavaScript variables are "containers" for storing information that will be used later in the code
 - ◇ The value stored by the variable can be used by using the variable name in an expression
- Variables have a name, are case-sensitive, and must follow certain rules
 - ◇ Variable names must begin with a letter
 - ◇ Variable names can also begin with \$ and _ (although not as common)
 - ◇ Variable names can contain numbers but can't start with them
 - ◇ Variable names can't be reserved JavaScript words
- You declare a variable with the '**var**' keyword

```
var firstName;
```

- You assign a variable a value by using the name of the variable and an equals sign

```
firstName = "Susan";
```

```
var lastName = "Bishop";
```



2.2 JavaScript Reserved Words

arguments	break	case	catch
class	const	continue	debugger
default	delete	do	else
enum	export	extends	false
finally	for	function	if
implements	import	in	instanceof
interface	let	new	null
package	private	protected	public
return	static	super	switch
this	throw	true	try
typeof	var	void	while

- You are also not allowed to use these key words: *Infinity*, *NaN* (not a number), and *undefined*



2.3 Dynamic Types

- JavaScript has dynamic types which means the same variable can be used as different types
 - ◇ Since the declaration of the variable doesn't tie it to a specific type, you can assign a value of a different type to a variable later without error

```
var answer = 42;
```

```
answer = "So long and thanks for all the fish...";
```

- ◇ Although this might let you reuse variables for other data later, you must also be careful in case a variable is storing data of a type different than what you expect
- "Literals" are types of expressions in JavaScript that can be evaluated
 - ◇ String literals have quotation marks (single or double) and numeric literals have no quotes
 - ◇ Boolean literal values are 'true' or 'false' with no quotes

```
var continue = true;
```

- ◇ A literal value of 'null' means no value and is often used to clear a variable of a value

```
answer = null;
```



2.4 JavaScript Strings

- Strings in JavaScript are created directly using string literals
- Literals can be delimited either by single (') or double (") quotes

```
var str1 = 'The value of str1'; var str2 = 'The value of str2';
```

- You concatenate string using the "+" operator
- You can use the backslash (\) character to escape some control characters as well as allowing single or double quotes inside so delimited string, e.g.

```
var str1 = "The value of str1 is \"The value of str1 is ...\" ";
```



2.5 Escaping Control Characters

- JavaScript follows the C / Java special character escaping notation:

<code>\\</code>	backslash
<code>\n</code>	new line
<code>\r</code>	carriage return
<code>\t</code>	tab
<code>\b</code>	backspace



2.6 What is False in JavaScript?

- JavaScript treats the following values as the **false** boolean value:
 - ◇ Numbers: *0*, *NaN*
 - Any non-zero numbers evaluate to neither *true* nor *false*
 - ◇ Strings: Empty strings and strings containing whitespace characters, e.g. ' ', or " ", or '\t', etc.
 - Any non-empty strings evaluate to neither *true* nor *false*
 - ◇ Boolean literal: *false*
 - ◇ Keywords: *undefined* and *null* (more on those a bit later ...)



2.7 Numbers

- JavaScript has only one type of numbers which are double precision floating-point numbers (which contain IEEE 64 bit values)
 - ◇ JavaScript stores numbers in 64 bits as follows (as per the IEEE 754 Floating Point Standard): the fraction (mantissa) is stored in bits 0 to 51, and the exponent is stored in bits 52 to 62; bit 63 is used for the sign
 - ◇ This way you can store values in a range of about $\pm 10^{-308} \dots 10^{308}$
- In addition to regular notation for numbers, like `var a = 123; var b = 45.99;` you can also use scientific (exponent) notation, e.g. `var e = 1.23e2;` (will evaluate to 123)



2.8 The Number Object

- JavaScript uses a system object called *Number* as a holder of some useful constants, e.g.
 - ◇ **Number.MAX_VALUE**
 - 1.7976931348623157e+308
 - ◇ **Number.POSITIVE_INFINITY**
 - it represents infinity (*Infinity*); returned on math operation overflow
 - E.g. `var x = 1/0;` will result in x getting the "Infinity" reserved numeric value, but not a run-time exception
 - ◇ **Number.NEGATIVE_INFINITY**
 - It represents negative infinity (*-Infinity*); returned on math operation overflow
- The '**Number**' object is also a wrapper for numeric values with some methods to convert between various forms (decimal and exponential)

```
var x = new Number(123.45);  
alert(x.toExponential());
```



2.9 Not A Number (NaN) Reserved Keyword

- *NaN* is a special keyword in JavaScript referring to a value or evaluation result which is "Not a Number"
- JavaScript offers a system function *isNaN* for checking if the value passed to it as a parameter gets evaluated to the *NaN* value
 - ◇ `isNaN ("Sun Microsystems");` // returns **true**
 - ◇ `isNaN (123);` // returns **false**



2.10 JavaScript Objects

- JavaScript variables can also refer to objects
- There are two ways to create a direct object instance:
 - ◇ Using the Object constructor and then adding individual properties

```
person = new Object();  
person.firstname = "John";  
person.lastname = "Doe";  
person.age = 50;  
person.eyecolor = "blue";
```

- ◇ Using an object literal with curly braces

```
person = {firstname:"John", lastname:"Doe", age:50, eyecolor:"blue"};
```

- Object property values can be accessed using a 'dot' notation

```
alert(person.firstName);
```



2.11 Operators

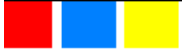
- JavaScript uses many of the same operators as other programming languages
 - ◇ Arithmetic: +, -, *, /, % (modulus, division remainder)
 - ◇ Increment/Decrement: ++, --
 - *Before* an operand, modifies a value before being used in the rest of the expression
 - *After* an operand, the original value is used before being modified
 - ◇ Assignment: =, +=, -=, *=, /=, %=
 - ◇ String concatenation with '+'
 - ◇ Logical: && (and), || (or), ! (not)
 - ◇ Comparison: <, <=, >, >=, ==, !=
 - ◇ "Conditional" or "Ternary" operator: (condition)?(true value):(false value)
- JavaScript also contains a unique '===' operator (triple equals sign) that tests if two operands are identical (equal value and equal type)
 - ◇ And also '!==' for "not identical" (different value or different type)

```
x = 5;  
x == "5";    // true  
x === "5";   // false
```



2.12 Primitive Values vs Objects

- JavaScript categorizes values into two groups (this is, to some extent, Java language influence):
 - ◇ The primitive values:
 - *booleans, strings, numbers, null, and undefined*
 - ◇ All other values are treated as objects (including arrays, dates, regex and functions)



2.13 Primitive Values vs Objects

- Objects differ from primitives in one substantial aspect: objects have their own identity, primitives have their identity expressed in their value

```
objA = { f : 99.99};
```

```
objB = { f : 99.99};
```

```
objA == objB; // false, same as with ===
```

```
objA.f == objB.f // true, the property f in both objects have the same value
```

```
objA.f === objB.f // also true (same value and same type)
```



2.14 Flow Control

- Similar to other programming languages, JavaScript has many control statements that are used to alter the flow of a program based on certain conditions
 - ◇ Decision statements
 - if
 - if...else
 - switch
 - ◇ Iteration statements
 - for
 - for / in
 - while
 - do...while



2.15 'if' Statement

- Most basic control flow statement
- Executes a statement if a condition is met
- The condition must be a boolean expression

```
if (condition)
    statement;
```

```
if (condition) {
    statement1;
    statement2;
    statement3;
}
```

```
var number = -4;
if (number < 0) {
    alert(
        "negative number");
    number = 0;
}
```



- May execute a single statement or a statement block
- It is possible to reassign variable values within the statement block



2.16 'if...else' Statement

- Extends basic **if** statement
- Executes **else** clause if condition of **if** statement is false
- May be a single statement or a statement block

```
if (condition)
    statement;
else
    statement;

if (condition) {
    statement1;
    statement2;
} else {
    statement1;
    statement2;
}
```



2.17 'switch' Statement

- Selects from multiple choices based on an expression
 - The value of the expression determines which case is executed
 - The **default** case handles values not matched by other cases
 - Use **break** to prevent the code from running into the next case automatically
- ```
switch (expression) {
 case n1:
 // case 1 statements
 break;
 case n2:
 // case 2 statements
 break;
 ...
 case n:
 // case n statements
 break;
 default:
 // default statements
 break;
}
```



## 2.18 'for' Loop

- Repeats a set of statements a certain number of times
- The control of the loop is defined within parenthesis after the for keyword
- The initialization, condition, and iteration parts of the control are separated by semi-colons
- More than one statement per control segment can be included separated by commas

```
for (var i = 0; i < 10; i++) {
 // executed 10 times
 ...
}

for (var i = 0, j = 10;
 i < j; i++) {
 // executed 10 times
 ...
}
```



## 2.19 'for / in' Loop

- The 'for/in' loop is a unique loop type for JavaScript
- This loops through the properties of an object, assigning each property name to a temporary variable
  - ◇ This could then be used to access the value of that property

```
var txt = "";
var person = {fname:"John", lname:"Doe", age:25};

for (var x in person)
{
 txt = txt + person[x];
}
// txt is 'JohnDoe25'
```



## 2.20 'while' Loop

- Repeats a set of statements while a condition is **true**
- The condition expression must return a **boolean** value
- If the condition is initially **false**, the loop block will not be executed at all
- Make sure something changes in the body of the loop to eventually let the condition be false

```
while (expression) {
 // These statements repeated
 ...
}

var number = 1.2;
while (number < 5.0) {
 number *= 2;
 ...
}
```



## 2.21 'do...while' Loop

- Similar to **while** loop
- Condition is tested at the end of the loop statement
- Semi-colon at the end of the loop
- Loop statement is always executed at least once

```
do {
 // These statements repeated
 // at least once
 ...
} while (expression);

var number = 6;
do {
 // this loop stills
 // executes once
 ...
} while (number < 5);
```





## 2.22 Break and Continue

- The **'break'** and **'continue'** statements can be used to alter the execution of loops
  - ◊ The break statement breaks the loop and continues executing the code after the loop (if any)

```
for (i = 0; i < 10; i++)
{
 if (i == 3) {
 break; // stops the entire loop after 0, 1, 2
 }
 x = x + "The number is " + i + "
";
}
```

- ◊ The continue statement breaks one iteration (in the loop) and continues with the next iteration in the loop

```
for (i = 0; i < 10; i++)
{
 if (i == 3) {
 continue; // skips over 3 for 0, 1, 2, 4, ...
 }
 x = x + "The number is " + i + "
";
}
```



## 2.23 Labeled Statements

- Statements may be labeled with an identifier
- Most often used with control statements to create a more complex structure
- Defined by placing an identifier and colon before the statement to be labeled

```
OuterLoop:
for (var i = 0; i < 10; i++) {
 // other statements
}
```

- Referred to by other statements

```
break OuterLoop;
```



## 2.24 The undefined and null Keywords

- JavaScript distinguishes between two situations when some information is missing
  - ◇ A variable has been declared but initialized (it does not have value), for example:
    - `var myVar; alert (myVar); // will show undefined`
    - `var myObj = {}; alert (myObj.myVar); // will show undefined (the myVar property has been defined by way of attaching it to the object variable via the dot notation, but not properly initialized)`
    - Your function gets fewer arguments than it declares
  - ◇ A "no object" situation, is treated as a *null*
    - It is used as the last element in the prototype chain and in some other arcane situations
- **Note:** JavaScript uses *undefined* in most cases of missing information, even in situations where you would expect a *null*



## 2.25 Checking for undefined and null

- You can use either JavaScript idiom when checking for missing information:
  - ◊ `if (myVar === undefined || myVar === null) { ... }`
  - ◊ `if (!x) { ... }`
- You can use the boolean not operator as both *undefined* and *null* are treated by JavaScript as **false**



## 2.26 Checking Types with `typeof` Operator

- The *typeof* operator is used in JavaScript to find out the type of primitives and objects
- Here are some of the examples:

```
typeof false; // "boolean"
```

```
typeof 123; // "number"
```

```
typeof "Yo JavaScript!"; // "string"
```

```
typeof new Object(); // "object"
```

```
typeof {}; // "object"
```



## 2.27 Date Object

- JavaScript has a built-in '**Date**' object to work with date and time values
- Constructing a new object without any parameters is the current date and time

```
var now = new Date();
```

- There are three other ways to initialize a Date object:
  - ◊ With the last approach, not specifying a value causes a '0' to be used

```
new Date(milliseconds) // milliseconds since 1970/01/01
```

```
new Date(dateString)
```

```
new Date(year, month, day, hours, minutes, seconds,
 milliseconds)
```

```
var birthday = new Date(1970, 3, 15);
```

- There are various methods like 'getFullYear', 'getDate', 'setHours', etc.

```
var age = now.getFullYear() - birthday.getFullYear();
```

- A later version of JavaScript also added methods for UTC (Coordinated Universal Time)



## 2.28 Document Object

- The '**Document**' object is part of the Document Object Model (DOM) API and provides access to the elements of the page
  - ◊ The DOM API is very extensive and is only introduced at a high level here
- There is a '**document**' variable that is automatically initialized as the reference to the current page
- One of the most common ways to access elements in the page is by id using the 'getElementById' method
- You can also change the content of an element with the 'innerHTML' property

```
document.getElementById("date").innerHTML = new Date();
```

- You can also alter the structure of the page with various methods to add and delete elements like 'createElement', 'appendChild', 'removeChild', and 'replaceChild'
- There is also a 'write' method on the Document object that could be used to directly write raw HTML, but access through the DOM API is preferred

```
document.write("It is: " + new Date());
```



## 2.29 Other Useful Objects

- The **'String'** object has useful methods for accessing and manipulating the characters in a String

- ◊ All String literals (declared with just quotes) are objects even without the 'new' keyword

```
var name = "Susan"; // same as new String("Susan")
var secondChar = name.charAt(1); // position starts at 0
var upper = name.toUpperCase();
```

- The **'Math'** object has various useful math functions like 'random', 'sin', 'cos', 'tan', 'ceil', 'floor', 'sqrt', 'abs' (absolute value), etc.

```
var random = Math.floor(Math.random() * 11);
```





## 2.30 Browser Object Detection

- One issue with writing JavaScript is dealing with the potential that an older browser will not recognize the code you use
  - ◇ This is becoming less of an issue but still may be a concern in some situations
- One way to deal with this is to use "object detection" which tests to see if an object, or a method on that object, exists before calling code that will use the object
  - ◇ This allows you to provide an alternate path in the code if the browser doesn't understand the object or method you are trying to use
- To do this, you use an 'if' statement where the condition is the name of the object or method you want to detect
  - ◇ If a browser doesn't understand an object or method, it will return **false**
  - ◇ You only use a method name and don't pass parameters to a method, as you are not calling the method in the conditional test

```
if (now.toLocaleDateString) {
 // This could cause errors in older browsers
 var date = now.toLocaleDateString();
} else {
 alert("Your browser does not give a locale date/time");
}
```



## 2.31 The eval Function

- The `eval()` function compiles and executes an argument which can be a JavaScript expression, variable, statement, or sequence of statements
- May pose some security threats due to possible harmful code injection attacks
- Example:

```
eval ("1 + 999"); // will return 1000
```



## 2.32 Enforcing the Strict Mode

- By default, JavaScript runs in the so-called "sloppy programming mode" where JavaScript interpreter is very lenient to user code ambiguities and makes a lot of assumptions (sometimes very wrong) about the programmer's intent
- You can enable Strict Mode to make JavaScript more stringent and enabling more warnings
- To enable Strict Mode, type the following:  
`'use strict';`
- as the first line in your JavaScript file or a `<script>` tag



### 2.33 Summary

- JavaScript variables are not declared with a type and can hold a value of any type
- JavaScript uses operators that are similar to other programming languages
- The flow control statements in JavaScript are also similar to other languages, although the 'for/in' statement is unique
- There are several built-in object types in JavaScript that can provide useful utility functions
- Using "browser object detection" to determine if a browser recognizes an object, before trying to use it in the code, makes it easier to write cross-browser compatible code than trying to detect the exact type of browser

## Chapter 3 - JavaScript DOM API

---

### *Objectives*

Key objectives of this chapter

- Introduce the DOM, or Document Object Model
- Understand major DOM elements
- Learn how to manipulate document element properties



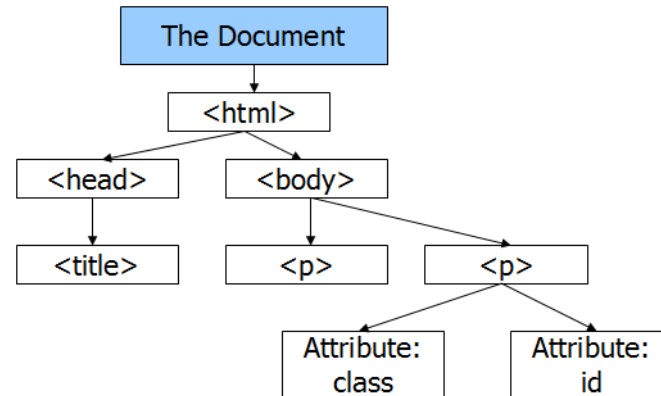
### 3.1 What is DOM?

- DOM stands for Document Object Model. According to this model, an XML document is loaded into memory as a tree like hierarchy of elements.
- DOM has dual purpose in a web browser:
  - ◇ First, the XHTML or plain HTML document for a page is loaded as a DOM tree.
  - ◇ Second, you can construct an arbitrary XML document using the DOM API. We will discuss this in a separate chapter.
- Using the JavaScript DOM API, we can read data from and manipulate an XML document. For example:
  - ◇ Look up an element.
  - ◇ Create or remove elements.
  - ◇ Change the attributes of an element.
- Simply put, this API allows us to alter the contents of a page that is currently open in the browser.
  - ◇ Before DOM, one could only use `document.write()` to change the contents. As we will see shortly, the DOM API allows for much more powerful page alteration.



## 3.2 Element Hierarchy

```
<html>
<head>
<title>Hello</title>
</head>
<body>
<p>Hello world.</p>
<p class="w1" id="p1">
How are you?</p>
</body>
</html>
```





### 3.3 DOM Standardization

- The DOM API is specified by W3C.
- It has many language bindings – C#, Java, C++ and JavaScript.
- Most browsers conform to this standard to a great extent. There are minor deviations at times.
  - ◇ These browser specific deviations are clearly labeled in this chapter. Here, **Mozilla** stands for FireFox and Netscape. And, **IE** stands for Internet Explorer.
- Find DOM API documentation:
  - ◇ W3C spec: <http://www.w3.org/TR/REC-DOM-Level-1/>
  - ◇ IE has slightly different DOM API for a web page and generic XML:
    - XHTML DOM: <http://msdn2.microsoft.com/en-us/library/ms533050%28vs.85%29.aspx>
    - DOM for XML: [http://msdn2.microsoft.com/en-us/library/ms764730\(VS.85\).aspx](http://msdn2.microsoft.com/en-us/library/ms764730(VS.85).aspx)
  - ◇ Mozilla: [http://developer.mozilla.org/en/docs/Gecko\\_DOM\\_Reference](http://developer.mozilla.org/en/docs/Gecko_DOM_Reference)





### 3.4 The Document Object

- Represents the root element of an XML (including HTML) document.
- The HTML document currently loaded by the browser is available to JavaScript as the global variable named **document**.
  - ◇ You can also load any other XML document using a URL or create a brand new document from scratch.
- Important properties of a document object:
  - ◇ `documentElement` – returns the direct child element of the document (which is usually `<html>`).
  - ◇ `firstChild` – returns the first child of the document. See below on how this may differ from `documentElement`.
  - ◇ `body` – returns the `<body>` element in the document.
  - ◇ `cookie` – returns a comma separated list of cookies.
  - ◇ `forms`, `anchors` and `images` – These properties return an array of all `<form>`, `<a>` and `<img>` elements in the document.
  - ◇ `location` or `URL` – Both properties return the full URL of the document.



## 3.5 The Document Object

- Important methods:
  - ◇ getElementById(id) – Returns an element object given its unique identifier. One of the most commonly used methods.
  - ◇ getElementsByName(name) – returns an array of elements with the given name.
    - Element names do not have to be unique.
    - This function is useful when you wish to get a list of similar elements and do something with them. Such as make all error messages blink.
    - For IE, only works for BUTTON, TEXTAREA, APPLET, SELECT, FORM, FRAME, IFRAME, IMG, A, INPUT, OBJECT, MAP, PARAM and META elements.
  - ◇ getElementsByTagName(tag) – returns an array of elements give their tag name. Use the tag name as "p", "div" etc., without the angle brackets.



## 3.6 The Document Object

- Important methods:
  - ◇ `createElement(tag)` – Given a tag name (such as "p" or "img"), returns a new element. To render this element, it must be added as a child of an existing element.
  - ◇ `createTextNode(text)` – Creates a blob of text that can be appended as body content of any other element.
  - ◇ `createAttribute(name)` – Returns a new attribute node object. It must be associated with an element later for it to have any effect. We will learn how to set attribute of an element later.



### 3.7 Nodes and Elements

- Each component of a DOM document is called a Node. This includes:
  - ◇ Comments
  - ◇ DOCTYPE declaration
  - ◇ Elements – These are actual tags such as <p> and <form>.
  - ◇ Attributes of an element are also represented as a Node.
- So, all Elements are also Nodes but not vice versa.
  - ◇ Element extends Node or inherits from Node in the Object Oriented parlance.
- In this chapter, we will focus on the properties, methods and events of the Elements (which will include the behavior available from the Node level).
- All elements have a common set of behavior (properties, methods and events). We will discuss them here.
  - ◇ Some complex elements, such as <form> and <table>, extend the Element and add additional behavior. They will be discussed in a separate chapter.



## 3.8 The Element Object

- An Element object is the core of all types of elements in a page.
  - ◇ Such as a form, img etc.
- An element object can be obtained by calling the document object's getElementById, getElementsByName or getElementsByTagName methods.
- Important properties:
  - ◇ id – The unique identifier.
  - ◇ childNodes – Returns an array of children elements.
  - ◇ innerHTML – The body content text of the element. This text may contain other element tags. System will automatically create these child elements.
  - ◇ firstChild – returns the first child element.
  - ◇ nextSibling – returns the next element that has the same parent as this element. You can iterate through all children by using firstChild in conjunction with nextSibling. Alternatively, just call childNodes.
  - ◇ parentNode – The parent node.



## 3.9 The Element Object

- Important properties:
  - ◇ name – The "name" attribute of the element. Non-unique. Not all elements can have name.
  - ◇ nodeName – Name of the node, such as FORM, P, IMG etc.
    - Returned in upper case by IE and FireFox if "text/html" MIME type is used. For "application/xhtml+xml", lower case names are returned.
  - ◇ nodeType – Integer value indicating the nature of the node. See list below.
  - ◇ textContent (Mozilla), innerText (IE) – The body text of an element. The difference from innerHTML is that, tags for all child elements are stripped out.
  - ◇ ownerDocument – The document object that eventually owns this element.
  - ◇ style – Returns the style object. More on style object later.
  - ◇ className – The name of the style class of the element.
  - ◇ parentNode – Returns the node this element's parent.



## 3.10 The Element Object

- Important methods:
  - ◇ `appendChild(element)` – Adds a child element at the end of the list of children. If the element being added is already in the DOM document, it will be first removed from its current parent. Same goes for `insertBefore`.
  - ◇ `insertBefore(element, beforeElement)` – Adds a new child element before the child element `beforeElement`. Note: Element `beforeElement` must be a child of the current element for which `insertBefore` is called. As a result, a common way to call `insertBefore` is:
    - `sibling1.parentNode.insertBefore(sibling2, sibling1)`. `sibling2` will be added before `sibling1`.
  - ◇ There is **no** `insertAfter()` function. Use this instead:
    - `sibling1.parentNode.insertBefore(sibling2, sibling1.nextSibling)`. `sibling2` will be added after `sibling1`.
  - ◇ `getElementsByTagName(tag)` – Returns an array of children elements that match the tag name (such as "p" or "img").



## 3.11 The Element Object

- Important methods:
  - ◇ `getAttribute(name)` – Returns the value of an attribute given its name. IE has unusual behavior for some properties. For example, use `getAttribute("className")` to get the "class" attribute.
    - You can set extended attributes in HTML: `<div foo="bar"/>`. `div.getAttribute("foo")` will return "bar".
  - ◇ `setAttribute(name, value)` – Sets the value of an attribute. You can set extended attributes not in XHTML spec: `div.setAttribute("foo", "bar")`.
  - ◇ `setAttributeNode(attribute)` – Sets an attribute node object. Attribute nodes are created using the `createAttribute` method of document.
  - ◇ `getAttributeNode(name)` – returns the attribute node object given its name.
  - ◇ `removeChild(child)` – Removes a child element. The method returns the removed element which stays in memory and can be added to another parent.
  - ◇ `replaceChild(newChild, oldChild)` – The element replaces one child element with another.





## 3.12 The Element Object

- Important methods:
  - ◇ `removeAttribute(name)` – Deletes the attribute of given name. The browser will use a default value for the attribute if applicable.
  - ◇ `hasChildNodes()` – Returns a true or false indicating if there is any child element.
  - ◇ `hasAttributes()` – Returns true or false indicating if the element has any attribute.
  - ◇ `cloneNode(deepCopy)` – Returns a replica of this element. If `deepCopy` is true, all attributes and children elements are also copied. The newly created element must be added to a parent element to be visible. If you need to create many elements that have the same style and list of children, it will be faster to create a template element first and simply clone it many times.



### 3.13 Element Event Handlers

- Nearly all elements support these event handler properties:
  - ◇ onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup and onresize.
- Set the value of these properties to a function object.
  - ◇ There are three ways to do this. See below.
- Event handling will be covered in more detail in a separate chapter.



### 3.14 The window Object

- The JavaScript global variable – window – represents the actual window that renders the DOM document.
  - ◇ Every window contains one document object.
- A new window is created using the open method of the window object.
- Main properties:
  - ◇ document – Returns the DOM document object.
  - ◇ frames – Returns an array of Frame objects.
  - ◇ history – Returns the browser's History object. Discussed later.
  - ◇ location – The URL of the page.
  - ◇ opener – Returns the Window object that opened this window. May be null if there are no opener.



### 3.15 The window Object

- Key methods:
  - ◇ open(URL, name, featureList) – returns a new window object and opens it.
    - URL – The URL to show in the window. If empty string, an empty document is shown.
    - name – The unique name of the window. If a window by that name is already opened, system loads in the new URL and returns that window object.
    - featureList – A comma separate list of name=value pairs. See list of common features below.
    - Example: `var w = window.open("http://abc.com", "news", "menubar=yes,location=yes,resizable=no");`



## 3.16 The window Object

### ■ Events

- ◇ onload – Fired after all elements of a document has been loaded from the network or cache.
- ◇ onunload – Fired, before a page is unloaded and the next page is loaded.
- ◇ onresize – Fired after the window is resized.
- ◇ onfocus/onblur – Fired when keyboard focus is gained or lost by the window.



### 3.17 The Frame Object

- Represents a <frame> element within a <frameset>
- Key properties:
  - ◇ `contentDocument` – The document rendered in the frame. IE may not support this. Some browsers may use the property name "document".
  - ◇ `src` – The URL of the frame.
  - ◇ `noresize` – if true, frame can not be resized.



### 3.18 The History Object

- Represents a browser's history.
- Obtained using the history property of the window object.
- Properties:
  - ◇ length – Number of items in the history. Read only.
- Methods:
  - ◇ back() – Goes to the previous page.
  - ◇ forward() – Goes to the next page.
  - ◇ go(index) – Goes to a page relative to the current page. For example, go(-1) is same as back() and go(1) is same as forward().
  - ◇ go(URL) – Goes to a page in the history that matches the given URL.



### 3.19 Summary

- DOM is a JavaScript API that allows script code to access and modify elements of a web page
- DOM constructs a hierarchy of elements contained on the page
- The 'document' variable represents the root of this hierarchy
- Elements in a document share some common properties and behaviors
- You can also access information about the browser window



## Chapter 4 - JavaScript Functions

---

### *Objectives*

Key objectives of this chapter

- Declaring JavaScript functions
- Function arguments and return values
- Local and global variables



## 4.1 Functions Defined

- A JavaScript function is a named block of code that can be called from other JavaScript code by using the function name
- Functions can optionally be defined to take arguments with values passed in by the code calling the function
- Functions can optionally return a value from the function that can be used by the code that called the function
- In JavaScript, functions are "first-class objects" that are treated as any other JavaScript objects
  - ◇ Functions can be passed to and returned from other functions (as an anonymous or *lambda* function); more on that in a later module



## 4.2 Declaring Functions

- Function declarations have specific syntax in JavaScript although it is somewhat simpler than other programming languages
- Functions are declared by using the '**function**' keyword, the name of the function, a set of parenthesis for arguments (possibly empty), and a set of curly brackets '{ }' around the statements that make up the body of the function

```
function myFunction() {
 // function body
 alert("Hello from inside the function!");
} // end of function declaration
```

◊ Note: there

- To call the function you simply use the function name along with any needed arguments in parenthesis
  - ◊ The parenthesis still need to be there even if empty

```
// call the function
myFunction();
```



### 4.3 Function Arguments

- If a function is declared to accept arguments, the names of the arguments are given in the parenthesis of the function declaration and separated by commas
  - ◇ These become variables available in the function whose value is initialized by whatever value is passed in by the code calling the function
  - ◇ Just like regular JavaScript variables, function arguments do not have a type declared for them and can accept any type

```
function sayHello(firstName, lastName) {
 alert("Hello " + firstName + " " + lastName + "!");
}
```



## 4.4 More on Function Arguments

- Missing arguments are treated in the body of the function as undefined
- Objects (covered later in the course) are passed to functions by reference
- When calling a function and passing in argument values, be careful the order the arguments are passed in to the function and the type that might be used

```
sayHello("Bob", "Smith");
sayHello(4, 12); // still legal, probably not as intended
```



## 4.5 Return Values

- Functions can return a single value back to where the function was called
  - ◇ This is done by using the '**return**' statement along with the value to be returned in the body of the function
  - ◇ There is nothing in the function declaration that indicates if a function returns a value, unlike some other programming languages
  - ◇ When a '**return**' statement is executed, the body of the function stops executing and returns the value

```
function multiply(a, b) {
 return a * b;
}
```

- The returned value is often stored in a variable where the function was called from

```
var result = multiply(2, 14);
// value of result variable is 28 after calling function
```

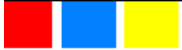


## 4.6 Multiple Return Values in ECMAScript 6

- The upcoming release of ECMAScript 6 (ES6 "Harmony") plans to add multiple function return values
- The proposed syntax is:
  - ◇ `return {var1, var2};` for a two-value return
  - ◇ You can then process the multi-value returns as objects
- Don't confuse this proposed syntax with returning an object in ECMAScript 5-compliant JavaScript engines, e.g.:

```
function foo () {
 return {i : 123, j: "OK"};
}
```

would return an Object {i: 123, j: "OK"}

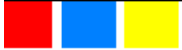


## 4.7 Optional Default Parameter Values

- Current version of JavaScript (ECMAScript 5) does not support optional function arguments or parameters
- **Note:** ECMAScript 6 spec includes provisions for default values
  - ◇ The proposed syntax is:

```
function f (x1, x2="Some optional value") {...}
```





## 4.8 Emulating Optional Default Parameter Values

- You can use the following pattern to emulate default values for function's arguments using the logical *or* operator: `||`

```
function foo(x) {
 x = x || -1;
// if x is missing or is undefined, it will be assigned the default value of -1;
// Now you can use x's real value, or its default value of -1
```



## 4.9 Anonymous Function Expressions

- JavaScript has another way to define a function called **anonymous function expressions**
- An anonymous function expression produces a function object that is assigned to a variable, which becomes the name of the function
  - ◇ For example:

```
var add = function (x, y) { return x + y };
add(6, 4); //will return 10
```



## 4.10 Functions as a Way to Create Private Scope

- A JavaScript function, when called, creates a new scope or execution context for your code running inside the function
- Variables defined within a function are only accessible from inside it and are not visible to the outside context
- So, in JavaScript, functions offer you a mechanism to establish a private scope



## 4.11 Linking Functions to Page Elements

- Functions can be called when certain events happen on page elements like clicking a button
  - ◊ This is done by having JavaScript code to call the function tied to an HTML element event attribute

```
<html><head><script>
function myFunction()
{
 alert("Hello World!");
}
</script></head>
<body>
<button onclick="myFunction()">Try it</button>
</body></html>
```

- Some of the most common events JavaScript functions are linked to are:
  - ◊ The 'onload' event on the 'window' object when the page is loaded
  - ◊ The 'onmousedown', 'onmouseup', or 'onclick' events of a button
  - ◊ The 'onchange' event of form input elements to trigger validation code
  - ◊ The 'onmouseover' and 'onmouseout' events of various elements



## 4.12 Local and Global Variables

- Any variable that is "local" to the function will go out of scope when the function is finished executing
  - ◇ This means the value will no longer be accessible
- Function arguments are automatically local variables
- Variables declared with the '**var**' keyword inside the body of the function are also local variables
- Any variables that are declared within a function and initialized without the '**var**' keyword are GLOBAL
  - ◇ These can be modified from within the function and still accessed once the function is finished executing
- Variables declared outside of a function are always global



### 4.13 Local and Global Variables

- The example code below shows global and local variables
  - ◊ '**localVariable**' is not accessible outside of the function

```
<script>
var globalVar = 45; // global variable

function myFunction(localParameter) {
 var localVariable = 12; // local to function
 alsoGlobal = 15; // global variable within function
 var answer = localParameter + localVariable +
 alsoGlobal;
 alert("Answer: " + answer);
}

myFunction(34) // shows alert with '61'
alert(alsoGlobal); // legal because the variable is global
alert(localVariable); // doesn't work outside function

</script>
```



## 4.14 Declaring Object Methods

- In JavaScript a "method" is just the terminology for a function that is declared on a specific object and is not a "global" function declared at the root of the script
  - ◇ The **'function'** keyword is still used and the difference is the way it is declared
  - ◇ The method parameters (if any) are defined immediately after the **'function'** keyword
- A method is a named property of an object that is initialized as a function
  - ◇ Within a method for an object, other properties of the object can be accessed by using the **'this'** keyword

```
person = new Object();
person.firstName = "John";
person.setFirstName = function(newName) {
 this.firstName = newName;
};
```

- You must call the method by using the 'dot' notation between the name of the object and the method name along with any arguments

```
person.setFirstName("Jonathan");
```



### 4.15 The arguments Parameter

- When a function is invoked, it is supplied a special parameter called **arguments** accessible inside the function
- It is an array-like structure which has the **length** parameter containing the number of parameters actually passed to the function
- *arguments* can be used to access parameters using indexes
- Missing parameters or any extra parameters are treated as *undefined*





## 4.16 Example of Using arguments Parameter

- In code below we are using the *console* object supported by most modern JavaScript engines to output content to the JavaScript browser console

```
function foo (x, y, z) {
 console.log ("The number of arguments actually passed in : " + arguments.length);
 console.log (arguments[0]);
 console.log (arguments[1]);
 console.log (arguments[2]);
 console.log (arguments[3]); // extra parameter !
}
```

```
foo (777, 'a');
```

OUTPUT:

The number of arguments actually passed in : 2

777 // arguments[0]

a // arguments[1]

undefined // arguments[2]

undefined // arguments[3]



## 4.17 Summary

- JavaScript functions are declared with the '**function**' keyword
- Function arguments are declared within parenthesis but do not have a declared type similar to "normal" JavaScript variables
- Functions can return values but nothing about the function declaration indicates if it does
- Function arguments and variables declared within the function with the '**var**' keyword are local to the function and not accessible outside the function
- Objects can have "methods" which are functions declared as object properties

## Chapter 5 - JavaScript Arrays

---

### *Objectives*

Key objectives of this chapter

- Define JavaScript arrays
- How to perform various actions with arrays
- Useful methods for arrays



## 5.1 Arrays Defined

- An array is an object that is implemented as a map (dictionary / hash table) where indexes are numbers converted into strings which are used as keys to retrieve the values
  - ◇ In contrast, in many programming languages, arrays hold a linear sequence of data of the same type
- An array can hold many values under a single name, and you can access the values by referring to an index number
  - ◇ JavaScript arrays use a '0' as the index of the first element in the array
- Just like JavaScript variables that store a single value, JavaScript arrays are not declared with a single data type
  - ◇ This means that different elements of the array can hold values of different types
- Arrays are efficient for holding sparse data (where some elements are missing)



## 5.2 Creating an Array

- There are three different ways to create and initialize an array

- ◊ Create empty array and initialize each element

```
var myCars = new Array();
myCars[0] = "Saab";
myCars[1] = "Volvo";
myCars[2] = "BMW";
```

- ◊ Supply the initial values of the array elements as parameters to the constructor

```
var myCars = new Array("Saab", "Volvo", "BMW");
```

- ◊ Use square brackets with an array literal

```
var myCars = ["Saab", "Volvo", "BMW"];
```

- You can also pass a single number to the Array constructor to indicate how many elements to create the array with

- ◊ If you do this, the value of the elements will start as 'undefined'
- ◊ Unlike other languages, you can add elements to an array at any time and do not need to work with a "fixed" number of elements

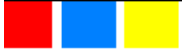
```
var myCars = new Array(3);
```



### 5.3 The length Array Member

- An array object has a '**length**' property that (surprise ! ) does not return the number of elements in the array (as is the case in many other programming languages)
- The *length* property contains a number which is the highest integer subscript in the array + 1
- Example of using the *length* property:

```
var a = [];
a[4]= 4;
a [1001] = 'Whatever ..';
a.length; // 1002 = 1001 + 1
```



## 5.4 Traversing an Array

- You can use the usual **for...in** syntax:

```
var array1 = ["John Smith", "Jimmy Dean"];
for (i in array1) {
 log("Item " + i + ": " + array1[i]);
}
```

- Or use the **length** property:

```
for (var i = 0; i < array1.length; ++i) {
 log("Item " + i + ": " + array1[i]);
}
```



## 5.5 Appending to an Array

- The correct way to append an element is to use the **push()** method

```
var a2 = ["John Smith", "Jimmy Dean", "Meg Ryan"];
a2.push("Iron Man", "Jackie Chan");
//Adds two elements to a2 and returns the current length.
```

- Alternatively, just add a new property to the object with the right index value

```
a2[3] = "Iron Man";
```

- ◇ Generally, the formula is: `myArray[myArray.length] = ...;`

- If you add a new element with an index greater than what the next index would normally be, any elements "between" the current elements and the element being added will be uninitialized

```
a2[5] = "Spider Man";
// a2.length will return 6. However,
// a2[4] will return undefined.
```





## 5.6 Deleting Elements

- You can use the **delete** keyword to remove an element, but that doesn't change the length of the array
  - ◊ This simply removes the value stored by that element

```
var a2 = ["John Smith", "Jimmy Dean", "Meg Ryan"];
delete a2[2];
// a2.length still returns 3 but a2[2] returns undefined
```

- The correct way to delete elements is to use the **splice()** method of the array object
  - ◊ splice(index to start deleting from, number of elements to delete)

```
var a2 = ["John Smith", "Jimmy Dean", "Meg Ryan",
 "Spider Man"];
a2.splice(1, 2);
// This will delete "Jimmy Dean" and "Meg Ryan"
// a2.length will now return 2
```



## 5.7 Inserting Elements

- The **splice()** method is also used to insert elements
  - ◇ **splice(index to start, number of elements to delete, list of new elements to insert)**

```
var a2 = ["John Smith", "Jimmy Dean", "Meg Ryan",
 "Spider Man"];
a2.splice(1, 2, "Iron Man", "Hulk", "Jackie Chan");
// This will delete "Jimmy Dean" and "Meg Ryan" and replace
// them with "Iron Man", "Hulk" and "Jackie Chan".
```

- ◇ To insert new elements without deleting existing elements of the array, use **0** for the number of elements to delete (the second splice's argument)

```
a2.splice(1, 0, "Iron Man", "Hulk", "Jackie Chan");

// the above command will produce a 6-element array (inserted elements
// are shown in bold):
["John Smith", "Iron Man", "Hulk", "Jackie Chan", "Jimmy Dean", "Meg
Ryan", "Spider Man"]
```



## 5.8 Other Array Methods

```
var a1=["Jackie Chan", "Iron Man"];
```

```
var a2=["John Smith","Jimmy Dean","Meg Ryan","Spider Man"];
```

- **pop()** – Removes the last element and returns it

```
a2.pop() // Returns "Spider Man" and deletes that element
```

- **shift()** – Removes the first element and returns it

```
a2.shift() // Removes "John Smith" and returns it
```

- **unshift()** - Adds new elements to the beginning of the array and returns the new length

```
newLength = a1.unshift("Meg Ryan", "Spider Man");
```

```
// Adds new elements to the start of the array and returns 4
```

- **sort()** - Sorts the elements of an array, taking a sorting function as an optional parameter
  - ◊ Default sort order is ascending alphabetical

```
a4 = [45, 12, 26, -34];
```

```
a4.sort(function(a,b) {return a-b}); // [-34,12,26,45]
```

- **reverse()** - Reverses the elements of an array
- **indexOf()** & **lastIndexOf()** - Searches from the start or end of the array for the given item and returns the position
  - ◊ An optional second parameter can give the position to start from



## 5.9 Other Array Methods

```
var a1=["Jackie Chan", "Iron Man"];
var a2=["John Smith","Jimmy Dean","Meg Ryan","Spider Man"];
```

- **concat(array or elements)** – Creates a new array by adding elements from another array

```
//Contents of a1 and a2 are combined and returned
```

```
a3 = a1.concat(a2);
```

```
//Elements of a1 and two new elements are combined and returned
```

```
a3 = a1.concat("Meg Ryan", "Spider Man");
```

- **slice(begin, stop)** – Returns a subset of the array starting with begin index and up to, but not including the stop index

```
a3 = a2.slice(1, 3)
```

```
// Returns an array containing "Jimmy Dean" and "Meg Ryan"
```

- **toString()** - Converts an array to a comma-separated String of the elements and returns the result
- **join()** - Joins the elements of an array into a String with an optional parameter for the separator between elements



## 5.10 Accessing Objects as Arrays

- You can access the properties of a JavaScript object with array syntax instead of the 'dot' syntax by using the property name as the array index

```
person = {firstname:"John", lastname:"Doe", age:50, eyecolor:"blue"};
alert(person["firstname"]);
```

- This is used in the 'for..in' loop where the temporary variable in the loop actually loops through the names of the object properties

```
var txt = "";
var person = {fname:"John", lname:"Doe", age:25};

for (var x in person)
{
 txt = txt + person[x];
}
// txt is 'JohnDoe25'
```



## 5.11 Summary

- Arrays in JavaScript do not have a declared type or size and are very dynamic
- An array has a 'length' property that can be important when working with the array
- There are various array methods that perform various actions on the array

## Chapter 6 - Advanced Objects and Functionality in JavaScript

---

### *Objectives*

Key objectives of this chapter

- Using JavaScript as an Object Oriented language
- Object Constructor functions
- JavaScript closures
- Prototype property
- Techniques for establishing inheritance in objects



## 6.1 Basic Objects

- A basic object is a collection of properties. Each property has a name and a value. In a way, an object is like a dictionary or a hash map
- A property can be of any type (a string, date, time, number or a function)
- There are three ways to create an object in JavaScript:
  - ◇ Using an object literal
  - ◇ Using a constructor object
  - ◇ Using the `Object.create()` method (reviewed after we cover inheritance)

- Object literal:

```
var person = {firstName: "John", lastName: "Smith"};
var employee = {name: "John Smith",
 address: {street: "...", city: "..."}};
```

- Here we use the `Object` constructor to create an object:

```
var person = new Object();
person.firstName = "John";
person.lastname = "Smith";
```



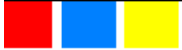


## 6.2 Constructor Function

- The problem with a basic object is that it has no constructor function that can initialize the object. In addition, a literal based object can not have multiple instances created using the "new" keyword
- A constructor function removes these restrictions and makes JavaScript more object oriented. Example constructor function:

```
function Person(fname, lname) {
 //Access object property using "this" keyword
 this.firstName = fname;
 this.lastName = lname;
 this.display = function () {
 alert(this.firstName + " " + this.lastName);
 }
}
```

- All properties for a function object must be defined with the "this." prefix
- Although not required, the naming convention is to capitalize the first letter of any function which defines and initializes a new object

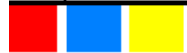


### 6.3 More on the Constructor Function

- New instances (objects) of the constructor function are created using the "new" keyword

```
var p1 = new Person("Jimmy", "Dean");
p1.display();
```

- The constructor functions (constructors) can be viewed as factories for objects created when invoked via the *new* operator
- Semantically, JavaScript constructor functions are similar to classes in other languages



## 6.4 Object Properties

- A property of an object can be accessed in one of two ways:
  - ◇ `obj.property` – The dot notation. This is similar to C++ or Java
  - ◇ `obj["property"]` – The associative array notation (a.k.a. *subscript* notation)

```
var p1 = new Person("Jimmy", "Dean");
alert(p1.firstName);
alert(p1["lastName"]);
p1["lastName"] = "Latimer";
p1.display();
```

- Arbitrary properties can be added to an object at any time

```
p1.salary = 40000.00; //A new property is added
alert("Salary is: " + p1.salary);
```

- Accessing an undefined property returns "undefined"

```
if (p1.department !== undefined) { // or (p1.department)
 alert("Department is: " + p1.department);
} else {
 alert("Department is not found"); }
}
```

- Trying to use an undefined property throws error

```
p1.department.name = "Engineering"; //Throws TypeError
```



## 6.5 Deleting a Property

- The true dynamic nature of JavaScript can be illustrated by the fact that you can delete a property of an object!
- To delete an existing property of an object, use the *delete* keyword

`delete obj.propName`

- ◊ Both dot and subscript notations are supported
- The *delete* keyword deletes both the value of the property and the property itself; the associated memory will be automatically reclaimed by the garbage collector



## 6.6 The instanceof Operator

- To find out if a variable is an instance of a particular object, use the *instanceof* operator which evaluates to a boolean
- Here is how it works:

```
function Person () {};//declare some dummy object with a constructor
var p = new Person(); //create an instance of it
p instanceof Person; // returns true
p instanceof Animal;
// We have not defined Animal, so get an
// Uncaught ReferenceError: Animal is not defined
var o = new Object();// Use the Object built-in constructor
o instanceof Person; // false
p instanceof Object; // true
```



## 6.7 Object Properties

- You can enumerate through all object properties using the for...in syntax:

```
var p1 = new Person("Jimmy", "Dean");
for (propertyName in p1) {
 alert("Property: " + propertyName);
 alert(p1[propertyName]);
}
```

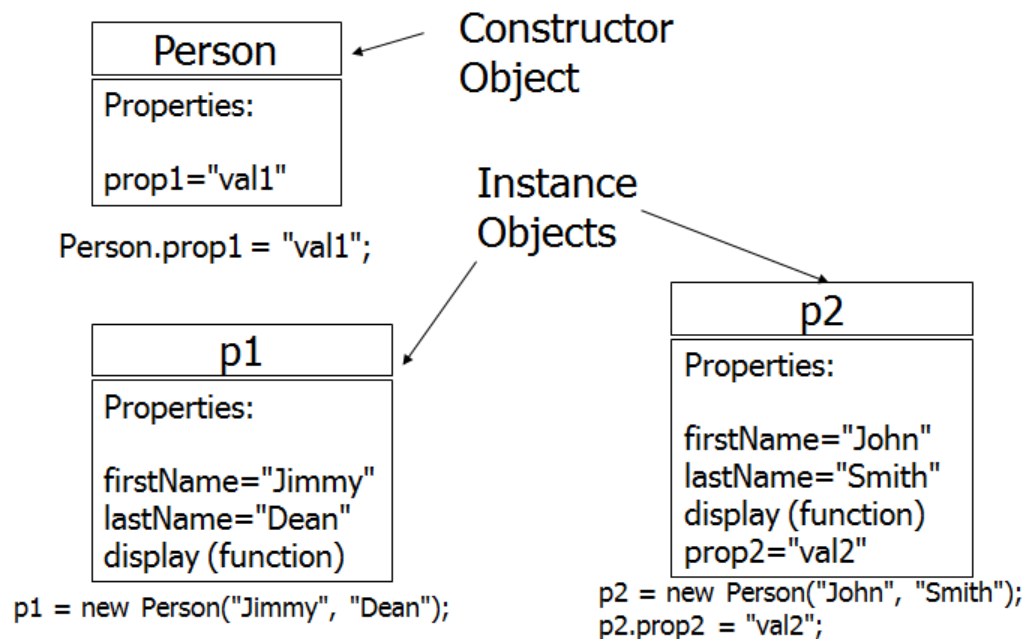
- Every property has a data type that can be retrieved using the typeof operator. Possible values are: "number", "string", "object", "function" and undefined

```
for (propertyName in p1) {
 if (typeof p1[propertyName] !== "function") {
 alert("Property: " + propertyName);
 alert(p1[propertyName]);
 }
}
```



## 6.8 Constructor and Instance Objects

- It is important to note that the constructor function, like any other JavaScript function, is an object.
- Only one instance of the constructor object will exist. With the use of the new keyword, you can create many instance objects
- Properties that are added outside of the constructor function are not shared by the instances





## 6.9 Constructor Level Properties

- In Java or C++ you can have class level methods and member variables. They are called 'static'. You can do the same in JavaScript using properties of the constructor function
- Constructor level properties can be data or function. Just like for static methods in Java, a function property does not have visibility into the instance level properties (such as firstName)
- Use constructor level properties to:
  - ◇ Define constant values
  - ◇ Define functions that do not operate on instance level data
  - ◇ Define global variables and functions in a unique namespace

```
function MyDate () { // constructor function
...
}
MyDate.JANUARY=1; //Constant
MyDate.FEBRUARY=2;
...
MyDate.isGreater = function(date1, date2){ ... }
//Calling the function
if (MyDate.isGreater(date1, date2)) { ... };
```





## 6.10 Namespace

- Use namespace to avoid any conflict of names for global variables and functions
  - ◊ A namespace is a prefix for variable or function names that make them virtually unique
- It is recommended that you use Java or C# style dot separated namespace, such as "com.webage.util"
- In JavaScript namespaces are created using properties of a constructor
- The namespace properties can also be added to the window object. The advantage of this option is that you can check for the existence of a property to make sure you don't overwrite it. Example:

```
function createNamespace(ns) {
 var nsParts = ns.split(".");
 var root = window;
 for(var i = 0; i < nsParts.length; i++) {
 if(root[nsParts[i]] == undefined) {
 root[nsParts[i]] = new Object();
 }
 root = root[nsParts[i]];
 }
}
createNamespace("com.webage.util");
com.webage.util.doTest = function doTest() {
 alert("Test worked");
}
```



## 6.11 Functions are First-Class Objects

- In JavaScript, functions are "first-class objects"
  - ◇ They co-exist with, and can be treated like, any other JavaScript object
- In particular, JavaScript functions enjoy the same capabilities as objects:
  - ◇ They can be created via literals
  - ◇ They can be assigned to variables, array entries, or properties of other objects
  - ◇ They can be passed as function arguments
  - ◇ They can be returned as values from functions
  - ◇ They can possess properties that can be dynamically created and assigned
- One example of this is passing a function that will be used to compare two values as a parameter to the 'sort' function of an Array

```
var values = new Array(...);
values.sort(function(value1, value2){
 return value2 - value1; });
```



## 6.12 Closures

- A closure is the scope created when a function is declared that allows the function to access and manipulate variables that are external to that function
  - ◇ Closures allow a function to access all the variables and other functions that are in scope when the function is declared
- The closure for a function creates a protected scope that can preserve local variable and function declarations that would normally have gone out of scope and would have been deallocated
  - ◇ The closure example below has an anonymous function that is assigned to the 'sayAlert' variable and then returned and executed from outside the scope of the 'sayHello2' function which creates a closure and protects access to the local variable 'text'

```
function sayHello2(name) {
 var text = 'Hello ' + name; // local variable
 var sayAlert = function() { alert(text); }
 return sayAlert; }
var say2 = sayHello2('Jane');
say2();
```

- Declaring one function inside another function creates a closure that allows the function to access variables that would normally have gone out of scope
  - ◇ A reference to the function also references the closure the function was created in



## 6.13 Closure Examples

- Local variables within a closure are not copied, they are kept by reference and can change value after the function that will use them is declared
  - ◊ In the following example the value of the 'num' variable changes after the inner function 'sayAlert' is declared and shows a '6' in the alert when the function is called again as 'sayNumba'

```
function say6() {
 // Local variable that ends up within closure
 var num = 5;
 var sayAlert = function() { alert(num); }
 num++; // changes after the function is declared
 return sayAlert;
}
var sayNumba = say6();
sayNumba(); // shows alert for 6
```



## 6.14 Closure Examples

- The following code

```
function increment () {
 var i = setUpIncrementor(0); // we start with 0
 i(1);
 i(10);
 i(100);}
function setUpIncrementor(initValue) {
 return function (inc) {
 initValue += inc; //first call: 0+1; then 1+10, etc.
 console.log("initValue in closure: " + initValue);
 return initValue;
 }
}
```

- will produce the following results (we use *console.log* to print results in the JavaScript console in most popular browsers):

```
initValue in closure: 1
initValue in closure: 11
initValue in closure: 111
```



## 6.15 Closure Examples

- If several functions are declared within the same scope they have the same closure and would see changes to the variable values in the closure
  - ◊ In the example below the variable 'num' is within the one closure that contains the three declared functions and all functions will see the current value in the closure when executed

```
function setupSomeGlobals() {
 // Local variable that ends up within closure
 var num = 6;
 // Store some function references as global variables
 gAlertNumber = function() { alert(num); }
 gIncreaseNumber = function() { num++; }
 gSetNumber = function(x) { num = x; }
}
gAlertNumber(); // alerts 6
gIncreaseNumber();
gAlertNumber(); // alerts 7
```



## 6.16 Private Variables with Closures

- One use of closures can be to encapsulate some information as a private variable and limit the scope of such variables
  - ◊ Properties that are declared "normally" in JavaScript do not have this kind of protection
- In this example the 'balance' variable declared within the 'Account' constructor function is part of the closure of the other functions that are defined but is not directly accessible from outside the Account constructor function

```
function Account() {
 var balance = 0;
 this.getBalance = function() { return balance; };
 this.deposit = function(amount) {
 balance += amount; };
}
var myAccount = new Account();
myAccount.getBalance(); // returns 0
myAccount.deposit(100);
myAccount.getBalance(); // returns 100
myAccount.balance // this is undefined (inaccessible)
```



## 6.17 Immediately Invoked Function Expression (IIFE)

- As you see in the above closure examples, they keep connections to the outer variables, which sometimes has negative side-effects
- Immediately Invoked Function Expression (pronounced "iffy") allow you to introduce a new variable scope to prevent it from becoming global using the following JavaScript idiom:

```
(function () {
 var x = ...; // x is not a global variable
})();
;
```

- An IIFE is called immediately after you define it





## 6.18 Prototype

- Every constructor function object implicitly has a property called "prototype". It is an object that is used as a template to create new instance objects from the constructor
- Any property defined in the prototype object is automatically added to an instance object

```
function Person(...) { ...
} // end of constructor function
Person.prototype.prop1 = 'val1';
// This is different than Person.prop1

var p1 = new Person();
var p2 = new Person();

p1.prop1 = 'new val1';
// Now p1.prop1 is 'new val1', but p2.prop1 is still 'val1'
```



## 6.19 Inheritance in JavaScript

- Inheritance is object-oriented code (behavior) reuse that, among many other benefits, helps reduce the cost and time of development
- There are two schools of thought on how best to do inheritance (two inheritance styles / models):
  - ◇ The classical one (Java, C#, C++, etc.)
    - It is a class-based inheritance (a class inherits from another class); objects are instances of classes
  - ◇ Prototype-based (JavaScript; Self, which actually influenced inheritance design of JavaScript)
    - This style of inheritance is a.k.a. prototypal, or instance-based; normally supported by the *delegation* feature in a language, where supported
    - There are no classes at present in JavaScript (although they will be added with ECMAScript 6 "Harmony")
    - Individual objects are cloned from the generic (template) object



## 6.20 The Prototype Chain

- Although, we have said all properties of the prototype are automatically added to an instance, in reality, it is achieved through a mechanism called **prototype chain**
  - ◇ Instances may not physically contain these properties; they inherit them from prototypes, if any
  - ◇ Prototype chaining is fundamental to the way inheritance is implemented in JavaScript
    - The *prototype chain* is somewhat similar to *multiple inheritance* in other languages (e.g. C++)
  - ◇ JavaScript does not impose any limits on how long the prototype chain can be (subject to memory availability in the JavaScript engine)
  - ◇ Prototype-based inheritance helps conserve memory as new objects don't need to carry the baggage of state and methods of the prototype(s)



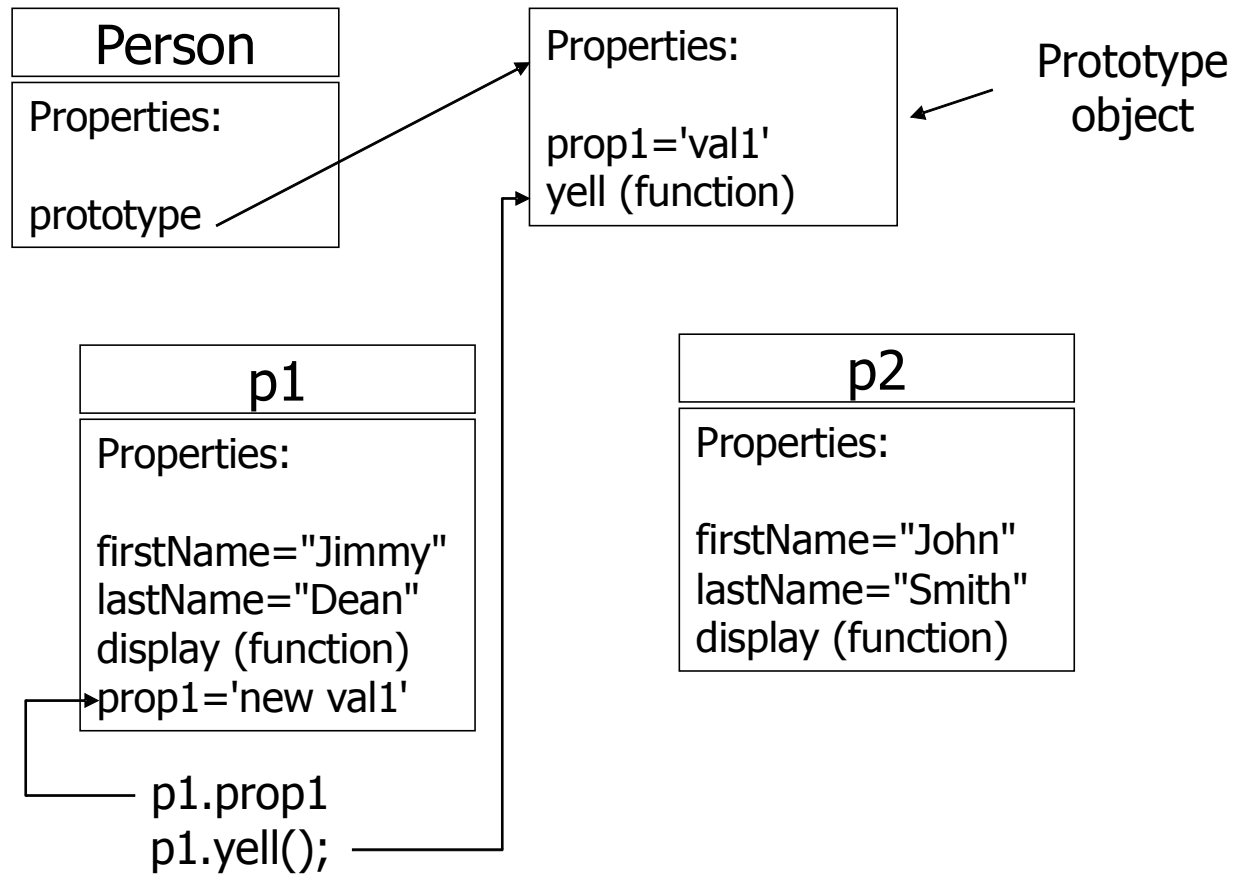
## 6.21 Traversing Prototype Property Hierarchy

- When you read the property of an instance object, the system first tries to locate the property in that instance. If it can not be found, the system tries to locate it in the prototype object. If it is not there, the system looks for it in the prototype of the prototype object, and so on
  - ◇ In the previous example, the reason `p2.prop1` returns 'val1' is because there is no property called `prop1` in `p2`. The system is actually locating the property in the prototype. `p1`, on the other hand, has a property called `prop1` with value 'new val1'
- This also means, new properties can be added to a prototype at any time, and it will show up for any instances that have been already created

```
var p3 = new Person('John', 'Smith');
Person.prototype.yell = function(){alert('what up?');}
p3.yell(); //Due to chaining, the system will
p1.yell(); // find the yell property in the prototype.
```



## 6.22 Prototype Chain





## 6.23 Inheritance Using Prototype

- Because an instance object inherits the properties of a prototype, you can use the mechanism to implement some aspects of inheritance

```
function Person(fname, lname) {
 this.firstName = fname;
 this.lastName = lname;
 this.display = function () { ... }
}
function Customer(fname, lname, c) {
 this.firstName = fname;
 this.lastName = lname;
 this.company = c;
 this.getOrderHistory = function() {
 return ...; //Business logic omitted
 }
}
//Setup a Person instance as prototype of Customer
Customer.prototype = new Person();
var c1 = new Customer('John', 'Smith', 'IBM');
// display will be found in the prototype
c1.display();
// getOrderHistory will be found in the instance itself
c1.getOrderHistory();
```



## 6.24 Extending Inherited Behavior

- There is a problem with the previous example. The Customer constructor function duplicates the behavior that is already in the Person constructor that initializes the firstName and lastName properties.
- Also, how can the Customer constructor implement its own display function that extends the display function that is in the Person constructor.
- The answer to these problems is the "call" method of a function object. Every function object (such as Person, display or getOrderHistory), implicitly have a call() method that can be used to execute the function in the context of any object instance. See an example below.
  - ◊ This will call a function (func), in the context of the object instance obj. Within the function, the "this" keyword will point to the object.

```
func.call(obj, param1, param2...)
```

- An alternative is the apply() function which takes the parameters in an array:

```
func.apply(obj, [param1, param2,...]);
```



## 6.25 Extending Inherited Behavior

- Using the 'call' function, we will call a function in the base constructor in the context of the current object instance

```
function Customer(fname, lname, com) {
 //Calls the Person() function
 Person.call(this, fname, lname);
 this.company = com;
 this.display = function () {
 //Calls the display function of Person
 Customer.prototype.display.call(this);
 log("Company: " + this.company); }
}
Customer.prototype = new Person();
var c1 = new Customer('John', 'Smith', 'IBM');
c1.display(); //Will print firstName, lastName and company
```

- Even if the base constructor did not take any parameters you would want to call it to make sure any properties initialized with default values are initialized for the current object.

```
function Base() {
 this.listOfNumbers = new Array(0); }
function Extension() {
 Base.call(this); // or array isn't initialized }
```





## 6.26 Enhancing Constructors

- You can use the prototype to add new properties to constructors that are defined by third party scripts including the ones defined by the scripting engine (such as Array and Date)
  - ◊ In this example, we add a stop watch capability to the built-in Date constructor

```
Date.prototype.swStart = null;
Date.prototype.startWatch = function () {
 this.swStart = new Date();
}
Date.prototype.endWatch = function () {
 if (this.swStart == null) {
 throw "Watch was never started";
 }
 var now = new Date();
 var elapsedTime = (now.getTime() - this.swStart.getTime());
 swStart = null;
 return elapsedTime;
}
var d1 = new Date();
d1.startWatch();
...
log("Time elapsed (ms): " + d1.endWatch());
```



## 6.27 Improving Constructor Performance

- Defining function properties outside the constructor using prototype can improve performance of the script

```
function Person(fname, lname) {
 this.firstName = fname;
 this.lastName = lname;
}

Person.prototype.display = function () {
 log(this.firstName + " " + this.lastName);
}

var p1 = new Person("Baby", "Jones");
p1.display();
```

- This way, the size of the constructor function becomes smaller and new instances can be created much quicker (especially, if you have many functions).
- Also, individual instance objects do not contain the function property, reducing their memory footprint



## 6.28 Inheritance with `Object.create`

- The *`Object.create()`* method creates a new object with the specified prototype object (and, optionally, properties)
- Simplified syntax:  

```
Object.create(prototypeObject) ;
```

where *`prototypeObject`* is the object which should be the prototype of the object to be created with the *`Object.create`* (factory) method
- First defined in ECMAScript 5.1 (ECMA-262) [<http://www.ecma-international.org/ecma-262/5.1/#sec-15.2.3.5>]



## 6.29 The `hasOwnProperty` Method

- The global *hasOwnProperty()* method returns a boolean (true or false) indicating whether the object owns (has) the specified property as opposed to a property inherited from a prototype

- Example:

```
var o1 = {po1 : 1} // object literal to define o1
var o2 = Object.create(o1); // o1 is prototype for o2
o2.po2 = 2; // create a property of o2 on the fly
o1.hasOwnProperty("po1"); // true
o1.hasOwnProperty("po2"); // false
o2.hasOwnProperty("po2"); true
o2.hasOwnProperty("po1"); false (inherited from o1)
```



### 6.30 Summary

- Even though JavaScript is a "functional" language, it can be used in an object oriented way
- Constructor functions can define properties and functions on objects when called with the 'new' keyword
- Closures in JavaScript can extend the scope of variables local to a function declaration
- The 'prototype' property on an object can establish a relationship similar to inheritance in other object oriented languages
- When accessing a property of an object, if it is not found JavaScript will look to the object referenced by 'prototype' to keep searching for the property
- In addition to the prototype property, one constructor function can call another to combine the properties and functions defined by both

## Chapter 7 - Introduction to AngularJS

---

### *Objectives*

In this chapter, we will

- Introduce the AngularJS JavaScript framework
- Explain the MVC pattern
- Explain the differences between one- and two-way data binding



## 7.1 What is AngularJS?



- AngularJS is a framework to build the browser side front end portion of a web application.
  - ◇ Web site: <https://angularjs.org/>
  - ◇ Github: <https://github.com/angular/angular.js>
- Open source under MIT license, which allows you to use AngularJS in commercial applications (see details below).
- Originally developed in 2009 by a group of developers who worked for Brat Tech LLC. Later some of them joined Google and continued to work on the framework. Currently, the project is officially funded and promoted by Google.



Miško Hevery (the father of AngularJS)



## 7.2 Why AngularJS?

- AngularJS is a significantly different approach to front end development when compared to jQuery or similar conventional frameworks. So, why should we bother learning AngularJS? What can it do for us that jQuery can't do easily?
- AngularJS excels at many things. But, these three stand out the most:
  - ◇ By employing two way data binding it lets us manipulate DOM with much less code than jQuery.
  - ◇ Through the use of HTML templates, it lets us dynamically add complex DOM element structure which will be very difficult to do using plain DOM API or jQuery.
  - ◇ It lets us write Single Page Applications (SPA).
- In summary, if you are writing an SPA style application or need to do many complex DOM manipulations, AngularJS will be a framework of choice.





### 7.3 Scope and Goal of AngularJS

- Traditionally, rendering of views is done in the server side using templating technologies such as JSP, ASP or PHP. AngularJS let's you move display logic entirely in the browser side with its own template engine.
- Also, control of page flow has been done in the server side by the controllers (like Java Servlet and Spring controller). This logic too can be moved to the browser side using AngularJS and build a SPA style application.
- In summary, AngularJS lets you move most of the View and Controller logic in the browser leaving mostly the Model layer business logic in the server side as RESTful web services.
  - ◇ Hence, the scope and goal of AngularJS is much larger than a DOM API wrapper framework like jQuery. The API surface area is also much larger leading to a steeper learning curve.



## 7.4 Using AngularJS

- You can download the required version of the *angular.js* library from <https://angularjs.org/> web site and reference it in your page using the *script* tag. Do these to improve download time:
  - ◇ You use the minified version of the library.
  - ◇ Enable gzip compression of JavaScript files in your web server.
- Alternatively, you can load the file from Google's Content Delivery Network (CDN):

```
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.3.0/angular.min
.js"></script>
```



## 7.5 A Very Simple AngularJS Application

Hello world

You entered: Hello world!

```
<!doctype html>
<html>
 <head>
 <script src="angular.min.js"></script>
 </head>
 <body ng-app>
 <input type="text" ng-model="message"/>
 <p>You entered: {{message}}!</p>
 </body>
</html>
```



## 7.6 Building Blocks of an AngularJS Application

- **Directives** are custom HTML elements, like `<ng-repeat>` that encapsulate display rendering logic. Large chunks of reusable display logic can be hidden away in custom directives.
- **Filters** format data. For example, the currency filter formats a number into a currency appropriate for the browser's locale.
- **Services** contain pure business logic without any concern for how data is displayed or how user enters input. In most complex applications services invoke remote web services to access server side business logic and data.
- **Controllers** handle user input events. In SPA, it controls are page flow.
- A **Module** is a package that contains a collection of other artifacts like directives, controllers and filters.



## 7.7 Use of Model View Controller (MVC) Pattern

- MVC is a well known pattern employed in GUI applications including web applications. It has many benefits including separation of concern, easier testability and reuse.
- AngularJS lets you separate application code in distinct MVC layers.
- The **Model** layer contains data and business logic. In AngularJS, JavaScript objects hold data and **Services** contain business logic. Actual business logic and data usually reside in the server and be exposed as web services.
- The **Controller** layer in AngularJS is made up of **Controllers**. They handle user input events, populate view with data and interact with the model layer.
- The **View** layer in AngularJS is made up of templates, custom **Directives** and **Filters**. They encapsulate any display rendering logic, data formatting and styling.



## 7.8 A Simple MVC Application

- We will build a login screen with simple business logic. If Daffy Duck tries to login we will deny access. Anyone else can login without a problem.

Email:	<input type="text" value="daffy@wb.com"/>	Email:	<input type="text" value="bugs-bunny@wb.com"/>
Password:	<input type="password" value="...."/>	Password:	<input type="password" value="...."/>
<input type="button" value="Login"/>		<input type="button" value="Login"/>	
Bad user. Login failed.		Login was successful.	



## 7.9 The View

```
<!doctype html>
<html>
 <head>
 <script src="angular.min.js"></script>
 <script src="login.js"></script>
 </head>
 <body ng-app="SimpleApp" ng-controller="LoginCtrl">
 Email: <input type="text"
 ng-model="userInfo.email"/>

 Password: <input type="password"
 ng-model="userInfo.password"/>

 <button ng-click="login()">Login</button>
 <p>{{message}}</p>
 </body>
</html>
```



## 7.10 The Controller

```
angular.module('SimpleApp', [])
 .controller('LoginCtrl', function($scope) {
 $scope.userInfo = {
 email: "",
 password: ""
 };
 $scope.message = "Please login.";
 $scope.login = function() {
 //In real life, we will invoke a model layer function
 if ($scope.userInfo.email == "daffy@wb.com") {
 $scope.message = "Bad user. Login failed.";
 } else {
 $scope.message = "Login was successful.";
 }
 }
 });
```





## 7.11 Data Binding

- One way data binding is used to render value of a model object as text node or attribute of an HTML element. Example:

```
<p>{{message}}.</p>
<p class="{{theClass}}">Hello</p>
```

- Two way data binding is usually used with form input elements.
  - ◇ When the input element is first shown, its value is set to the value of the bound object.
  - ◇ As user interacts with the input element, AngularJS automatically updates the value of the bound object.

```
<input type="text" ng-model="userInfo.email"/>
```



## 7.12 Basics of Dependency Injection (DI)

- DI lets us use objects without knowing how or when such objects are created. You are simply handed an object that is ready to use.
  - ◇ This way, DI can simplify object life cycle management such as initialization.
  - ◇ Also, during testing, mock objects can be injected instead of real ones.
- There are many ways to do DI in AngularJS. Most commonly, objects are injected in the initialization functions of controllers and services.

```
angular.module("SimpleApp", [])
 .service("myService", function($http) {
 console.log("Initializing myService...");
 })
 .controller("TestCtrl", function($scope, myService) {
 console.log("Initializing TestCtrl...");
 });
```

```
<div ng-controller="TestCtrl"></div>
```

- System discovers artifacts by matching the parameter names so the order of the parameters does not matter.



### 7.13 Other Client Side MVC Frameworks

- There are other frameworks that take similar approach to front end development as AngularJS and have their own unique strengths and weaknesses.
- **Ext JS.** Unlike AngularJS it includes a full GUI component library and charting library.
- **React.** Appears to have a lower learning curve and higher performance according to many users.
- **Backbone.js** is a lightweight MVC framework created after AngularJS in 2010
  - ◇ It positions itself as a lean alternative to heavyweight MVC frameworks such as Ext JS



## 7.14 Summary

- AngularJS is an open-source JavaScript-based web application framework that brings the popular Model–View–Controller (MVC) pattern into the web page design and development
- Angular supports two-way data binding which compares much more favorably with the very rigid one-way data binding traditionally used in web page development
- Angular brings structure to your code as well as a clear separation of concerns between the MVC components

## Chapter 8 - AngularJS Module

---

### *Objectives*

Key objectives of this chapter

- What is a module?
- Module life cycle
- Module dependency management



## 8.1 What is a Module?

- A module is a container that contains a collection of other AngularJS artifacts like controller, service, provider, directives and filters.
- A module is created using the **angular.module()** method. It takes the name of the module and a list of modules that it depends on and returns the module object. **Note:** The dependency list is mandatory.

```
angular.module("SimpleApp", []);
angular.module("SimpleApp", ["AnotherModule"]);
```

- All module methods return the module object which allows us to chain methods:

```
angular.module("SimpleApp", [])
 .controller("MyCtrl", function(){...})
 .service("MyService", function(){...});
```

- A module that is already created can be looked up using the same method but without using the dependency list.

```
var m = angular.module("SimpleApp");
```



## 8.2 Benefits of Having Modules

- Artifacts like directive and controller can be named within the name space of the module without using the global name space thereby avoiding any conflict.
  - ◇ For example: two different modules can have controllers by the same name.
- A module is like a Java or C application with an entry point function. This function is registered using the `run()` method and is executed after all necessary artifacts have been initialized.
  - ◇ This entry point function allows us to perform some business logic before the HTML view has been rendered.



### 8.3 Life Cycle of a Module

- System first determines that the module needs to be instantiated and initialized. This can happen due to several reasons:
  - ◇ The module is referred using the ng-app attribute. E.g.:

```
<div ng-app="SimpleApp">
```

- ◇ Another module that needs to be instantiated depends on this module.
- System begins the **configuration phase** and executes all the configuration functions.
- System begins the **run phase** and executes all the run functions.





## 8.4 The Configuration Phase

- Configuration phase gives you a chance to change internal settings of a module before other artifacts in the module are initialized.
- You can register any number of functions using the `config()` method of the module that will be called during the configuration phase. Example:

```
angular.module('SimpleApp', [])
 .config(function() {
 //...
 })
 .config(function(myServiceProvider) {
 //...
 })
```

- Configuration phase happens very early before any service can be initialized. Hence, you can not inject services in to the parameters of a configuration function, but, you can inject service providers and constants.
  - ◇ This makes service providers and constants as ideal places to put any configuration settings. We will learn about them later.
  - ◇ Due to this limitation, avoid trying to do any complex business logic in the configuration phase. Do so in the run phase.



## 8.5 The Run Phase

- Run phase is used to perform one time initialization of the module. Unlike the configuration phase, in run phase you can perform complex business logic.
- You can register any number of run functions using the `run()` method of the module. These functions are called during the run phase. Example:

```
angular.module('SimpleApp', [])
 .run(function() {
 //...
 })
 .run(function(myService, $http) {
 //...
 })
```

- You can inject services in the parameter of a run function. Such services will be initialized first before the run function is called.



## 8.6 Module Wide Data Using Value

- You can store data at the module level that can be accessed by any artifact defined within the module. This is done using the `value()` method of the module.

```
angular.module("SimpleApp", [])
 .value("globalSettings", {
 baseUrl: "http://localhost/",
 timeout: 2000
 })
 .controller("MyCtrl", function(globalSettings) {
 console.log(globalSettings.baseUrl);
 });
```

- Values can be injected into services and controllers but not service providers.



## 8.7 Module Wide Data Using Constant

- A constant is similar to a value, except it can be injected into a configuration function. This makes a constant an ideal place to store any module wide settings.

```
angular.module('SimpleApp', [])
 .constant("backendSettings", {
 baseUrl: "http://localhost", //Default value
 publicToken: "ABC"
 })
 .config(function(backendSettings) {
 backendSettings.baseUrl = "http://example.com";
 })
 .controller("MyCtrl", function(globalSettings) {
 console.log(globalSettings.baseUrl);
 });
```



## 8.8 Module Dependency

- A module can depend on other modules. For example, if a controller from module A uses a service from module B then you should declare module A dependent on module B.

```
angular.module("ModuleA", ["ModuleB", "ModuleC"])
```

- Before the configuration and run methods of the module is executed, system will run them for the modules that it depends on.
- The HTML file that uses a module must include JavaScript files for all modules in the dependency chain. AngularJS will not do this for us.

```
<script src="angular.min.js"></script>
```

```
<script src="module_a.js"></script>
```

```
<script src="module_b.js"></script>
```

- The order in which the module JavaScript files are imported does not matter. Except, the AngularJS file must be imported first.
- If a module never appears in the dependency chain, importing its JavaScript file will do nothing. The module will never be initialized.



## 8.9 Using Multiple Modules in a Page

- AngularJS will initialize only the first module in a page. All subsequent modules need to be manually initialized using **angular.bootstrap()** method. Example:

```
<body>
<div ng-app="SimpleApp">
</div>
<div ng-app="AnotherApp" id="div2">
</div>
<script>
angular. bootstrap(
 document.getElementById("div2"), ["AnotherApp"]);
</script>
</body>
```



## 8.10 Summary

- A module provides a simple name space mechanism where artifacts like services and directives are defined.
- During initialization, system first runs the configuration functions followed by run functions.
- A configuration function is run very early before any service can be initialized. Hence, you can inject service providers only and can not perform any complex business logic.
- A run function acts as an entry point to the module. You can perform one time initialization of the module. You can inject services to these functions.
- Modules can depend on other modules. AngularJS takes care of initialization of these other modules in the dependency chain.

## Chapter 9 - AngularJS Controllers

---

### *Objectives*

In this chapter, we will

- Review controller main responsibilities
- The \$scope object considerations
- Compare in-line expressions with Controller-based logic





## 9.1 Controller Main Responsibilities

- In AngularJS, controllers have the following responsibilities:
  - ◇ Set up the initial state of the model. Usually, a controller retrieves the data from a remote web service.
  - ◇ Handle user input events. This usually involves validating the input and invoking a remote web service to complete a task.
  - ◇ Observe and react to changes to the model objects.
- Primary use of controllers is to act as a liaison between the view and the model layers. For larger applications, avoid doing any complicated business logic directly from a controller. Move such activities to services and use services from controllers.
- In a Single Page Application, controllers contain the logic to control the page transition sequence.



## 9.2 About Constructor and Factory Functions

- AngularJS uses two types of functions to create instances of artifacts like controller and services.
- A **constructor function** – a standard concept in JavaScript – is used as a template to create a new object at a later time.

```
var Circle = function(r) {
 this.radius = r;
}
var c1 = new Circle(10);
```

- A **factory function** creates the object and returns it.

```
var CircleFactory = function(r) {
 return {
 radius: r
 }
}
var c1 = CircleFactory(100);
```

- Not all artifacts can be created using both types of functions. Details are covered in the specific chapters for the artifacts.



### 9.3 Defining a Controller

- A controller is defined by registering a constructor function using the **controller()** method of a module.

```
angular.module("SimpleApp", [])
 .controller("MyCtrl", function() {});
```

- The initialization function can receive injected objects as parameters.

```
angular.module("SimpleApp", [])
 .controller("MyCtrl", function($scope, $http) {});
```

- The constructor function is where the initial state of the model is set up.



## 9.4 Using the Controller

- To use a controller in a page, use the **ng-controller** attribute with a DOM element.

```
<div ng-app="SimpleApp">
 <div ng-controller="MyCtrl">
 </div>
</div>
```

- That DOM element will serve as the root of the hierarchy that will be managed by the controller instance.
- A new instance of the controller is created for each declaration using ng-controller using the constructor function.

```
<div ng-app="SimpleApp">
 <div ng-controller="MyCtrl"> ...</div>
 <div ng-controller="MyCtrl"> ...</div>
</div>
```



## 9.5 Controller Constructor Function

- This is a JavaScript constructor function that is used to create a new instance of a controller. This is where we setup the initial model data.
- There are two different ways to store model data:
  - ◇ Using the scope of the controller instance. This is the prevalent approach and we will learn about it first.
  - ◇ As properties of the controller instance.
- The scope of a controller can be obtained by injecting the `$scope` object. Example:

```
.controller("MyCtrl", function($scope) {
 $scope.product = {
 productId: 1001,
 title: "Baseball gloves",
 price: 23.99
 }
});
```

- Once the model data has been set up, it can be bound to DOM elements:

```
<div ng-controller="MyCtrl">{{product.title}}</div>
```



## 9.6 More About Scope

- A scope is an object where you can store model objects as properties.
- Every module instance gets its own scope. This can be accessed by injecting the **\$rootScope** object.
- Every controller instance gets its own scope that can be accessed by injecting the **\$scope** object.
- The \$scope of a controller inherits from the \$rootScope using prototype chaining. Which means, objects added to the \$rootScope is visible to the child \$scope.
- A controller can contain other child controllers. The \$scope of the child controllers will inherit from the parent's \$scope using prototype chaining.



## 9.7 Example Scope Hierarchy

```
angular.module("SimpleApp", [])
 .run(function($rootScope) {
 $rootScope.message = "Hello";
 })
 .controller('ParentCtrl', function($scope) {
 $scope.message = $scope.message + " World.";
 })
 .controller('ChildCtrl', function($scope) {
 $scope.message = $scope.message + " How are you?";
 });
```



## 9.8 Using Scope Hierarchy

```
<div ng-app="SimpleApp">
 {{message}} <!-- Root scope -->
 <div ng-controller="ParentCtrl">
 {{message}}
 <div ng-controller="ChildCtrl">
 {{message}}
 </div>
 </div>
</div>
```

Hello  
Hello World.  
Hello World. How are you?





## 9.9 Modifying Objects in Parent Scope

- The way JavaScript prototype works, you can read a property from parent object if the property is not defined in the child. However, if a child adds the property, it is added to the child and not to the parent.
- This means, to be able to change a value in parent scope, you must add it as a JavaScript object rather than a primitive type like String.

```
.controller('ParentCtrl', function($scope) {
 $scope.message = "Hello World.";
 $scope.product = {
 title: "Baseball gloves"
 };
})
.controller('ChildCtrl', function($scope) {
 $scope.message = "Hello Moon."; //Doesn't update parent
 $scope.product.title = "Volleyball net"; //Updates parent
});
```



## 9.10 Modified Parent Scope in DOM

```
<div ng-controller="ParentCtrl">
 {{message}} {{product.title}}
 <div ng-controller="ChildCtrl">
 {{message}} {{product.title}}
 </div>
</div>
```

Hello World. Volleyball net  
Hello Moon. Volleyball net

- The child has successfully updated parent scope's product.title but not the message property.
- Since all controller instances are created before the DOM is rendered, we never get to see the original value of product.title - "Baseball gloves".



## 9.11 Handling Events

- Add event handling functions to the \$scope as properties.

```
.controller("TestCtrl", function($scope) {
 $scope.message = "Hello ";
 $scope.popIt = function(planet) {
 alert($scope.message + planet);
 }
});
```

- Register the event handler in DOM using `ng-event_name` attribute.

```
<div ng-controller="TestCtrl">
 <button ng-click="popIt('World')">Test</button>
</div>
```

- Event handler methods can take any number of arguments.



## 9.12 Another Example for Event Handling

```
.controller('TestCtrl', function($scope) {
 $scope.className = "normal";
 $scope.activate = function() {
 $scope.className = "active";
 }
 $scope.normalize = function() {
 $scope.className = "normal";
 }
});
```

```
<style>
.normal {
 background: green
}
.active {
 background: red
}
</style>
```

```
<div ng-controller="TestCtrl" class="{{className}}"
 ng-mouseenter="activate()" ng-mouseleave="normalize()">
```



Hello there!

</div>



### 9.13 Storing Model in Instance Property

- An alternative way to store model data is to add them as properties of the controller instance.

```
.controller('TestCtrl', function() {
 this.message = "Hello World";
 this.showIt = function() {
 alert(this.message);
 }
});
```

- Declaring the controller in DOM requires a slightly different approach.

```
<div ng-controller="TestCtrl as test">
I will say {{test.message}}
<button ng-click="test.showIt()">Test</button>
</div>
```



## 9.14 Summary

- An AngularJS controller's primary job is to:
  - ◇ Initialize model data from its constructor function.
  - ◇ Handle DOM events as the user interacts with the page.
- There are two ways you can set up the model data:
  - ◇ Using the controller's scope.
  - ◇ Use properties of the controller object.
- The scope of a controller inherits from the root scope of the module using prototype chain. The same goes for a child controller embedded inside a parent controller.

## Chapter 10 - AngularJS Expressions

---

### *Objectives*

In this chapter, we will

- Discuss AngularJS expressions
- Review considerations for using *src* and *href* HTML attributes





## 10.1 Expressions

- The dynamic nature of AngularJS pages is shown in its support for expressions that are passed to Angular directives
- Angular expressions are JavaScript-like code snippets normally placed in bindings:  
`{{ expression }}`
  - ◇ The `{{2 + 2}}` expression will be diligently evaluated to 4
- Expressions attached to some user controls, should be used as a string literal, e.g.

```
<button ng-click='i = i + 1'>Increment the value</button>
```

- Expressions use references to data model variables that can be used to create complex logic using a wide range of supported operations (math, logical, etc.)
- They are safe to use (more on this a bit later ...)



## 10.2 Operations Supported in Expressions

- Math operations

- +, -, /, \*, % (modulo)

- {{some\_model\_var % 2}}

- {{sum + sum/10}}

- Comparisons

- ==, !=, >, <, >=, <=

- {{ variable > threshold }}



## 10.3 Operations Supported in Expressions

- Boolean logic

&&, ||, !

- `{{ taskComplete = stepOneDone && stepTwoDone }}`

- Bitwise operations

^, &, |



## 10.4 AngularJS Expressions vs JavaScript Expressions

- Expressions are a much simplified form of regular JavaScript expressions
- AngularJS expressions are more limited in functionality compared to JavaScript expressions
  - ◇ JavaScript expressions are evaluated against the global *window* object
  - ◇ Angular expressions are evaluated against a *scope* object



## 10.5 AngularJS Expressions are Safe to Use!

- AngularJS expressions are evaluated by its own parser (not by using JavaScript's *eval()* function)
- in this way, expressions are handled by Angular not to throw `NullPointerException`, `ReferenceError` or `TypeError` exceptions when dealing with *undefined* and *null* model values
- When the Angular parser fails to evaluate an expression, it simply suppresses any exceptions and UI stays unchanged
  - ◇ You may get a *NaN* result, though
- **Note:** If you want to evaluate an Angular expression outside of the `{{}}` binding notation, you can use the *\$eval()* method



## 10.6 What is Missing in Expressions

- AngularJS expression lack the following features:
  - ◇ Looping constructs
    - for, while
  - ◇ Flow-of-control operators
    - if-else if-else, exceptions
  - ◇ Data mutators
    - ++, --
- **Note:** You have to implement those types of operations in your controllers or via custom directives



## 10.7 Considerations for Using src and href Attributes in Angular

- Avoid combining Angular binding interpolation notation with regular `<img>` or `<a>` HTML tags, e.g.

```
 <!-- Not safe !!-->
```

- Browsers may start fetching images on a thread separate from the one used to load the angular.js file
  - ◇ This creates a race condition which may result in AngularJS not being present to update the View with Model values
- To deal with these situations, AngularJS provides "safe" directives for handling image and a/href bindings:
  - ◇ ng-src
  - ◇ ng-href



## 10.8 Examples of ng-src and ng-href Directives

- The **ng-src** and **ng-href** directives offers developers a "safe" way to load images and build a/href links that you want to use in your AngularJS pages:

```

```

```
<a ng-href="/catalog/{{ item }}">View the item
```





## 10.9 Summary

- In this module we discussed AngularJS expressions
- AngularJS expressions support a wide range of math, boolean and other operations normally expected in expressions
- Angular uses its own parser to evaluate expressions which, in case of an evaluation error, will not result in an error message printed on the page

## Chapter 11 - Basic View Directives

---

### *Objectives*

In this chapter, we will review some of the directives used to control UI of your Angular applications, such as:

- ng-show,
- ng-hide,
- ng-class



## 11.1 Introduction to AngularJS Directives

- AngularJS directives are custom HTML elements and attributes that are used to manipulate DOM elements in a page.
  - ◇ We have already encountered a few directives like **ng-app** and **ng-controller**.
- AngularJS ships a number of directives and you can develop your own.
- Directives capture bulk of the view layer logic. Example tasks that you can achieve using them include:
  - ◇ Show or hide DOM elements.
  - ◇ Apply certain styling to the elements based on the current model state.
  - ◇ Iterate through a list of objects and display their information.



## 11.2 Controlling Element Visibility

- Showing and hiding HTML elements is done with the help of *ng-show* and *ng-hide* directives.

```
<div ng-show="expression">Hello</div>
```

- These directives are used as attributes to DOM elements. Their value being a boolean expression.
  - ◇ *ng-show* shows the element if the expression evaluates to true.
  - ◇ *ng-hide* hides the element if the expression is true.

```
.controller("TestCtrl", function($scope) {
 $scope.buttonLabel = "Hide";
 $scope.isShown = true;
 $scope.showIt = function() {
 $scope.isShown = !$scope.isShown;
 $scope.buttonLabel = $scope.isShown ? "Hide" : "Show";
 }
});
```

```
<div ng-controller="TestCtrl">
 <div ng-show="isShown">Now you see me!</div>
 <button ng-click="showIt()">{{buttonLabel}}</button>
</div>
```



### 11.3 Adding and Removing an Element

- ng-show and ng-hide directives hide an element by simply setting the "display" style to "none". The element still remains in the DOM tree.
- To completely remove and add an element from the DOM tree use the **ng-if** directive. If the value expression is false the element is completely removed from the DOM. If it is true, it is added back in.
- The same example as above can be implemented using ng-if.

```
<div ng-controller="TestCtrl">
 <div ng-if="isShown">Now you see me!</div>
 <button ng-click="showIt()">{{buttonLabel}}</button>
</div>
```



## 11.4 Dynamically Changing Style Class

- Simply set the class attribute to an expression:

```
.controller('TestCtrl', function($scope) {
 $scope.classList = "bold red";
});
```

```
<div class="{{classList}}">
```

- If the value of the expression changes, AngularJS will update the DOM with the new class name.
- The expression can evaluate to a space separate list of classes and all of those classes will be applied to the element.



## 11.5 The ng-class Directive

- When an element needs to be applied multiple different classes based on certain model state, the **ng-class** directive can be easier to use than directly setting the class.
- *ng-class* and *ng-style* take an expression that gets evaluated to one of the following:
  - ◇ A string representing space-delimited class names
  - ◇ An array of class names
  - ◇ A JavaScript object where the name of each property is a class name and the value is an expression. If the expression evaluates to true then that class is applied.



## 11.6 Example Use of ng-class

```
.controller('TestCtrl', function($scope) {
 $scope.isBold = true;
 $scope.isError = true;
 $scope.isUnderlined = false;
});
```

```
<div ng-controller="TestCtrl">
<div ng-class="{
 underlined: isUnderlined, bold: isBold, error: isError
 }">Style me!</div>
</div>
```

- System will apply the bold and error classes.





## 11.7 Setting Image Source

- Dynamically setting image source using an expression such as `src="{{expression}}"` does not work well. Most browsers will start to download image by literally interpreting `"{{expression}}"` as the URL. This will fail.
- To solve the problem, use the **ng-src** directive instead.

```
.controller('TestCtrl', function($scope) {
 $scope.product = {
 sku: "P1029",
 title: "Batman mask",
 image: "/images/mask.jpg"
 };
});


```



## 11.8 Setting Hyperlink Dynamically

- Setting `<a href="{{expression}}">` can cause problem if user clicks the link before AngularJS had a chance to process the element.
- To resolve the issue, use the **ng-href** directive instead.

```
<a ng-href="/details/{{product.sku}}">{{product.title}}
```



## 11.9 Preventing Initial Flash

- Some browsers, notably IE, briefly shows the raw HTML before AngularJS had a chance to process it. This can lead to an unpleasant flash of the raw code on screen.
- To prevent that, use **ng-cloak** directive. The applicable element will be hidden until AngularJS has finished processing the DOM.
  - ◇ You can apply at the body level or for a child element.

```
<body ng-cloak>
```



## 11.10 Summary

- Directives contain bulk of the logic for the view layer.
- AngularJS ships with a bunch of directives and you can create your own.
- We learned about a few directives in this chapter:
  - ◇ ng-show, ng-hide – Controls visibility of an element.
  - ◇ ng-if – Adds or removes an element from the DOM.
  - ◇ ng-class – Dynamically changes style class of an element.
  - ◇ ng-src – Sets image source dynamically.
  - ◇ ng-href – Sets hyperlink dynamically.
  - ◇ ng-cloak – Prevents initial display of raw HTML in some browsers.

## Chapter 12 - Advanced View Directives

---

### *Objectives*

In this chapter, we will discuss these directives:

- ng-repeat
- ng-switch
- ng-include



## 12.1 The ng-repeat Directive

- The *ng-repeat* is used to iterate over a collection of items, usually an array, and render them in the view.
- Apply the ng-repeat directive to the element that will be repeated along with its children.
- Also, *ng-repeat* will continuously monitor the source collection for any changes and dynamically reflect those changes in the View
  - ◇ It works directly on DOM, moving, adding, changing and deleting DOM nodes as needed



## 12.2 Example Use of ng-repeat

```
.controller('TestCtrl', function($scope) {
 $scope.colors = ["Red", "Green", "Blue"];
});
```

```
<div ng-controller="TestCtrl">
```

Available colors:

```

 <li ng-repeat="color in colors"> {{color}}

</div>
```

Available colors:

- Red
- Green
- Blue

- The `<li>` element will be repeated. That is, a new `<li>` element will be added to the DOM for each color.



## 12.3 Dynamically Adding Items

- Adding or removing items from the collection will be immediately reflected on the view.

```
.controller('TestCtrl', function($scope) {
 $scope.colors = ["Red", "Green", "Blue"];
 $scope.newColor = "";
 $scope.addColor = function() {
 $scope.colors.push($scope.newColor);
 }
});
```

```
<div ng-controller="TestCtrl">
```

Available colors:

```
 <li ng-repeat="color in colors"> {{color}}
 <input type="text" ng-model="newColor"/>
 <button ng-click="addColor()">Add</button>
</div>
</div>
```





## 12.4 Special Properties

- The *ng-repeat* directive also has special properties that may greatly simplify the developer's life:
  - ◇ **\$index** - The index) of the current element, starting with 0.
  - ◇ **\$first, \$last** - A boolean that is set to true if the element is the first in the collection or last.
  - ◇ **\$even, \$odd** - A boolean value indicating if \$index is even or odd.



## 12.5 Example: Using the \$index Property

```
<div ng-controller="TestCtrl">
Available colors:
 <p ng-repeat="color in colors">
 Color {{$index}} is: {{color}}
 </p>
</div>
```

Available colors:

Color 0 is: Red

Color 1 is: Green

Color 2 is: Blue



## 12.6 Scope and Iteration

- Each iteration creates a new instance of an HTML template consisting of the element where ng-repeat was applied and its children.
- Each template instance gets its own scope.
- Each item in the collection and special properties like \$index are saved in the instance's scope.
- For example, the scope of each template instance will have a "color" property along with \$index, \$first, \$last etc.

```
<p ng-repeat="color in colors">
 Color {{$index}} is: {{color}}
</p>
```

- This use of separate scope greatly simplifies event handling as we will see next.



## 12.7 Event Handling in Iterated Elements

```
.controller('TestCtrl', function($scope) {
 $scope.directory = [
 {id: 100, name: "Daffy Duck"},
 {id: 200, name: "Bugs Bunny"}
];
 $scope.selectPerson = function(p) {
 alert("You selected " + p.name);
 }
});
```

```
<div ng-controller="TestCtrl">
<p ng-repeat="person in directory">
 {{person.name}}
 <button ng-click="selectPerson(person)">Choose</button>
</p>
</div>
```



## 12.8 The ng-switch Directive

- Makes it easy to conditionally swapping DOM elements in and out of view. Easier to use than ng-show when one of several possible views need to be displayed.
- The **on** attribute of ng-switch takes an expression which is evaluated.
- Each element that will be shown or hidden is given an **ng-switch-when** attribute. This points to an expression. If the evaluated value of the ng-switch expression matches this, the element is shown.



## 12.9 Example Use of ng-switch

```
.controller('TestCtrl', function($scope) {
 $scope.state = {
 page: "page-1"
 };
});

<div ng-controller="TestCtrl">
 <div ng-switch on="state.page">
 <h1>Survey</h1>
 <div ng-switch-when="page-1">Page 1 <a href="#" ng-
click="state.page='page-2'">Next</div>
 <div ng-switch-when="page-2">Page 2 <a href="#" ng-
click="state.page='page-3'">Next</div>
 <div ng-switch-when="page-3">Page 3 <a href="#" ng-
click="state.page='page-1'">Start over</div>
 <div ng-switch-default
```




## 12.10 Inserting External Template using ng-include

- ng-include lets us dynamically switch the view like ng-switch. Except, the HTML template comes from an external file.
  - ◊ Use ng-include in place of ng-switch when the HTML markup of the views is complex or when the markup needs to be re-used in multiple places.
- The same example reworked:

```
.controller('TestCtrl', function($scope) {
 $scope.state = {
 page: "page-1.html"
 };
});
```

```
<div ng-controller="TestCtrl">
 <h1>Survey</h1>
 <div ng-include src="state.page">
 </div>
</div>
</div>
```

```
<!-- The page-1.html file -->
<div>Page 1 <a href="#" ng-click="state.page='page-
```

2.html'">Next</a></div>





## 12.11 Summary

- The *ng-repeat* directive is a very powerful mechanism used by AngularJS for iterating over data collections
- *ng-switch* is used to flip between a collection of views.
- *ng-include* is used to display content from an external template file.

## Chapter 13 - Working with Forms

---

### *Objectives*

In this chapter, we will

- Explain the way AngularJS works with HTML form elements
- Review main ideas behind the "Unobtrusive" JavaScript
- Learn how to initialize HTML form's elements



## 13.1 Forms and AngularJS

- If you plan on moving the display rendering logic from server to the browser side, working with forms can become challenging. These are some of tasks you will have to deal with:
  - ◇ Initialize the state of the input elements. Such as populate text box, check the correct radio button etc.
  - ◇ Dynamically add <option> elements to <select> element and select the correct option element based on the current model state.
  - ◇ Do client side validation, show error message and style invalid input elements.
- AngularJS makes these common tasks pretty easy to achieve.
  - ◇ Most of the heavy lifting is done through the two way data binding.
  - ◇ AngularJS also has support for client side validation.



## 13.2 Scope and Data Binding

- As we already discussed, if an input element appears in a nested child scope of the controller, then the bound model variable should not be a primitive data type, such as string or number. It should be a property of an object.
  - ◇ This is a very common beginner mistake. See the detail example in the notes section.
- Directives like ng-if and ng-switch create their own scope.
- Not all directives create their own scope. For example, ng-show and ng-hide do not need to create their own scope.
- To be safe, if a variable is to be bound to a form input, make sure that it is a property of an object.



### 13.3 Role of a Form

- In a typical AngularJS application, submitting a form should not cause a regular POST and loading of a new page. Most applications will do a POST (or GET) using Ajax and show the result on the existing page or transition to a new view.
- Technically, you don't even need to include input elements in a `<form>`. Simply issue an Ajax request from the `ng-click` handler of a button.
- Since, the user can submit a form by hitting enter in a text box, it is safer to use a `<form>`. If the `action` attribute is missing (which should be the case for most AngularJS application), then instead of calling the DOM submit event handler:
  - ◇ The **ng-submit** event handler of the form is called.
  - ◇ Alternatively, you can set an **ng-click** event handler for the first submit button of the form and it will be called.

```
<form ng-controller="MyCtrl" ng-submit="sendrequest()">
 ...
</form>
```



## 13.4 Using Input Text Box

- Bind a string value to input type text and number value to input type number. AngularJS will preserve this data type which is important if you need to send the input data via a JSON request.
- Optionally set **ng-trim="true"** to trim input.

```
.controller("TestCtrl", function($scope) {
 $scope.user = {age: 0, email: ""};
});
```

```
<div ng-controller="TestCtrl">
 <input type="text" ng-model="user.email" ng-trim="true"/>
 <input type="number" ng-model="user.age"/>
</div>
```



## 13.5 Using Radio Buttons

- Group radio buttons by binding them to the same model. There is no need to use the name attribute to group them as in standard HTML.

```
.controller("TestCtrl", function($scope) {
 $scope.user = {
 currency: "USD",
 emailMe: "Y"
 };
});
```

```
<div ng-controller="TestCtrl">
 E-mail me promotions:
 <input type="radio" ng-model="user.emailMe" value="Y"/> Yes
 <input type="radio" ng-model="user.emailMe" value="N"/> No

 Preferred currency:
 <input type="radio" ng-model="user.currency" value="USD"/> USD
 <input type="radio" ng-model="user.currency" value="CAD"/> CAD
 <input type="radio" ng-model="user.currency" value="EUR"/> EUR
</div>
```



## 13.6 Using Checkbox

- For the simplest use, bind a boolean value to the checkbox.

```
.controller("TestCtrl", function($scope) {
 $scope.user = {
 emailMe: true
 };
});

<div ng-controller="TestCtrl">
 <input type="checkbox" ng-model="user.emailMe"/>
 E-mail me promotions.
</div>
```





## 13.7 Using Checkbox - Advanced

- You can also bind string values to a checkbox. In that case set:
  - ◇ **ng-true-value** to a constant that will be set to the bound model if user checks the checkbox.
  - ◇ **ng-false-value** to a constant that will be set to the bound model if user unchecks the checkbox.

```
.controller("TestCtrl", function($scope) {
 $scope.user = {
 emailMe: "Yes"
 };
});
```

```
<div ng-controller="TestCtrl">
 <input type="checkbox"
 ng-model="user.emailMe"
 ng-true-value="'Yes'"
 ng-false-value="'No'"/>
 E-mail me promotions.
</div>
```



## 13.8 Using Select

- Bind the value of the selected option to the <select> element.

```
.controller("TestCtrl", function($scope) {
 $scope.country = "MX"; //Default selected
});
```

```
<div ng-controller="TestCtrl">
<select ng-model="country">
 <option value="US">USA</option>
 <option value="CA">Canada</option>
 <option value="MX">Mexico</option>
</select>
</div>
```

- Mexico will be selected by default.
- If user selects Canada then \$scope.country will be set to "CA".



## 13.9 Using Select – Advanced

- If `<option>` elements need to be dynamically added to a `<select>` then you can use an array to store the data that will be used to create the options.

```
.controller("TestCtrl", function($scope) {
 $scope.countryList = [
 {code: "US", name: "USA"},
 {code: "CA", name: "Canada"},
 {code: "MX", name: "Mexico"}
]; //This data can come from server
 $scope.country = "MX"; //Default selected
});
```

- Use the **ng-options** attribute to add the options to a `<select>`

```
<select ng-model="country"
 ng-options="c.code as c.name for c in countryList">
</select>
```



### 13.10 Reacting to Model Changes in a Declarative Way

- The *ng-change* directive allows developers to specify a JavaScript function that Angular will invoke when the property referenced by the *ng-model* changes due to user input
- Angular transparently intercepts the appropriate *on-change* event for the input control and forwards it to the user-defined function for handling
  - ◇ For example, the *ng-change* directive applied to a text input box will associate the input control's *onKeyUp* JavaScript event with the user-defined function
- The user "on-change" function is specified within the matching Controller
- It also needs to be declared as a property of the *\$scope* context object



## 13.11 Example of Using the on-change Directive

- First, you add the *on-change* directive to the HTML Form element you need:

```
<input type="checkbox" ng-model="userChoice"
 ng-change= "changeHandler()">
```

- Then, you reference the function in the Controller code:

```
$scope.changeHandler = function() {
 // here you have access to the userChoice property via
 // $scope.userChoice reference
}
```



## 13.12 Summary

- AngularJS uses the *ng-model* attribute to bind all the standard HTML form elements to the Model
- Data binding does its best to preserve data type. For example, a number can be bound to an input of number type and boolean value can be bound to a checkbox.
- We can dynamically add option elements to a select using an array of objects.

## Chapter 14 - Formatting Data with Filters in AngularJS

---

### *Objectives*

In this chapter, we will

- Review Angular filters
- See examples of using filters



## 14.1 What are AngularJS Filters?

- An Angular filter formats or transforms the value of an expression for subsequent use in code and rendering in the View
- In addition to the View (UI templates), filters can be used in controllers and services
- Filters can be piped together by using the '|' character creating complex filtering and transformation logic
  - ◇ The output produced by the first filter in the piped chain will be used as input to the second filter in the pipe chain, and so on (similarly to the way commands piping in the Unix command-line works)
- Developers can also create their own filters





## 14.2 The Filter Syntax

- In the HTML template binding scenario, AngularJS filters use the following syntax:

```
{{ expression | filterName : parameter1 : ...parameterN }}
```

where

- *expression* is any valid Angular expression
- *filterName* is the name of the filter
  - the *filterName* is separated from the preceding expression by the pipe ( '|' )
- *parameter(s)* is a colon-separated list of input parameters to the filter
  - A parameter can be any valid Angular expression



## 14.3 Angular Filters

Filter Name	Description
<b>filter</b>	Selects a subset of items from an array
<b>currency</b>	Formats a number to a currency format
<b>number</b>	Formats a number as text
<b>date</b>	Formats a date to a string as per format
<b>orderBy</b>	Orders an array by an expression



## 14.4 More Angular Filters

Filter Name	Description
<b>json</b>	Converts a JavaScript object into JSON string
<b>lowercase</b>	Formats a string to lowercase
<b>uppercase</b>	Formats a string to uppercase
<b>limitTo</b>	Creates a new array or string containing only a specified number of elements



## 14.5 Using Filters in JavaScript

- You can also use filters in your controllers and services by invoking the *\$filter()* AngularJS function which has the following syntax:

```
$filter('filter_name')(model_variable[, param1, param2, ...])
```

where

- *filter\_name* is the name of the filter
- *model\_variable* is the variable you apply the filter to
- (optional) *param1*, *param2*, ... are parameters to the filter



## 14.6 Using Filters

- `{{ someText | uppercase }}` will display the content of the *someText* model variable in the upper case
- `{{ (base + base / 10) | currency }}` will apply the currency formatting filter (just a formatter, really) to the `base + base / 10` mathematical expression and render the output as per browser's Locale (e.g. \$123.80 in North America)
  - ◇ **Note:** You can optionally pass the currency symbol as a parameter to the currency filter, e.g. *currency : "CAD"*



## 14.7 A More Complex Example

- You can use filters in directives (the example below shows how you can pipe filters together using the '|' (pipe) character)
- The following command

```
ng-repeat="i in items | filter:id | orderBy:'category'"
```

- ◇ will build a list containing only those items from the underlying *items* collection that have *id*'s matching the *id* parameter;
- ◇ the generated list will also be sorted by the *category* field



## 14.8 The date Filter

- The full syntax of the date filter is as follows:

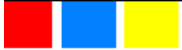
- ◊ In HTML template binding scenario:

```
{{ date_expression | date : format : timezone }}
```

- ◊ In JavaScript:

```
$filter('date')(date, format, timezone)
```

- ◊ **Note:** The format and timezone parameters are optional (the browser default locale will be used when they are omitted; the date will be rendered using the *mediumDate* format: 'MMM d, y' for en\_US locale)



## 14.9 The date's format Parameter

- **'yyyy'** - 4 digit year
- **'yy'** - 2 digit year
- **'MMMM'** - month in year (January, February, ...)
- **'MM'** - padded month in year (01 - 31)
- **'H'** - hour in day (0-23)
- etc.
- You can see a complete list of supported formatting elements at <https://docs.angularjs.org/api/ng/filter/date>





## 14.10 Examples of Using the date Filter

- The following binding

```
{{1409927335005 | date:"MM/dd/yyyy 'at' h:mm:s"}}
```

will produce 09/05/2014 at 10:28:55

And

```
{{1409927335005 | date:'yyyy-MM-dd HH:mm:ss Z'}}
```

will produce 2014-09-05 10:28:55 -0400

- **Note:** 1409927335005 is the number of milliseconds since epoch (1 January 1970)



## 14.11 The `limitTo` Filter

- The full syntax of the *limitTo* filter is as follows:

- ◊ In the HTML template binding scenario:

```
{{ array_or_string | limitTo : limit }}
```

- ◊ In JavaScript:

```
$filter('limitTo')(array_or_string, limit)
```

- ◊ **Note:** The *limit* is a positive or negative integer. A positive *limit* will take the `limitTo` number of elements / characters from the beginning of the input array / string , while a negative value will select elements / characters from the end



## 14.12 Using limitTo Filter

- The following filtered binding

```
{{ [1,2,3,4,5] | limitTo: 2}}
```

- will produce [1,2]

```
{{ 'I, robot' | limitTo: 5}}
```

- will produce 'I, ro'

```
{{ [1,2,3,4,5] | limitTo: -2}}
```

- will produce [4,5]

```
{{ 'I, robot' | limitTo: -3}}
```

- will produce 'bot'



### 14.13 Filter Performance Considerations

- When filters are used in the View (UI template), they become a regular AngularJS expression and are frequently evaluated
- This is how AngularJS detects changes in any object's state (using the "dirty checking" technique) in order to trigger UI repaints
- Complex filter chains (or poorly written custom filters) may impact performance or user experience of your applications (e.g. page flickering)
- In some cases, extracting and applying the filtering logic to the original data set before it is rendered (now without the filter) is one way to address such issues



## 14.14 Summary

- Angular offers a number of stock filters that are used for data formatting and transformation
- You can use filters in HTML template bindings, as well as in controllers and services using the *\$filter()* JavaScript function
- You can also create your own filters

## Chapter 15 - AngularJS \$watch Scope Function

---

### *Objectives*

In this chapter, we will

- Introduce the \$watch scope function
- Review potential developer pitfalls and gotchas when working with \$watch
- Look at associated performance considerations



## 15.1 The \$watch Function

- AngularJS provides the *\$watch()* function as means to observe and, if necessary, react to changes in the Model
- In addition to observing individual model properties you can also observe computed results as returned by JavaScript functions
- There are some performance issues with using this function (more on it in subsequent slides)



## 15.2 The \$watch Function Signature

- The *\$watch* function has the following signature:

`$watch(observableObject, watchAction, deepWatchFlag)`

- ◊ Description of the parameters is given on the next slide





## 15.3 The \$watch Function Details

- **observableObject**

- ◇ An Angular expression or a function that returns the current value of the model that you want to observe

- **watchAction**

- ◇ A user-defined JavaScript function or expression to be called when the *observableObject* changes
- ◇ The function form of the callback has the following signature:

```
function(newValue, oldValue, scope)
```

where the *newValue* and *oldValue* apply to the new and old (before the change) state of *observableObject*, *scope* is a reference to the scope context object

- **deepWatchFlag**

- ◇ Optional boolean flag to indicate if each property within the watched object needs to be observed for changes



## 15.4 Canceling the Watch Action

- The *\$watch* function returns a function that can be used to cancel the observation of the model
  - ◊ The underlying monitoring listener will stop running and will be de-registered
- Usage:

```
var cancel = $scope.$watch('a_model_property', userCallBack());
. . . // you observe the changes in a_model_property and get notification on its
changes in userCallBack();
. . .
cancel(); // You cancel the monitoring process
```



## 15.5 Example of Using \$watch

- The following skeleton code shows how you can implement a re-stock alert to an operator when the number of items (backed up by an array of objects) drops below 1000

```
var watchForReStockLimit = function() {
 if ($scope.items.length < 1000) {
 alert("Need to re-stock!");
 }
}

...
$scope.$watch('items.length', watchForReStockLimit);
```



## 15.6 Things to be Aware Of

- In order to be properly notified on model modifications, you need to place your *\$watch* function at the bottom of your Controller definition
- Your *watchAction* callback function may change the Model, which, in turn, will trigger the *\$watch* model change listener to call it again, resulting in a recursive call chain
  - ◇ AngularJS sets up the limit of recursive calls to 10 to prevent such situations



## 15.7 More Things to Be Aware Of

- During watcher initialization sequence, the listener may be called even if the observable model parameter didn't change
- If this "unnecessary" call during the watch initialization sequence must be avoided, use the *watchAction* in the function form: `function(newValue, oldValue, scope)`
  - ◇ Inside the function, compare the **newVal** and **oldVal** using the strict comparison operator (`===`)
    - If these two values are identical (`'==='` comparison operation returns true), then ignore that invocation



## 15.8 Performance Considerations

- **observableObject**

- ◇ This expression gets evaluated multiple times; make sure it is not computationally complex

- **deepWatchFlag**

- ◇ If you use this (optional) boolean flag, you may impact performance of your application forcing AngularJS to watch elements in a potentially large collection of properties



## 15.9 Speeding Things Up

- AngularJS implements its data binding in JavaScript which has inherent speed limitations as any other interpreted language
- Google is working with the TC39 – ECMAScript (JavaScript) technical committee to see if a new low-level structure called *Object.observe()* can be natively implemented in JavaScript runtimes which will give browsers the ability to track JavaScript object (data model) changes
- With the *Object.observe()* implementation in place, data binding can be performed at native data speeds



## 15.10 Summary

- AngularJS provides the *\$watch()* function as a means to observe and react to changes in the Model
- There are some things to be aware of, like "unnecessary" system calls during the watch initialization sequence that you should handle programmatically, if needed
- You should be aware of performance implications of having complex observable expressions and using the *deepWatchFlag* optional parameter



## Chapter 16 - Communicating with Web Servers

---

### *Objectives*

In this chapter, we will

- Introduce the \$http service
- Review \$http shortcut methods
- Review ways to add payload and parameter to HTTP requests
- Provide a short overview of testing with ngMock



## 16.1 The \$http AngularJS Service

- Communication with remote HTTP Servers in Angular is done using the ***\$http*** service
- The ***\$http*** service leverages browser's *XMLHttpRequest* object (for making XHR calls) or uses the *JSONP* communication technique
- ***\$http*** is a robust and secure service that has built-in protection against both JSON vulnerabilities and XSRF



## 16.2 The Promise interface

- The AngularJS HTTP connection API implements what is commonly known as the Promise interface recently popularized and implemented by various JavaScript libraries
- The main idea behind the Promise API is to bring order to asynchronous calls that need to be made in some logical sequence
  - ◇ This is done through call chaining and consistent error handling (see *Notes*)
- AngularJS calls to the remote server are asynchronous in nature with the server response expected to arrive to the browser at some time in future
- The Promise interface guarantees a predictable handling of asynchronous calls



### 16.3 \$http Set Up

- The \$http service object can be used as a function to make Ajax requests or you can use one of the shortcut methods like `get()` and `post()` of the service object.
- All of these methods return a promise object that supports three methods:
  - ◇ **success**(func) – Registers a callback function that is called after a successful response. A server response status code between 200 and 299 is considered a success.
  - ◇ **error**(func) – Registers a callback function that is called in case of any error.
  - ◇ **then**(success, error) – Registers a success and error callback and is used for more advanced cases.
- **Note:** *\$http* will transparently follow any server redirect responses (3xx HTTP status codes)



## 16.4 \$http Function Invocation

- The typical invocation of the \$http function looks as follows:

```
$http({method: 'GET', url: 'theUrl'}).
 success(function(data, status, headers, config) {
 /* this callback method will be called by Angular asynchronously as the server response
becomes available */
 }).
 error(function(data, status, headers, config) {
 /* this callback method will be called by Angular asynchronously in case of an error
*/
 });
```

where

`{method: 'GET', url: 'theUrl'}` is the HTTP request configuration object in JSON format

- **Note** the '.' notation used in attaching the *success* and *error* callback functions to the *\$http* function object



## 16.5 Callback Parameters

- The success and error callback functions receive these parameters.

Configuration Parameter	Description
<b>data</b>	HTTP response data (payload)
<b>status</b>	The standard HTTP response status code
<b>headers</b>	A function that accesses HTTP response headers
<b>config</b>	The original config object used to make the HTTP request



## 16.6 Request Configuration Properties

- The configuration object passed to the `$http()` function can have these properties.

Configuration Parameter	Description
<b>method</b>	HTTP method to be issued
<b>url</b>	The target URL
<b>params</b>	Parameters to be added to the URL query string
<b>headers</b>	Additional HTTP headers to be added to a request
<b>timeout</b>	Timeout (in milliseconds) after which an AJAX (XHR) request will be terminated
<b>cache</b>	Enables XHR GET request caching



## 16.7 Shortcut Methods

- You may also use simpler forms of *\$http* for GET, POST and PUT requests (called shortcut methods):

```
$http.get('theUrl').success(success_fn).error(error_fn);
$http.post('theUrl', data).success(success_fn).error(error_fn);
 where
```

*success\_fn* and *error\_fn* are user-defined handlers for respective callbacks

- **Note:** POST and PUT methods take any valid JavaScript object as its *data* payload object





## 16.8 Complete List of Shortcut Methods

- `$http.get`
- `$http.head`
- `$http.post`
- `$http.put`
- `$http.delete`
- `$http.jsonp`
- `$http.patch`

See [https://docs.angularjs.org/api/ng/service/\\$http](https://docs.angularjs.org/api/ng/service/$http) for details



## 16.9 Passing Parameters to \$http GET Requests

- For GET HTTP request, parameters use the *params* key in configuration object which should point to a map of keys and values that will be translated to URL parameters. For example:

```
$http.get('theUrl',{params:{key1: 'val1', key2: 'val2'}})...
```

request will build the following URL:

```
theURL?key1=val1&key2=val2
```



## 16.10 Working with JSON Response

- If the response Content-Type is "application/json" or if the response starts with "{" or "[" then system will convert the response to a JavaScript object and pass that to the success callback method.
- Example response:

```
{
 "name": "House of Haunted Souls",
 "released": 1973,
 "genre": ["horror", "comedy"]
}
```

- Example success callback:

```
$http.get("movies/1029")
 .success(function(data) {
 $scope.movie = data; //Converted to object
 console.log("Received movie: " + $scope.movie.name);
 });
```



## 16.11 Making a POST Request

- For POST HTTP request, you set the request body using the second parameter (after the URL) as shown in the example below:

```
var myPostData = {key1: 'val1', key2: 'val2'}
$http.post('theUrl',myPostData)...
```

- AngularJS does a JSON POST (and not a form POST). Which means, the request body is a JSON document (and not a name value pair separated by &). The content type of the request is "application/json".
  - ◇ You will need to write a web service to process the request.
- **Note:** AngularJS reserves properties beginning with a dollar sign ('\$') for its own private use and its default marshaling (conversion) of JavaScript objects into JSON will ignore any user property that starts with a '\$'. To work around this constraint, use the `JSON.stringify` method to populate the data object



## 16.12 Combining \$http POST Request Data with URL Parameters

- For RESTful web services requests, you may be required to combine data to be POST'ed with URL parameters (to identify a resource by id)
- In this case, simply add the *params* configuration object as we did in the *\$http GET Requests* slide above to the *\$http* function invocation string:

```
var myPostData = {key1: 'val1', key2: 'val2'}
var config = {params: {id: '123'}}; // will append ?id=123
$http.post('theUrl',myPostData, config) ...
```



### 16.13 The then() Method of the Promise

- The then() method of the promise object provides an alternate way to register the success and error callback functions. It takes two callbacks as parameters:
  - ◇ The success callback.
  - ◇ The error callback.

```
$http.get("movies/1029").then(
 function(response) {
 $scope.movie = response.data; //The payload
 },
 function(response) {
 //Handle error
 });
```



## 16.14 The Response Object

- The response object passed to the success and error callback by the `then()` method has these properties:
  - ◇ `data` – The HTTP response body (the payload).
  - ◇ `status` – The reply status.
  - ◇ `headers` – A function to obtain a response header.
  - ◇ `config` – The configuration object used to make the request.

```
$http.get("movies/1029").then(
 function(response) {
 $scope.movie = response.data; //The payload
 var type = response.headers("Content-Type");
 var status = response.status;
 });
```



## 16.15 Setting Up HTTP Request Headers

- By default, AngularJS adds the following HTTP headers to all requests:
  - ◊ `Accept: application/json, text/plain, /`
  - ◊ `X-Requested-With: XMLHttpRequest`
- You can also add your own headers by specifying the *headers* configuration object key, e.g.

```
headers: {'Content-Type': 'application/x-www-form-urlencoded'}
```

- **Note:** You can also add your special headers to the default *\$http* headers group so that they will be used with all outgoing HTTP requests (we are not reviewing this technique here)





## 16.16 Caching Responses

- By default, AngularJS does not cache HTTP responses for your GET requests
- When it make sense and you would like to enable caching of GET responses (e.g. you are OK with seeing stale server data), it is quite simple to do:

Just set the `cache : true` configuration parameter, e.g.

```
$http.get('theURL', {cache: true})...
```

- **Note:** If there are several concurrent GET requests for the same URL and you have caching enabled for all of them, only one GET request will be sent to the server while the remaining requests be blocked and fulfilled from cache filled with server response data from the first request



## 16.17 Setting the Request Timeout

- You can set the *timeout* configuration parameter to specify the time in milliseconds you can wait before the request should be treated as timed out

- For example

`timeout: 3000`

will set the timeout to 3 seconds



## 16.18 Unit Testing with ngMock

- The *ngMock* module provides support for unit testing Angular services
- It is used in conjunction with the *\$httpBackend* object which is a fake (stubbed) HTTP backend implementation suitable for unit testing applications
- To start using it:

- ◇ First, get and include the *angular-mocks.js* file in your HTML page:

```
<script src="angular-mocks.js">
```

- ◇ Note: You can get the needed file version from Google CDN  
(*ajax.googleapis.com/ajax/libs/angularjs/<version>/angular-mocks.js*)
- ◇ Then, load the ngMock service as a dependency:

```
angular.module('app', ['ngMock']);
```

- You all set and ready to go



## 16.19 Writing Unit Tests

- Basically, your tests will follow the following steps:
  - ◇ `$httpBackend.expectGET('theURL').respond(some object);` // *set the expected behavior*
  - ◇ `$http.get(...);` // *use \$http object*
  - ◇ `$httpBackend.flush();` // *Flush each pending request using trained responses*
- **Note:** We are not reviewing AngularJS Unit Testing in more detail here as it goes beyond the scope of this module



## 16.20 Summary

- Communication with remote HTTP Servers in Angular is done using the *\$http* service
- The *\$http* service leverages browser's XMLHttpRequest object or uses the JSONP communication technique
- To simplify command syntax, \$http supports a number of shortcut methods, e.g.
  - ◇ \$http.get
  - ◇ \$http.head
  - ◇ \$http.post
  - ◇ \$http.put
  - ◇ \$http.delete
  - ◇ at all
- You can configure a variety of parameters to control your HTTP requests, like request timeout and specific HTTP headers
- Angular provides techniques for unit testing *\$http* requests

## Chapter 17 - Custom Directives

---

### *Objectives*

Key objectives of this chapter

- What are directives?
- Defining custom directives.
- Using custom directives.



## 17.1 What are Directives?

- An AngularJS directive is a JavaScript object that encapsulates reusable view layer logic.
  - ◇ They generate HTML markup, manipulate DOM elements and apply styles.
  - ◇ Avoid writing any complicated business logic in directives. Do so in services.
- Directives, like services are singleton. Only one instance of a directive is created per page.
  - ◇ Be careful about storing any state in directives.
- AngularJS ships with a number of directives like ng-app and ng-repeat. You can define your own.



## 17.2 Directive Usage Types

- Depending on how a directive is defined, you can use it in a page in these ways:
  - ◇ Element directive. Such as `<my-directive></my-directive>`.
  - ◇ Attribute directive. Such as `<div my-directive>`.
  - ◇ CSS directive. Such as `<div class="my-directive">`
  - ◇ Comment directive. Such as `<!-- directive: my-directive -->`
- By default a directive can be used as an element or attribute. This is also the best practice.
  - ◇ You can restrict how a directive can be used as a part of its definition.
- When used as an element, you must provide the closing tag. Otherwise, some browsers may ignore it.
  - ◇ Good: `<my-directive></my-directive>`
  - ◇ Bad: `<my-directive/>`





### 17.3 Directive Naming Convention

- A directive is registered using its normalized name. Such as "myDirective".
- In the HTML markup, it must be referred to using one of these names:
  - ◇ "my-directive". As in `<div my-directive>` or `<my-directive>`. This is the common approach.
  - ◇ "data-my-directive". As in `<div data-my-directive>`. This is the HTML5 way to add custom attributes.
  - ◇ "my:directive".
  - ◇ "my\_directive".
  - ◇ "x-my-directive".
- It is a good idea to prefix custom directives so that the names don't collide with other toolkits. For example, "pcsGallery" instead of just "Gallery".



## 17.4 Defining a Custom Directive

- Use the **directive()** method of a module to register a custom directive.
- The directive() method takes as input:
  - ◇ The normalized name of the directive.
  - ◇ A factory function that returns an instance of the directive.

```
angular.module("SimpleApp", [])
.directive("myDirective", function($http, $cookies) {
 return {
 template: "<h1>Hello World</h1>"
 };
});
```

- You can inject services into the factory function.



## 17.5 Using the Directive

```
<div ng-app="SimpleApp">
 <my-directive></my-directive>
 <my-directive>Body ignored</my-directive>
</div>
```

- Each `<my-directive>` tag will be replaced by its template: `"<h1>Hello World</h1>"`.
- The directive creation function is called only once and only one instance of the directive is created.
- By default, the body of a custom directive is discarded. The template is the sole provider of content. Advanced components can enable the use of the body and children elements.



## 17.6 Scope of a Directive

- By default, a directive does not create its own scope. It simply works in the context of whatever scope it was used in the page.
- In this example, the directive uses the scope of the TestCtrl controller.

```
angular.module("SimpleApp", [])
.directive("myDirective", function() {
 return {
 template: "<h1>Hello {{planet}}</h1>"
 };
})
.controller("TestCtrl", function($scope) {
 $scope.planet = "Mars";
});

<div ng-controller="TestCtrl">
 <my-directive> </my-directive>
 <input type="text" ng-model="planet"/>
</div>
```



## 17.7 Isolating Scope

- By default a directive works with the scope of the parent controller. This has two major problems:
  - ◇ The directive has to know the names of the properties in the scope (planet in our example). This tightly couples the directive with the controller making it less reusable.
  - ◇ The directive can not be used more than once in any meaningful way. For example, if the controller has different planets in scope, how can we use the directive for different planets?
- The solution is for the directive to have its own scope and provide a way to copy certain properties from the parent scope to the scope of the directive. This is called scope isolation.



## 17.8 Example Scope Isolation

```
.directive("myDirective", function() {
 return {
 template: "<h1>Hello {{planet}}</h1>",
 scope: {
 planet: "="
 }
 };
})

.controller("TestCtrl", function($scope) {
 $scope.planet1 = "Mars";
 $scope.planet2 = "Earth";
});

<div ng-controller="TestCtrl">
 <my-directive planet="planet1"> </my-directive>
 <my-directive planet="planet2"> </my-directive>
</div>
```



## 17.9 Using External Template File

- If a directive needs to generate complex HTML markup, consider creating a separate HTML template file.

```
.directive("myDirective", function() {
 return {
 templateUrl: "my-directive.html",
 scope: {
 planet: "="
 }
 };
})
```

```
<!-- my-directive.html -->
```

```
<h1>Hello {{planet}}</h1>
```



## 17.10 Manipulating a DOM Element

- For a directive to manipulate an existing DOM element, use the link function. The following directive sets the border style:

```
.directive("myBordered", function() {
 return {
 link: function(scope, element, attrs) {
 element.attr("style", "border: thin solid black");
 }
 };
});
```

```
<div my-bordered>
 <h2>Hello World</h2>
 <p>Now is the winter of our discontent.</p>
</div>
```





## 17.11 The Link Function

- The link function gets called after the directive instance is associated with a DOM element.
- The function receives these arguments:
  - ◇ The scope of the directive.
  - ◇ A jQuery like object representing the DOM element where the directive is applied. It provides a subset function of jQuery such as `attr()` and `on()`. This is known as the jqLite API. It is basically an array of DOM elements.
  - ◇ A map containing all the attributes of the element.



## 17.12 Event Handling from a Link Function

- A directive can handle DOM events by attaching handlers from the link function. The following will show a border when mouse enters an element.
  - ◊ To attach a handler, you can use the `on()` method made available by jqLite, or, you can use the `addEventListener()` standard DOM method.

```
.directive("liveBorder", function() {
 return {
 link: function(scope, element, attrs) {
 element.on("mouseenter", function() {
 element.attr("style", "border: thin solid black");
 });
 element.on("mouseleave", function() {
 element.attr("style", "");
 });
 }
 };
});
```

```
<div live-border>Hello</div>
```



### 17.13 Wrapping Other Elements

- By default, if a directive provides its own template then the body of the element is ignored. To include the body of the directive in the HTML markup output by its template, you will need to use the **transclusion** feature.

```
.directive("myQuote", function() {
 return {
 transclude: true,
 template: "<p>This was said:</p>" +
 "<div ng-transclude style='margin-left: 50px; padding-left:
10px; border-left: thick solid gray'></div>",
 };
 });
});
```

This was said:

**Hello World**

Now is the winter of our discontent.



```
<bb-quote>
 <h2>Hello World</h2>
 <p>Now is the winter of our discontent.</p>
</bb-quote>
```



## 17.14 Summary

- Reusable view layer logic is captured in custom directives.
- By default, a custom directive uses the same scope as the parent which reduces its reusability. Most directives will need to setup its own scope using scope isolation.
- The link function of a directive is called right after the singleton instance of a directive is associated with a DOM element. You can manipulate the element or attach event handlers from the link function.
- To include the body of the directive in the HTML markup output by its template, you will need to use the transclusion feature.

## Chapter 18 - AngularJS Services

---

### *Objectives*

In this chapter we will learn:

- What are services?
- How to create your own services?
- How to use services from a controller?



## 18.1 Introduction to Services

- AngularJS Services are JavaScript objects that contain reusable business logic.
- Services form the model layer. Generally speaking avoid writing view layer logic such as DOM manipulation.
- When core business logic resides in the server and exposed as web services, AngularJS services act as proxy to those web services. Controllers simply call methods of these services without knowing anything about web services.
- Service instances are injected at runtime to their callers.
  - ◊ This means, during testing, you can inject an alternate version of a service that is designed to simplify testing.
- A service instance is lazily created only when needed to satisfy an injection request.
- Services are singleton. Within a page, maximum one instance of a service is created. Hence, be careful about making the services stateful.



## 18.2 Defining a Service

- AngularJS ships with a number of ready to use services. For example, `$http` is used to make Ajax requests. In addition to that, you can define your own services.
- There are three different ways you can define a service:
  - ◇ Using the module's **factory()** method.
  - ◇ Using the module's **service()** method.
  - ◇ Using the module's **provider()** method.
- Each of these approaches are fairly the same in the end. The provider approach has certain advantages as we will see.





### 18.3 The factory() Method Approach

- The factory() method of a module is used to register a factory function that returns a service instance. The factory() method takes as input:
  - ◇ The name of the service that will be used later for injection.
  - ◇ A factory function that creates, initializes and returns an instance of the service.

```
angular.module("SimpleApp", [])
.factory("mySvc", function($http) {
 var svc = {}; //New instance
 svc.sayHello = function(planet) {
 console.log("Hello " + planet);
 }

 return svc; //Return the service instance
})
```

- A service exposes its API through a set of methods, like sayHello() here.
- You can inject other services in the argument of the service creation function. For example, \$http above.



## 18.4 The service() Method Approach

- The service() method of a module is used to register a JavaScript constructor function.
- The service() method takes the name of the service and the constructor function as argument.

```
angular.module("SimpleApp", [])
.service("mySvc", function($http, $cookies) {
 this.sayHello = function(planet) {
 console.log("Hello " + planet);
 }
});
```

- You can inject other services in the argument of the constructor function. For example, \$http and \$cookies above.



## 18.5 The provider() Method Approach

- You use the provider() method to register a constructor function. However, this constructor is used by AngularJS to create a provider instance. The \$get property of the provider instance needs to point to a service creation function that returns a service instance.
- The provider() method takes as argument the name of the service and the provider constructor function.

```
angular.module("SimpleApp", [])
.provider("mySvc", function() {
 this.$get = function($http) {
 var svc = {}; //Service instance
 svc.sayHello = function(planet) {
 console.log("Hello " + planet);
 };
 return svc;
 };
});
```

- Here, the provider instance will be registered with the name mySvcProvider and the service instance will be registered with the name mySvc.
- You can not inject any service in the provider constructor but you can do so in the \$get function.



## 18.6 Using a Service

- No matter how a service is defined, it is used in the same fashion.
- Any controller or service that needs to use a service needs to inject the service instance using its registered name. For example:

```
.controller("TestCtrl", function($scope, mySvc) {
 $scope.dump = function() {
 mySvc.sayHello("Saturn");
 }
});
```

```
<div ng-controller="TestCtrl">
 <button ng-click="dump()">Test</button>
</div>
```



## 18.7 Configuring a Service using its Provider

- Certain parameters that affect the behavior of a service can be configured at module configuration phase before the service is created.
- In this example, we use the greeting variable as a configurable parameter.

```
angular.module("SimpleApp", [])
.provider("mySvc", function() {
 var greeting = "Hello";
 this.setGreeting = function(g) {
 greeting = g;
 };
 this.$get = function() {
 var svc = {}; //Service instance
 svc.sayHello = function(planet) {
 console.log(greeting + " " + planet);
 };
 return svc;
 };
});
))
.config(function(mySvcProvider) {
 mySvcProvider.setGreeting("Hola");
});
```



## 18.8 Summary

- A service is a JavaScript object that contain reusable business logic.
- The API of a service is exposed as a set of methods in the service object.
- Service instances are singleton. It is safe to store configuration parameters as state but be careful about storing any other business data as state.
- Services are define using one of three different methods of a module – `factory()`, `service()` and `provider()`.
- The provider approach gives a little bit more flexibility for complex services. You are able to change various configurable settings at the module configuration phase.

## Chapter 19 - Unit Test using Jasmine

---

### *Objectives*

Key objectives of this chapter

- Introduction to Node JS
- What is Jasmine
- Running Jasmine
- Running Tests Using Jasmine
- Unit Test Example Explained
- End-to-End Testing with Protractor and Jasmine



## 19.1 Introduction to Node JS

- Node.js is a set of asynchronous libraries, built on top of the Google V8 Javascript Engine.
- Node.js has many things, but mostly it's a way of running JavaScript outside the web browser.
- Aside from its use in testing Angular applications, Node is used to produce highly scalable servers using an architecture called an event loop, that makes concurrent programming both easy and safe.
- Node Package Manager, npm allows to install of third-party packages and dependences.
- Node comes with assertions built in, and all testing frameworks build on the Assert module.
- There are at least 30 different testing frameworks to use.





## 19.2 What is Jasmine

- Jasmine is a behavior-driven development (BDD) JavaScript testing framework.
- Jasmine allows tests to be written independent of the DOM or other JavaScript frameworks.
- You can run Jasmine in a browser, or in Node.js.
- Jasmine API includes features such as:
  - ◇ A more natural BDD syntax for organizing the test logic than JUnit style assertion test frameworks
  - ◇ Asynchronous testing
  - ◇ Mocks
  - ◇ Spies
  - ◇ Easy to create custom matchers (matchers are used in evaluating pass or fail criteria in tests).
  - ◇ Ability to share or isolate behaviors between tests within a spec encapsulating parts of your spec.
  - ◇ Continuous integration support
- Jasmine is a unit testing framework, ideally suited to run unit tests and regression tests.



- Jasmine is not 100% of your integration tests, because Jasmine only tests the Javascript code on one system at a time.
- There are other tools such as Protractor, karma etc ... available for end to end testing.



### 19.3 Running Jasmine

- Many options available for running your Jasmine tests.
  - ◇ Tests can be run manually, in browsers.
  - ◇ Headless outside a browser, via a continuous integration server (CI), and so on.
- Unit tests are written the same way regardless of how they are run.
- Some examples of how to run them are:
  - ◇ SpecRunner.html: Available in the standalone version.
  - ◇ You can run tests from a local browser. No server required.
  - ◇ Inside an IDE (for example using the Jasclipse plugin)
  - ◇ JS-Test-Driver: (A popular JavaScript test framework) Able to run tests in multiple browsers.
    - You can run JS-Test-Driver from your IDE, command line, and so on.
  - ◇ Java with Maven: Using the Jasmine Maven plugin
  - ◇ Run headless in Rhino, Envy, or Jasmine-headless-webkit: for testing without a browser. Because the headless test does not attach to a browser instance, it can run faster.
  - ◇ Using Node.js through the jasmine-node module.



- ◇ Using multiple browsers using the Karma package.
- Jasmine Folder structure
  - ◇ lib - most important directory, which contains the Jasmine test framework code
  - ◇ src - testing files
  - ◇ spec - test Specification files
  - ◇ SpecRunner.html - is the file used to invoke the unit tests.



## 19.4 Running Tests

- **Suites:** describe Your Test
- A test suite begins with a call to the global Jasmine function `describe` with two parameters: a string and a function.
  - ◇ The string is a name or title for a spec suite – usually what is being tested.
  - ◇ The function is a block of code that implements the suite.
- **Specs:**
  - ◇ Specs are defined by calling the global Jasmine function `'it'`, which, like `describe` takes a string and a function.
  - ◇ The string is the title of the spec and the function is the spec, or test.
  - ◇ A spec contains one or more expectations that test the state of the code.
  - ◇ An expectation in Jasmine is an assertion that is either true or false.
  - ◇ A spec with all true expectations is a passing spec.
  - ◇ A spec with one or more false expectations is a failing spec.





## 19.5 Example Explained

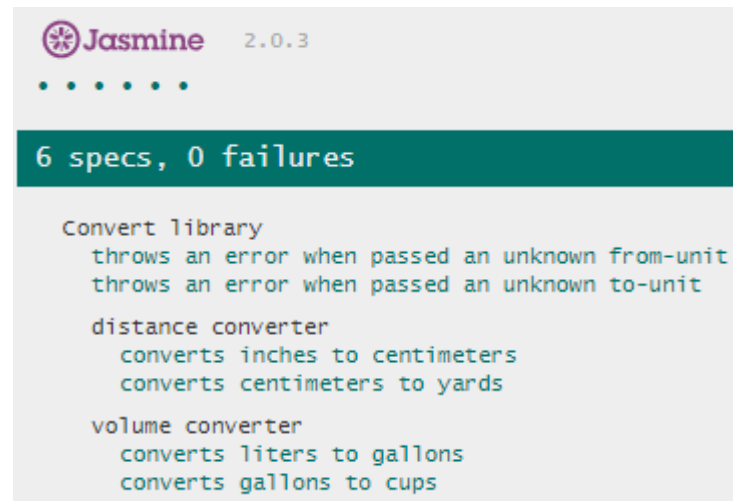
- Script file to test(convert.js)
- Creating suites:(convertSpec.js)

```
describe("distance converter", function () {
 it("converts inches to centimeters", function () {
 expect(Convert(12, "in").to("cm")).toEqual(30.48);
 });
 it("converts centimeters to yards", function () {
 expect(Convert(2000, "cm").to("yards")).toEqual(21.87);
 });
});
```

- **Test Runner HTML**(specRunner.html)
  - ◇ Remove the links to the sample project files in SpecRunner.html and add these lines:
  - ◇ Add these two entries to the standard HTML template comes with Jasmine.
    - `<script src="src/convert.js"></script>`



- `<script src="spec/convertSpec.js"></script>`
- ◇ Open `SpecRunner.html` file using Chrome browser. It should execute the tests.
- ◇ Expected result:









## 19.6 End-to-End Testing with Protractor

- For Integration testing we write end-to-end tests, performing a sequence of a work flow.
- Integration testing can be done by automating tests using tools like Selenium with Protractor or Karma.
- Protractor is a Node.js program.
- You can install Protractor using npm, which comes with Node.js.
- Version of Node.js should be greater than v0.10.0.
- By default, Protractor uses the Jasmine test framework for its testing interface.
- To control the browser, Protractor can use servers such as Selenium or Karma.
- In order to run local standalone Selenium Server, you need to have the Java Development Kit (JDK) installed.





## 19.7 Writing E2E Test with Protractor

- Protractor needs two files to run: a spec file and a configuration file.
- Example of Spec JS:

```
//Spec.js
describe('E2E form', function() {
 beforeEach(function() {
 browser.get('http://localhost/unit-test/app/SimpleE2ETest.html');
 });
 it('display initial text', function() {
 expect(element(by.binding('yourName')).getText()).toEqual("Hello !");
 });
 it('greets Protractor',function(){
 element(by.model('yourName')).sendKeys("WebAge Instructor");
 expect(element(by.binding('yourName')).getText()).toEqual("Hello WebAge Instructor!");
 });
});
```

- Example of Conf JS:



```
// conf.js
exports.config = {
 seleniumAddress: 'http://localhost:4444/wd/hub',
 specs: ['SimpleSpec.js']
}
```

- Running the test :
  - ◇ protractor conf.js



## 19.8 Summary

- What is integration testing
- Integration testing tools
- Testing with Protractor and Jasmine