HTML5, CSS, BOOTSTRAP, JAVASCRIPT, TYPESCRIPT, JSON, JQUERY, ANGULAR.JS, AND NODE.JS:

XXXStart Handling Events p.116.

XXXCompleted Working with CSS. from end of p.196 to end of p.203.

XXXXStart CHAPTER 7 SportsStore: A Real Application. p.109.

XXXStart with JavaScript Functions, p.322, Deitel, Fouth Edition.

Atom.io Text Editor and Live Server for Testing HTML Pages with CSS and JavaScript Scripting:

*Download the atom.io text editor from: https://atom.io, and install atom.io.

*To activate the atom.io live server , go to menu: File, Settings, Install, and search for and install "atom-live-server".

*To add a project, choose menu: File, Add Project Folder, browse the project folder that you want installed.

The HTML Document Structure:
*There are some key elements that define the basic structure of any HTML document: the DOCTYPE, html, head, and body elements. The following shows the relationship between these elements with the rest of the content removed.

```
<!DOCTYPE html>
<html>
<head>
  ...head content...
</head>
<body>
  ...body content...
</body>
</html>
```

Figure: The basic structure of an HTML document.
*Each of these elements has a specific role to play in an HTML document:
*(1) DOCTYPE Indicates the type of content in the document. The DOCTYPE element tells the browser that this is an HTML document and, more specifically, that this is an HTML5 document. Earlier versions of HTML required additional information. For example, following is the DOCTYPE element for an HTML4 document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

*(2) The html element denotes the region of the document that contains the HTML, which is usually the entire document. This element always contains the other two key structural elements: head and body.
*(3) The head element denotes the region of the document that contains metadata for the document, which are one or more elements that describe or operate on the content of the document but that are not directly displayed by the browser. Examples of metadata elements in the head section include: title, script, and style. The title element is the most basic and the contents of this element are used by browser to set the title of the window or tab, and all HTML documents are required to have a title element. The other two elements are described later.
*(4) The body element denotes the region of the HTML document that contains the content in an HTML document. These are the elements that the browser will display to the user and that the metadata elements, such as script and style, operate on.

Anatomy of HTML Elements:
*At the heart of HTML is the HTML element, which tells the browser what kind of content each part of an HTML document represents. Following is an HTML element:

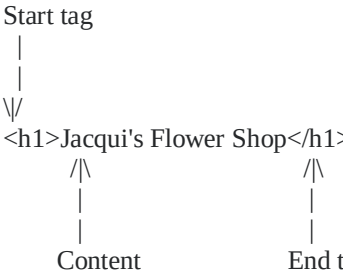

Figure: The anatomy of a simple HTML element.
*This HTML element has three parts: the start tag, the end tag, and the content, as illustrated above.
*The name of this HTML element, also referred to as the tag name or just the tag, is h1, and it tells the browser that the content between the tags should be treated as a top-level header. We create the start tag by placing the tag name in angle brackets, the < and > characters. We create the end tag in a similar way, except that you also add a / character after the left-angle bracket (<).
*The latest version of HTML is known as HTML. HTML5 has more than 100 elements, and each of them has its own purpose and functionality. A brief description of some elements are provided below:

HTML Element	Description
title	Denotes the title of the document; used by the browser to set the title of the window or tab used to display the document's content.
h1	Denotes a header.
button	Denotes a button; often used to submit a form to the server.
img	Denotes an image.
form	Denotes an HTML form, which allows you to gather data from the user and send them to a server for processing.
input	Denotes an input field used to gather a single data item from the user, usually as part of an HTML form.
div	A generic element; often used to add structure to a document for presentation purposes.
script	Denotes a script, typically JavaScript, that will be executed as part of the document.
style	Denotes a region of CSS settings.

Table: Some HTML elements.

Attributes of HTML Elements: We can provide additional information to the browser by adding attribu-tes to the HTML elements. The following shows an HTML element with an attribute:
<label for="aster">Aster:</label>
Figure: Defining an attribute in an HTML element.
*This is a label element, and it defines an attribute called for. Attributes are always defined as part of the start tag. This attribute has a name and a value. The name is for, and the value is aster. Not all attributes require a value; just defining them sends a signal to the browser that you want a certain kind of behavior associated with the element. The following shows an HTML element with such an attribute:
<input name="snowdrop" value="0" required>
Figure: Defining an attribute that requires no values.
*This element has three attributes. The first two, name and value, are assigned a value. The names of these attributes are name and value. The value of the name attribute is snowdrop, and the value of the value attribute is 0. The third attribute is just the word required. This is an example of an attribute that
doesn't need a value.

*The id and class Attributes of HTML Elements: Two attributes are particularly important: the id and class attributes. One of the most common tasks we need to perform with jQuery is to locate one or more elements in the document so that we can perform some kind of operation on them.
*The id and class attributes are useful for locating one or more elements in the HTML document.
*The id Attributes of HTML Elements to Identify a Specific HTML Element: We use the id attribute to define a unique identifier for an element in a document. No two elements are allowed to have the same value for the id attribute. Using the id value lets us find a specific element in the document. The following shows a simple HTML document that uses the id attribute.

```
<!DOCTYPE html>
<html>
<head>
  <title>Example</title>
</head>
<body>
  <h1 id="mainheader">Welcome to Jacqui's Flower Shop</h1>
  <h2 id="openinghours">We are open 10am-6pm, 7 days a week</h2>
  <h3 id="holidays">(closed on national holidays)</h3>
</body>
</html>
```

Figure: Using the id attribute.
*We've defined the id attribute on three of the elements in the document. The h1 element has an id value of mainheader, the h2 element has an id value of openinghours, and the h3 element has an id value of holidays.
*The class Attributes of HTML Elements to Identify a Group of HTML Elements: The class attribute arbitrarily associates elements together. Many elements can be assigned to the same class, and elements can belong to more than one class, as shown in the simple HTML document that uses the class attribute.

```
<!DOCTYPE html>
<html>
<head>
  <title>Example</title>
</head>
<body>
  <h1 id="mainheader" class="header">Welcome to Jacqui's Flower Shop</h1>
  <h2 class="header info">We are open 10am-6pm, 7 days a week</h2>
  <h3 class="info">(closed on national holidays)</h3>
</body>
</html>
```

Figure: Using the class Attribute
*Here, the h1 element belongs to the header class, the h2 element belongs to the header and info classes, and the h3 element belongs just to the info class. As seen, we can add an element to multiple classes just by separating the class names with spaces.

HTML Element Contents: Elements can contain text, but they can also contain other elements. Following is an example of an element that contains other elements, as nested elements:

```
<div class="dcell">
  <img src="rose.png"/>
  <label for="rose">Rose:</label>
  <input name="rose" value="0" required>
</div>
```

*The div element contains three other elements: an img, a label, and an input element.
*We can define multiple levels of nested elements, not just the one level shown here. Nesting elements like this is a key concept in HTML because it imparts the significance of the outer element to those contained within.
*We can mix text content and other elements, as follows:

```
<div class="dcell">
  Here is some text content
  <img src="rose.png"/>
  Here is some more text!
```

<input name="rose" value="0" required>

*HTML Elements with No (Void) Contents: The HTML specification includes elements that may not contain content. These are called void or self-closing elements and they are written without a separate end tag. Following is an example of a void element:

*A void element is defined in a single tag, and you add a / character before the last angle bracket, the > character. Strictly speaking, there should be a space between the last character of the last attribute and the / character, as follows:

*However, browsers are tolerant when interpreting HTML and we can omit the space character.
*Void elements are often used when the element refers to an external resource. In this case, the img element is used to link to an external image file called rose.png.

Multipurpose Internet Mail Extension (MIME) Type: describes a file's content. CSS documents use the MIME type text/css, GIF images use type image/gif, JavaScript uses type text/javascript, and so on.

Cascading Style Sheets 2.0 (CSS 2.0) for XHTML Elements and XHTML Documents:

*For a reference of CSS properties, see: https://www.w3schools.com/cssref/default.asp

*Cascading Style Sheets, which are used to control the presentation of HTML elements. Cascading Style Sheets (CSS) are the means by which you control the appearance, more properly known as presentation, of HTML elements.

*CSS 2.0 is a W3C technology that allows document authors to specify the presentation of HTML/ XHTML elements (including the <body> element backgrounds) on a web page, such as fonts, spacing, colors, etc., separately from the structure of the document, such as section headers, body text, links, etc. This separation of structure of document from presentation of document simplifies maintaining and modifying a web-page, and provides uniformity of presentation styles across multiple pages.

*When the browser displays an element on the screen, it uses a set of properties, known as CSS properties, to work out how the element should be presented. There are more than 130 CSS properties, each of which controls an aspect of an element's presentation. As there are too many CSS properties, we're unable to list them all, and a few of the CSS properties are listed below:

Property	Description
Color	Sets the foreground color of the element, which typically sets the color of text.
background-color	Sets the background color of the element.
font-size	Sets the size of the font used for text contained in the element.
Border	Sets the border for the element.

Table: Some CSS properties.

*CSS consists of a set of CSS properties and their values that are applied to XHTML elements. All CSS rules, either in inline styles or in embedded style sheets or in linked external style sheets, use the same syntax for CSS rules: the CSS property is followed by a colon (:) and the value of the property. Multiple CSS properties and their values are separated by semicolons (;). CSS style sheets enable a separation between the CSS rules and the XHTML elements that it styles.

***The Default Values of CSS Properties**: The browser still ha4s to display the elements even if you haven't provided values for the CSS properties, and so each element has a style convention; a default value that it uses for

CSS properties when no other value has been set in the HTML document. The HTML specification defines style conventions for elements, but browsers are free to vary them, which is why so you will see variations in the style convention between, say, Google Chrome and Internet Explorer.

XXXXXXXStand

XXXXXXXEnd

CSS Cascading Style Rules: CSS style rules may be defined by an author or a user-agent, such as a web-browser. An author is the person that writes the document, and the user-agent is the program used to render and display the document. CSS styles "cascade", or flow together, such that the appearance of XHTML elements on a page results from combining styles defined by the author and the user-agent. Styles defined by the author takes precedence over styles defined by the user-agent. Also, most styles defined for parent elements are also inherited by child (nested) elements.

CSS Validator: The W3C provides a CSS code validator located at jigsaw.w3.org/css-validator/. It is a good idea to validate all CSS code with this tool to make sure that your code is correct and works on as many browsers as possible.

CSS Inline Styles for XHTML Elements with the style Attribute: You can declare document styles in several ways. Inline style allows an individual element's format using the XHTML attribute style. Inline styles override any other styles applied using other techniques.

*For example, the following applies inline styles to a p element to alter the font size and color:

Hello World .

*Each CSS property, such as font-size, color, is followed by a colon (:) and a value for the property; and multiple CSS properties and their values are separated by semi-colons.

*Note that CSS inline styles do not truly separate presentation from content. To apply similar styles to multiple elements, use embedded style sheets or external style sheets. Style sheets are a convenient way to create a document with a uniform theme.

CSS Embedded Style Sheets for XHTML Elements in the XHTML Document in XHTML Documents Head Section, <style> Element: A second technique for using style sheets is CSS embedded style sheets, which enable a separation between the CSS code and the XHTML elements that it styles. Embedded style sheets enable you to embede an entire CSS document in an XHTML document's <head> section inside a style element.

Example: A CSS embedded style sheet inside the XHTML documents <head> section, style element. CSS comments may be placed in any type of CSS code: inline styles, embedded styles sheets and external style sheets, and always start with a /* and end with a */. Text between these delimiters is ignored by the browser.

```
<html xmlns = "http://www.w3.org/1999/xhtml">

  <head>

    <title>CSS Emedded Style Sheets Example</title>

    <!-- The style element declares a CSS embedded style sheet -->
```

```
   <style type = "text/css">
      body  { background-image: url(logo.gif); background-color: #eeeeee; font-family: arial, helvetica }
      h1  { font-family: tahoma, helvetica, sans-serif }
      p   { font-size: 12pt; font-family: arial, sans-serif }
      /* Declare CSS style classes that begin with periods that can be applied to any XHTML elements */
      .favoritestyle  { font-size: 14pt; color: #666ff }
   </style>
  </head>
  <body>
    <!-- The XHTML element class attribute applies the .favoritestyle style -->
    <h1 class = "favoritestyle">Hello World</h1>
    <p> This is an introduction to the study of ...</p>
  </body>
</html>
```

*The style element inside the XHTML document's <head> section defines the embedded style sheet. CSS rules in embedded style sheets are enclosed in braces (the { and } characters) and are preceded by a selector, and use the same syntax as inline styles: Each CSS property is followed by a colon (:) and the value of the property. Multiple CSS properties and their values are separated by semicolons (;). CSS styles placed in the <head> apply to matching elements wherever they appear in the entire document. A CSS selector determines which XHTML elements will be styles according to the CSS rules. For example, one of the selectors we used in the example is h1, which means that the style declarations that are contained in the braces should be applied to every h1 element in the document.

*CSS embedded style sheets also allow us to declare CSS style classes that begin with periods, such as .favoritestyle, that can be applied to any XHTML element using the elements class attribute. We can apply this style to any element type, whereas the other CSS rules in the style sheet apply only to specific element types defined in the style sheet, such as p and h1. Style-class declarations must be preceded by a period.

*The style element's type attribute specifies the Multipurpose Internet Mail Extension (MIME) type that describes a file's content. CSS documents use the MIME type text/css. The body of the style sheet element declares the CSS rules for the style sheet.

*CSS selectors are important in jQuery because they are the basis by which you select elements in order to perform operations on them.

Linking CSS External Style Sheets for XHTML Elements in the XHTML Document in XHTML Documents Head Section, <link> Element: Rather than define the same set of styles in each of your HTML documents, you can create a separate style sheet. This is a file, with the conventional .css file extension, into which you put your styles.

*External style sheets are a convenient way to create a document with a uniform theme across multiple pages. With external style sheets, which is a separate document with a .css extension that contains only CSS rules, you can provide a uniform look and feel to an entire web-site. We don't need to use a style element in a style sheet. You just define the selectors and declarations directly. We then use the link element to bring the styles into your document, as shown below.

*Different pages on a site can all use the same style sheet. When changes to the style are required, the author needs to modify only a single CSS file to make style changes across the entire website. Note that while embedded style sheets separate content from presentation, both are contained in a single XHTML file. External style sheets separate the content and style into separate files. Always use an external style sheet when developing a website with multiple pages. External style sheets separate content from presentation, allowing for more consistent look-and-feel, more efficient development, and better performance. External style sheets reduce load time and bandwidth usage on a server, since the style sheet can be downloaded once, stored by the web browser, and applied to all pages on a website.

Example: The following is an external CSS style sheet. CSS comments may be placed in any type of CSS code: inline styles, embedded styles sheets and external style sheets, and always start with a / and end with a */. Text between these delimiters is ignored by the browser.

```
/* External style sheet: styles.css */

body  { background-image: url(logo.gif); background-color: #eeeeee; font-family: arial, helvetica }

h1  { font-family: tahoma, helvetica, sans-serif }

p   { font-size: 12pt; font-family: arial, sans-serif }

/* Declare CSS style classes that begin with periods that can be applied to any XHTML elements */

.favoritestyle  { font-size: 14pt; color: #666ff }
```

*An XHTML document references an external style sheet in the XHTML header with a link element that uses the rel attribute with value "stylesheet" to specify a relationship between the document and another document, in this case a CSS stylesheet. The link element's type attribute specifies the Multipurpose Internet Mail Extension (MIME) type that describes a file's content. CSS documents use the MIME type text/css. The link elements href attribute provides the URL for the document containing the style sheet. We can link to as many style sheets as we need, referencing one per link element. The order in which we import style sheets is important if we define two styles with the same selector. The one that is loaded last will be the one that is applied.

*Example: An linked CSS external style sheet inside the XHTML documents <head> section, link element.

```
<html xmlns = "http://www.w3.org/1999/xhtml">

  <head>

    <title>Linking CSS External Style Sheets Example</title>

    <!-- The link element declares a linked CSS external style sheet styles.css in same directory -->

    <link rel = "stylesheet" type = "text/css" href = "styles.css"/>

  </head>
```

```
  <body>

    <!-- The XHTML element class attribute applies the .favoritestyle style -->

    <h1 class = "favoritestyle">Hello World</h1>

    <p> This is an introduction to the study of ...</p>

  </body>

</html>
```

CSS Selectors to Select HTML Elements: Each CSS selector selects elements in the document, and the different kinds of selector tell the browser to look for elements in the document in different ways.
*There are different kinds of CSS selectors in the style sheet: for element names, such as h1 and input; for class names, which start with a period, such as .dtable and .row; and for id names, which start with a pound, such as #butonDiv and #oblock.
*Different kinds of selectors are defined by CSS, starting with the core selectors, listed below. These selectors are the most widely used.

CSS Selector	Description
*	Selects all elements.
<type>	Selects elements of the specified type, such as h1 and input.
.<class>	Selects elements of the specific class, irrespective of element type, such as .dtable and .row, where dtable and row are element class attribute values.
<type>.<class>	Selects elements of the specified type that are members of the specified class.
#<id>	Selects elements with the specified value for the id attribute, such as #butonDiv and #oblock, where butonDiv and oblock are element id attribute values.

Table: The core CSS selectors.
*A selector can also have multiple components, such as: .dcell > *. Such a selector in a style sheet, such as the following in the example below: .dcell > * {vertical-align: middle}. This selector matches all of the elements that are children of elements that belong to the dcell class, and the declaration sets the vertical-align property to the value middle.
*Example: The following shows the contents of a CSS file styles.css, into which we've placed the styles for a HTML document.
//File: The styles.css file:

```
h1 {
   min-width: 700px; border: thick double black; margin-left: auto;
   margin-right: auto; text-align: center; font-size: x-large; padding: .5em;
   color: darkgreen; background-image: url("border.png");
   background-size: contain; margin-top: 0;
}
.dtable {display: table;}
.drow {display: table-row;}
.dcell {display: table-cell; padding: 10px;}
.dcell > * {vertical-align: middle}
input {width: 2em; text-align: right; border: thin solid black; padding: 2px;}
label {width: 5em; padding-left: .5em;display: inline-block;}
#buttonDiv {text-align: center;}
#oblock {display: block; margin-left: auto; margin-right: auto; min-width: 700px;}
```

*We don't need to use a style element in a style sheet. We just define the selectors and declarations directly. We then use the link element to bring the styles into your document, as shown in Listing 3-6.

XXXXXXXStart

XXXXXXXEnd

JavaScript and TypeScript: Client-Side Scripting for XHTML Elements and XHTML Documents

*Ref: TypeScript homepage: http://www.typescriptlang.org/

*The JavaScript scripting language facilitates a disciplined approach to designing computer programs that enhance the functionality and appearance of web pages. JavaScript is the de-facto standard client-side (browser-based) scripting language for web-based applications. All major web browsers contain JavaScript interpreters, which processes the commands written in JavaScript. Both Internet Explorer and Firefox use JavaScript as the default client-side scripting language. The client-side scripting makes web pages more dynamic and interactive. JavaScripting can also be used for server-side scripting.

*JavaScript can be either inline scripting, in which the JavaScript is written in the <body> section of an XHTML document, or can be embedded JavaScript, in which the JavaScript is written in the <head> section <script> element of an XHTML document. The browser interprets the contents of the <head> section first, so the JavaScript program executes before the <body> of the XHTML document displays.

*TypeScript, an Introduction: Angular applications are written in TypeScript, which is a superset of JavaScript. Working with TypeScript provides some useful advantages, but requires that TypeScript files are processed to generate backward-compatible JavaScript that can be used by browsers.

*TypeScript simplifies working with Angular and, with just a few exceptions, you can write Angular applications using the JavaScript skills you already have.

*Angular applications are typically written in TypeScript. TypeScript is a superscript of JavaScript, but one of its main advantages is that it lets you write code using the latest JavaScript language specification with features that are not yet supported in all of the browsers that can run Angular applications.

*The TypeScript compiler generates browser-friendly JavaScript files automatically when a change to a TypeScript file is detected. The TypeScript compiler processes the code to generate JavaScript code that uses only the subset of features that are widely supported by browsers. TypeScript files generally have the .ts extension, and the TypeScript compiler generates the JavaScript code, with the .js extension, that will work in browsers.

*Even with the most recent additions to the JavaScript specification, the TypeScript compiler will convert them to code that can run in older browsers. One of the most important features of TypeScript is that you can just write "normal" JavaScript code as though you were targeting the browser directly.

The Browser JavaScript Console for JavaScript Debugging: The console is a basic, and useful, tool that the browser provides that lets you display debugging information as your script is executed. Each browser has a different way of showing the console. For Google Chrome, we select JavaScript console from the Tools menu. For Internet Explorer, we use the browser's F12 developer tools (so called because they are typically opened by pressing the F12 key) and look at the JavaScript console that shows the effect JavaScript calls. You can also see the HTML content with all the DOM changes made by the DOM API, jQuery, or Angular using the DOM API. You can't do this looking at the page source, which just shows the HTML sent by server and not the changes made by the DOM API, jQuery, or Angular using the DOM API. Other browsers have similar features.

*The following shows a simple example:

```
//A simple inline script:

<!DOCTYPE HTML>

<html>

<head>

  <title>Example</title>

  <script type="text/javascript">

    console.log("Hello");

  </script>

</head>

<body>

  This is a simple example

</body>

</html>
```

*This script writes a message to the console. we can see that the output from calling the console.log method is displayed in the console window, along with the details of where the message originated, in this case on line 6 of the example.html file.

*The following is the output shown on the JavaScript console:

```
Hello
```

JavaScript Comments:
*JavaScript comments are similar to those found in C/C++/Java.
*JavaScript single-line comments start with the delimiters // to indicate that the remainder of the line is a comment, called as a single-line comment.
JavaScript multi-line comments begin with the delimiter / and end with the delimiter */.
Examples of JavaScript comments are:

```
// This is a comment.

/* This is another comment. */
```

JavaScript Statements:

*The basic JavaScript building block is the statement. Each statement represents a single command, and statements are usually terminated by a semicolon (;). The semicolon is optional, but using them makes your code easier to read and allows for multiple statements on a single line.

*The following are JavaScript statements:

```
//JavaScript statements:

console.log("This is a statement");

console.log("This is also a statement");
```

*The browser executes each statement in turn. In this example, all the statements simply write messages

to the console. The results are as follows:

```
This is a statement

This is also a statement
```

JavaScript Executable Statement Termination: Like the programming languages C++/Java, an executable statement in JavaScript must end with a semicolon (;).

JavaScript is case sensitive: JavaScript is case sensitive with keywords, and variable identifiers – that is, it handles uppercase and lowercase letters as different characters.

JavaScript Variable Declaration and Variable Identifiers:
*A variable refers to a location in memory where a value can be stored for later use by the program.
*Every variables has a name, a type and a value.
*Unlike C/C++/Java, JavaScript does not require variables to have a declared type before they can be used in a program. A variable in JavaScript can contain a value of any data type, and in many situations
JavaScript automatically converts between values of different types for you. For this reason, JavaScript is referred to as a loosely typed language.
*All variables should be declared with the keyword var before they can be used. Note: using var to declare variables is not strictly required, but declaring variables with var is recommended and ensures proper behavior of a script. For example, the following statements declares variables name and helloWorld.

```
var name;
var helloWorld = "Hello World!";
```

The value null is a JavaScript keyword signifying that a variable has no value. Note that null is not a string literal, but rather a predefined term indicating the absence of a value. For example:

```
var name = null;
```

Variable declarations end with a semicolon (;) and can be split over several lines with each variable in the declaration separated by a comma, known as a comma-separated list of variable names. Several variables may be declared either in one declaration or in multiple declarations. An example of a comma-separated list of variables:

```
var name, helloWorld = "Hello World!";
```

*The name of a variable can be any valid identifier. JavaScript variable identifiers consists of a series of letters, digits, the underscore (_) character, and the dollar sign ($) and can begin only with a letter or an underscore character or a dollar character and is not a JavaScript keyword; it cannot begin with a digit. Identifiers cannot contain spaces. Some valid identifiers are Welcome, $value, _value, m_inputField1, and button7. The name 7button is not a valid indentifier because it begins with a digit, and the name input field is not valid, because it contains a space.
*Since JavaScript is case sensitive, JavaScript variable identifiers are case sensitive.
*Unlike strongly-typed languages like C/C++/Java in which the programmer must declare the variable type before using the variable in the program and the variable declared can only be used to hold any data of the declared type

and cannot be used to hold data of other types, JavaScript uses dynamic typing, which means that JavaScript determines an object's type during program execution. For example, if object a is initialized to 2, then the object is of type integer. Similarly, if object b is initialized to the string "Hello World!", then the object is of type string. The same JavaScript variable can be used to hold different types of data at different points in the program.

*__JavaScript Variable Scopes__: The scope of a variable is the region of a program in which it is defined. JavaScript variable will have only two scopes.

*(1) Global Variables: have global scopes which means it is defined everywhere in the JavaScript code.

*(2) Local Variables: will be visible only within a function where it is defined. Function parameters are always local to that function. Within the body of a function, a local variable takes precedence over a global variable with the same name:

```
var myVar = "global";     // ==> Declare a global variable
function ( ) {
  var myVar = "local";   // ==> Declare a local variable
  document.write(myVar); // ==> local
}
```

*__Declaring Block Scope Variables Using the let Keyword in ECMAScript 6__: The let keyword is used to declare variables and, optionally, assign a value to the variable in a single statement.

*Variables declared with let are scoped to the block of code in which they are defined, as shown below:

```
//Using let to declare variables:
let messageFunction = function (name, weather)
{
  let message = "Hello, Adam";
  if (weather == "sunny")
  {
    let message = "It is a nice day";
    console.log(message);
  }
  else
  {
    let message = "It is " + weather + " today";
    console.log(message);
  }
  console.log(message);
}
messageFunction("Adam", "raining");
```

*In this example, there are three statements that use the let keyword to define a variable called message.

*The scope of each variable is limited to the block of code that it is defined in.

*When this function is invoked with:

```
messageFunction("Adam", "raining");
```

*It produces the following results:

```
It is raining today
Hello, Adam
```

*As seen above, there is another keyword that can be used to declare variables: var.

*The let keyword is a relatively new addition to the JavaScript specification that is intended to address some oddities in the way var behaves.

*The following takes the above example and replaces let with var.

```
//Using var to declare variables:
let messageFunction = function (name, weather)
{
  var message = "Hello, Adam";
  if (weather == "sunny")
  {
    var message = "It is a nice day";
    console.log(message);
  }
  else
```

```
    {
      var message = "It is " + weather + " today";
      console.log(message);
    }
    console.log(message);
}
```

*When this function is invoked with:

```
messageFunction("Adam", "raining");
```

*It produces the following results:

```
It is raining today
It is raining today
```

*The problem is that the var keyword creates variables whose scope is the containing function, which means that all the references to message are referring to the same variable.

*This can cause unexpected results, and is the reason that the more conventional let keyword was introduced.

*You are free to use let or var in Angular development; we have used let throughout.

*The Differences Between Declaring Function Scope Variables with the var Keyword and Block Scope Variables with the let Keyword in ECMAScript 6: ECMAScript 6 introduced the let statement, to create variables with local block scopes.

*The difference between declaring variables with let and var is variable scoping.

*var is scoped to the nearest function block and let is scoped to the nearest enclosing block, which can be smaller than a function block. Both are global if outside any block.

*Also, variables declared with let are not accessible before they are declared in their enclosing block. This will throw a ReferenceError exception.

*A variable defined using a var statement is known throughout the function it is defined in, from the moment it is defined onward.

*A variable defined using a let statement is only known in the block it is defined in, from the moment it is defined onward.

*Some people would argue that in the future we'll only use let statements and that var statements will become obsolete.

*Declaring Global Variables Using the let and var Keywords: They are very similar when used like this outside a function block, and both provide global scope.

```
let me = 'go';  //Globally scoped

var i = 'able'; //Globally scoped
```

However, global variables defined with let will not be added as properties on the global window object like those defined with var.

```
console.log(window.me); // undefined

console.log(window.i); // 'able'
```

*Declaring Function Scoped Variables Using the let and var Keywords: They are identical when used like this in a function block.

```
function ingWithinEstablishedParameters()

{

  let terOfRecommendation = 'awesome worker!'; //Function block scoped

  var sityCheerleading = 'go!'; //Function block scoped

}
```

*Declaring Blocked Scoped Variables Using the let and var Keywords: Here is the difference. let is only visible in the for() loop and var is visible to the whole function.

```
function allyIlliterate()

{

  //tuce is not visible out here.

  for( let tuce = 0; tuce < 5; tuce++ )

  {

    //tuce is only visible in here (and in the for() parentheses)

    //and there is a separate tuce variable for each iteration of the loop.

  }

  //tuce is *not* visible out here.

}

function byE40()

{

  //nish is visible out here.

  for( var nish = 0; nish < 5; nish++ )

  {

    //nish is visible to the whole function.

  }

  //nish is visible out here.

}
```

*Redeclaring the Same Scoped (Global, Function, Blocked) Variables Using the let and var Keywords: Assuming strict mode, var will let you re-declare the same variable in the same scope. On the other hand, let will not:

```
'use strict';

let me = 'foo';

let me = 'bar'; //SyntaxError: Identifier 'me' has already been declared

'use strict';

var me = 'foo';

var me = 'bar'; //No problem, `me` is replaced.
```

***Comparing JavaScript Variable Values undefined and null**: JavaScript defines a couple of special values that you need to be careful with when you compare them: undefined and null. The undefined value is returned when you read a variable that hasn't had a value assigned to it or try to read an object property that doesn't exist. When a variable is declared in JavaScript, but is not given a value, the variable has an undefined value. Attempting to use the value of such a variable is normally a logic error. JavaScript is unusual in that it also defines null, another special value. The null value is slightly different from the

Undefined alue. The undefined value is returned when no value is defined, and null is used when you want to indicate that you have assigned a value but that value is not a valid object, string, number, or boolean; that is, you have defined a value of no value. When variables are declared, they are not assigned values unless specified by the programmer. Assigning the value null to a variable indicates that it does not contain a value.

*To help clarify this, the following shows the transition from undefined to null.

```
//Using undefined and null values.
<!DOCTYPE HTML>
<html>
<head>
  <title>Example</title>
  <script type="text/javascript">
    var myData =
    {
      name: "Adam",
    };
    console.log("Var: " + myData.weather);
    console.log("Prop: " + ("weather" in myData));
    myData.weather = "sunny";
    console.log("Var: " + myData.weather);
    console.log("Prop: " + ("weather" in myData));
    myData.weather = null;
    console.log("Var: " + myData.weather);
    console.log("Prop: " + ("weather" in myData));
  </script>
</head>
<body>
  This is a simple example
</body>
</html>
```

*We created an object and then try to read the value of the weather property, which is not defined:

```
console.log("Var: " + myData.weather);
console.log("Prop: " + ("weather" in myData));
```

*There is no weather property, so the value returned by calling myData.weather is undefined, and using the in keyword to determine whether the object contains the property returns false. The output from these two statements is as follows:
Var: undefined
Prop: false
*Next, we assign a value to the weather property, which has the effect of adding the property to the object:
myData.weather = "sunny";
console.log("Var: " + myData.weather);
console.log("Prop: " + ("weather" in myData));
*We read the value of the property and check to see whether the property exists in the object again. As we might expect, the object does define the property and its value is sunny:
Var: sunny
Prop: true
*Now we set the value of the property to null, like this:
myData.weather = null;
*This has a specific effect. The property is still defined by the object, but we've indicated it doesn't contain a value. When we perform my checks again, we get the following results:
Var: null
Prop: true
*This distinction is important when it comes to comparing undefined and null values because null is an object and undefined is a type in its own right.
*Checking JavaScript Variable Values for undefined and/or null: If we want to check to see whether a property is null or undefined, and we don't care which, then we can simply use an if statement and the negation operator (!), as shown below. Here, we rely on type coercion that JavaScript performs such that the values we are checking are treated as boolean values. If a variable or property is null or undefined, then the coerced boolean value is false.

```
//Checking to see whether a property value is null or undefined
<!DOCTYPE HTML>
<html>
<head>
  <title>Example</title>
  <script type="text/javascript">
    var myData =
    {
      name: "Adam",
      city: null
    };
    if (!myData.name)
    {
      console.log("name IS null or undefined");
    }
    else
    {
      console.log("name is NOT null or undefined");
    }
    if (!myData.city)
    {
      console.log("city IS null or undefined");
    }
    else
    {
      console.log("city is NOT null or undefined");
    }
  </script>
</head>
<body>
  This is a simple example
```

```
</body>
</html>
```
*Here, we rely on type coercion that JavaScript performs such that the values we are checking are treated as boolean values. If a variable or property is null or undefined, then the coerced boolean value is false.
*The example produces the following output:
name is NOT null or undefined
city IS null or undefined
*To differentiate between the null and undefined values for a variable, then to treat an undefined value as being the same as a null value, then we can use the equality operator (==) and rely on JavaScript to convert the types. An undefined variable will be regarded as being equal to a null variable, for example. If we want to differentiate between null and undefined values, then we need to use the identity operator (===). The following shows both comparisons.
```
//Equality and identity comparisons for null and undefined values:
<!DOCTYPE HTML>
<html>
<head>
  <title>Example</title>
  <script type="text/javascript">
    var firstVal = null;
    var secondVal;
    var equality = firstVal == secondVal;
    var identity = firstVal === secondVal;
    console.log("Equality: " + equality);
    console.log("Identity: " + identity);
  </script>
</head>
<body>
  This is a simple example
</body>
</html>
```
*The output from this script is as follows:
Equality: true
Identity: false

Adding JavaScript to an XHTML Document Head Section Using the <script> Element: JavaScript code is added to an HTML document as a script: a block of JavaScript statements that the browser will execute. There are different ways we can add scripts. We can define an inline script, where the content of the script is part of the HTML document. We can also define an external script, where the JavaScript is contained in a separate file and referenced via a URL, which is also how we access the jQuery library. Both of these approaches rely on the script element.

*JavaScript code is added to an HTML document using the script element. The src attribute is used to specify which JavaScript file should be loaded.

*The HTML <script> Tag: The <script> tag is used to define a client-side script (JavaScript). The <script> element either contains scripting statements, or it points to an external script file through the src attribute. Common uses for JavaScript are image manipulation, form validation, and dynamic changes of content. Note: If the "src" attribute is present, the <script> element must be empty.

*Embeeded JavaScript for XHTML Elements: JavaScript Global Script Level Variables in the XHTML Document Head Section, <script> Element: Embedded JavaScript is written in the <head> section <script> element of an XHTML document. The browser interprets the contents of the <head> section first, so the JavaScript program executes before the <body> of the XHTML document displays.

*Global variables or script-level variables are declared in the <head> section, <script> element, and are accessible in any part of the script, and are said to have global scope. Thus, every function in the script can potentially use these variables.

*The <script> element inside the XHTML <head> section is used to indicate to the browser that the text which follows is part of a script. The <script> element's type attribute specifies the Multipurpose Internet Mail Extension (MIME) type that describes a file's content. JavaScript documents use the MIME type text/javascript.

*Some older web-browsers do not support scripting. In such browsers, the actual text of a script often will display in the web page. To prevent this from happening, many script programmers enclose the script code in an XHTML comment, so that browsers that do not support scripts will simply ignore the script. Browsers that support scripting will interpret the JavaScript as expected. The syntax used is as follows:

```
<script type = "text/javascript">

 <!--

 JavaScript codes here.

 // -->

</script>
```

*Adding JavaScript to an XHTML Document Body Section Using the <script> Element: We can also add <script> elements at the end of the body element. When browsers finds a script element in a document, it executes the JavaScript statements right away, before the rest of the document has been loaded.

*This presents a problem when you are working with the DOM because, for example, it means searches for elements via the Document object are performed before the objects we are interested in have been created in the model. To avoid this, we can place the script element at the end of the document.

*jQuery also provides a nice way of dealing with this issue, using the jQuery ready() function, which Waits for the Document Object Model (DOM) to be created, see the jQuery section.

JavaScript Assignment Operators:
*The JavaScript assignment operator is =. It is a binary operator because it takes two operands.
*JavaScript augmented assignment operators: JavaScript provides several augmented assignment operators for abbreviating assignment expressions. Any statement of the form
variable = variable operator expression
where operator is a binary operator, such as +, -, *, /, **, or %, can be written in the more compact form:
variable operator= expression

Augmented assignment operator	Description
+=	c += 7, equivalent to c = c + 7.
-=	c -= 7, equivalent to c = c – 7.
*=	c *= 7, equivalent to c = c * 7.
/=	c /= 7, equivalent to c = c / 7.
%	c %= 7, equivalent to c = c % 7.

JavaScript Arithmetic Operators (Binary Operators):

*JavaScript arithmetic operators are binary operators in that they take two operands.
*JavaScript arithmetic binary operators:

Operator	Description
+	Addition. f + 7
-	Subtraction. f – 7
*	Multiplication. f * 7
/	Division. f / 7
%	Modulus. f % 7

JavaScript Pre- and Post- Increment and Decrement Arithmetic Operators (Unary Operators):
*JavaScript also provides the arithmetic pre-increment and post-increment unary operator ++ and the pre-decrement and post-decrement unary operator --, all of which are unary operators in that they take only one operand.
*If a scalar variable c needs to be incremented by 1, the pre-increment or post-increment unary operator ++ can be used to do so, rather than the expression c = c + 1 or c += 1. Similarly, if the scalar variable c needs to be decremented by 1, the pre-decrement or post-decrement unary operator -- can be used to do so, rather than the expression c = c - 1 or c -= 1. If an increment or decrement operator is placed before a variable name, it is referred to as pre-increment or pre-decrement operator, and it causes the variable to be incremented or decremented by 1, and the new value of the variable is used in the expression in which the variable appears. If an increment or decrement operator is placed after a variable name, it is referred to as post-increment or post-decrement, and it causes the current value of the variable to be used in the expression in which it appears, and then the value of the variable is incremented or decremented by 1.
*JavaScript arithmetic pre- and post- increment and decrement unary operators:

Operator	Description
++	Pre-increment. Example: ++c. Increment c by 1, then use the new value of c in the expression in which c resides.
++	Post-increment. Example: c++. Use the current value of c in the expression in which c resides, then increment c by 1.
--	Pre-decrement. Example: --c. Decrement c by 1, then use the new value of c in the expression in which c resides.
--	Post-decrement. Example: c--. Use the current value of c in the expression in which c resides, then decrement c by 1.

JavaScript Equality and Relational Operators for JavaScript Primitive and Object Types:

*Conditions in if structures can be formed by using the equality operators , the identity operator, and the relational operators. JavaScript primitives are compared by value, but JavaScript objects are compared by reference.

*String comparisons in JavaScript are case sensitive, so “august” does not equal “August”. Also, whitespace is significant in string comparisons, so neither of the previous two strings equals “Au gust”. String comparisons in JavaScript compare strings alphabetically using the ASCII value of each character, comparing one character at a time, beginning at the first character of each string and proceeding through both strings until the first inequality is found. Thus, since letters later in the alphabet have greater ASCII value, “rabbit” is greater than “dragon” and “trombone” is less than “trumpet”. The ASCII value of the letter r (114) is greater than that of the letter d (100), so any word beginning with r is considered greater than any word beginning with a d. Also, a word beginning with A

(65) is less than a word beginning with a (97). This string comparison can lead to strange results, such as string "100" is less than string "2", since the ASCII value of 1 (49) is less than that of 2 (50). Finally, in the special case where one string is the prefix of another string, such as "dog" is the prefix to "doghouse", the string relational operators regard the longer string as greater than the other, and so "doghouse" gt "dog" returns true.

Relational Operator	Description
== (Equality operator)	*Equality operator: Equality operators can be used for comparing numeric values, strings, etc. *Note: Using the assignment operator = for equality in a conditional statement is a syntax error. Using the equality operator == for assignment is a logical error. *Usage: if(result == 1) { statement(s) } *Usage for strings: var result = "August"; if(result == 1) { statement(s) }

===
 	
(Identity operator)
	*The identity operator (===): Both value and type are equal.
	
	*Note: The equality operator vs. the identity operator: The equality and identity operators are of particular note.
	
	*The equality operator will attempt to coerce (convert) operands to the same type in order to assess equality. This is a handy feature, as long as you are aware it is happening.
	
	*The following example shows the equality operator in action:
	
	//Using the equality operator:
	
	let firstVal = 5;
	
	let secondVal = "5";
	
	if (firstVal == secondVal)
	
	{
	
	console.log("They are the same");
	
	}
	
	else
	
	{
	
	console.log("They are NOT the same");

	}
	
	*The output from this script is as follows:
	
	They are the same
	
	*JavaScript is converting the two operands into the same type and comparing them. In essence, the equality operator tests that values are the same irrespective of their type.
	
	*If you want to test to ensure that the values and the types are the same, then you need to use the identity operator (===, three equal signs, rather than the two of the equality operator), as below:
	
	//Using the identity operator:
	
	let firstVal = 5;
	
	let secondVal = "5";
	
	if (firstVal === secondVal)
	
	{
	
	console.log("They are the same");
	
	}
	
	else
	
	{
	
	console.log("They are NOT the same");
	
	}
	
	*In this example, the identity operator will consider the two variables to be different. This operator doesn't coerce types.
	
	*The result from this script is as follows:
	
	They are NOT the same

!=	Not equal operator: Not equal operators can be used for comparing for comparing numeric values, strings, etc. *Usage: if(result != 1) { statement(s) } *Usage for strings: var result = "August";

	if(result != “Au gust”) { statement(s) }
!==	Not identity operator (!==): Both value and type are not equal.
<	Less than operator. * sage: if(result < 1) { statement(s) }
>	Greater than operator. *Usage: if(result > 1) { statement(s) }
<=	Less than or equal to operator. *Usage: if(result <= 1) { statement(s) }
>=	Greater than or equal to operator. *Usage: if(result >= 1) { statement(s) }

JavaScript Logical Operators for JavaScript Primitive and Object Types: JavaScript provides logical operators that are used to form more complex conditions by combining multiple simple conditions. The JavaScript logical operators are && (logical AND), || (logical OR), and ! (logical NOT, also called logical negation). JavaScript evaluates to true or false all expressions that include relational operators, equality operators, and/or logical operators.

Logical operator	Description

&&	Logical and.
	
	*Usage:
	
	if(gender == ‘Female’ && age >= 65)
	
	{
	
	++seniorFemale;
	
	}
	
	Note that because the precedence of == and >= are higher than the precedence of &&, no parenthesis are necessary. This can be made more readable by adding redundant parenthesis:
	
	if((gender == ‘Female’) && ($age >= 65))

	{
	
	++seniorFemale;
	
	}
	
	Shortcircuit evaluation: If the condition on the left is false, then the entire logical && expression evaluates to false, and the right-side condition is not evaluated. However, if the condition on the left is true, the simple condition to the right of the && operator is also evaluated next.

||	Logical or.
	
	*Usage:
	
	if(semesterAvg >= 90 || finalExam >=90)
	
	{
	
	document.writeln("Grade is A");
	
	}
	
	Note that because the precedence of >= is higher than the precedence of ||, no parenthesis are necessary. This can be made more readable by adding redundant parenthesis:
	
	if((semesterAvg >= 90) || (finalExam >=90))
	
	{
	
	document.writeln("Grade is A");
	
	}
	
	Shortcircuit evaluation: If the condition on the left is true, then the entire logical || expression evaluates to true, and the right-side condition is not evaluated. However, if the condition on the left is false, the simple condition to the right of the || operator is also evaluated next.

!	Logical not, also called logical negation. It reverses the truth or falsity value of a condition. Unlike the "and" and the "or" operators that combine two conditions operands (binary operators), not has a single condition as the operand (or is a unary operator). The not operator is placed before a condition.
	
	*Usage:
	
	if(!(grade >= 90))
	
	{
	
	document.writeln("Grade is not A");
	
	}

	The parenthesis around the condition (grade >= 90) are needed because the logical negation operator has a higher precedence than the >= operator. Normally, the programmer can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator. For example, the above is equivalent to the simpler form:
	
	if(grade < 90)
	
	{
	
	document.writeln("Grade is not A");
	
	}

Table: JavaScript logical operators.

JavaScript Selection Control Structures: JavaScript provides three types of selection structures: if, if-else, and if-elseif-else. The if selection structure either performs (selects) an action if a predicate condition is true or skips the action if the condition is false. The conditions in selection statements can be formed by using the equality operators (==, !=), identity operator (===), and relational operators (>, <, >=, <=) and logical operators. JavaScript primitives are compared by value, but JavaScript objects are compared by reference.

Control structure	Availability

if	if single selection structure.
	
	The if selection structure either performs (selects) an action if a predicate condition is true or skips the action if the condition is false. Note that the parenthesis around the test condition are necessary:
	
	if(grade >= 60)
	
	{ statements(s) }
	
	Curly braces ({}) must enclose the body of the selection control structure. The exception to this rule for if selection that executes only one statement is shown below: If only one statement is to be executed when a condition is met, such as in a print statement, JavaScript allows the code to be written without curly braces, as in:
	
	if(sales >= 50)
	
	document.write(" Earned bonus! ");

if-else	if-else double selection structure.
	
	if(grade >= 60)
	
	{ statements(s) }
	
	else
	
	{ statements(s) }
	
	Curly braces ({}) must enclose the body of the selection control structure. The exception to this

	rule for if selection that executes only one statement. If only one statement is to be executed when a condition is met, JavaScript allows the code to be written without curly braces.

if-else if-else	if-else if-else multiple selection structure.
	
	Note the else statement of the if-else if-else structure is optional.
	
	if(grade >= 90)
	
	{ statements(s) }
	
	elsif(grade >= 80)
	
	{ statements(s) }
	
	elsif(grade >= 70)
	
	{ statements(s) }
	
	elsif(grade >= 60)
	
	{ statements(s) }
	
	else
	
	{ statements(s) }
	
	Curly braces ({}) must enclose the body of the selection control structure. The exception to this rule for if selection that executes only one statement. If only one statement is to be executed when a condition is met, JavaScript allows the code to be written without curly braces.

?: conditional operator (ternary operator)	?: (conditional operator, or ternary operator) double selection structure. JavaScript provides an operator, called the ternary operator or the conditional operator (?:), that is closely related to the if/else structure. The operator ?: is JavaScript's only ternary operator: it takes three operands. The operands together with the ?: form a conditional expression. The first operand is a boolean expression, the second is the value for the conditional expression if the condition evaluates to true and the third is the value for the conditional expression if the condition evaluates to false. For example, the statement document.writeln(studentGrade >= 60 ? "Passed" : "Failed"); contains a conditional expression that evaluates to the string "Passed" if the condition studentGrade >= 60 is true and evaluates to the string "Failed" if the condition is false. The precedence of the conditional operator is low, so the entire conditional expression is normally placed in parentheses to ensure that it evaluates correctly
switch	switch multiple selection structure: If a series of decisions in which a variable or expression is tested separately for each of the finite set of values it may assume, and different actions are taken for each value. JavaScript provides the switch multiple-selection statement to handle such decision making. The switch statement consists switch statement with a controlling expression in parenthesis, followed by a series of case labels and an optional default case. When the flow

	of control reaches the switch statement, the script evaluates the controlling expression in parenthesis following keyword switch. The value of this expression is compared with the value in each of the case labels, starting with the first case label. Each of the break statements in the case blocks causes program control to proceed with the first statement after the switch statement. The break statement is used because the cases in a switch statement would otherwise run together. If break is not used anywhere in a switch statement, then each time a match occurs in the statement, the statements for all the remaining cases execute. Thus, having several case labels listed together, such as case 1: case 2: with no statements between the cases, simply means that the same set of actions is to occur for each case.
	*Usage:
	//Ask a user to select a list item style:
	var choice = window.prompt(“Select a list items style: \n” +
	“1 (numbered), 2 (lettered), 3 (roman)”, “1”);
	var validInput = true; //Indicates if user input is valid.
	var listType; //Type of list item as a string.
	var startTag, endTag; //Start list item tag and end list item tag.
	switch(choice)
	{
	case “1”: //Numbered List Items
	startTag = “<o1>”;
	endTag = “</o1>”;
	listType = “ Numbered List Items ”;
	break;
	case “2”: //Lettered List Items
	startTag = “<o1 style = \”list-style-type: upper-alpha\”>”;
	endTag = “</o1>”;
	listType = “ Lettered List Items ”;
	break;
	case “3”: //Roman Numbered List Items
	startTag = “<o1 style = \”list-style-type: upper-roman\”>”;
	endTag = “</o1>”;
	listType = “ Roman Numbered List Items ”;
	break;
	default: //If user input is not valid.
	validInput = false; //Indicates user input is not valid.
	break;
	} //switch(choice)
	//Display list items in the user selected list item style:
	if(validInput == true)
	{
	document.writeln(listType + startTag);
	for(var i = 1; i <=3; ++i)
	{
	document.writeln(“ List item “ + i + “ ”);
	} //for(var i = 1; i <=3; ++i)
	document.writeln(listType + startTag);
	} //if(validInput == true)
	else //if(validInput == false)
	{
	document.writeln(“Invalid user selection for list items:“ + choice);
	} // else if(validInput == false)

Table: JavaScript selection control structures.

*Example: Using conditional statements: Many of the JavaScript operators are used in conjunction with conditional statements. The following example uses both the if/else and switch statements:

```
//Using the if/else and switch conditional statements:
let name = "Adam";
```

```
if (name == "Adam")
{
  console.log("Name is Adam");
}
else if (name == "Jacqui")
{
  console.log("Name is Jacqui");
}
else
{
  console.log("Name is neither Adam or Jacqui");
}
//Do a switch selection on name:
switch (name)
{
  case "Adam":
    console.log("Name is Adam");
    break;
  case "Jacqui":
    console.log("Name is Jacqui");
    break;
  default:
    console.log("Name is neither Adam or Jacqui");
    break;
}
```

*The results from the listing are as follows:

Name is Adam

Name is Adam

JavaScript Iteration (Repetition) Control Structures: JavaScript provides four types of iteration structures: while, do-while, for, and for .. in for arrays.

Control structure	Availability

while	while iteration structure.
	
	The loop continuation test is tested at the beginning of the loop, before the body of the loop is executed.
	
	*Usage:
	
	while(product <= 1000)
	
	{ statements(s) }
	
	The curly braces ({}) must enclose the body of the loop. The exception to this rule for while iterations that execute only one statement as shown below:
	
	while(bid_price <= highest_bid)
	
	bid_price += increment

do/while	do/while iteration structure.
	
	The loop continuation test is tested at the end of the loop, after the body of the loop is executed. Thus, the body of the loop is always executed once.
	
	*Usage:
	
	do
	
	{
	
	statements(s)
	
	} while(product <= 1000);
	
	The curly braces ({}) must enclose the body of the loop. The exception to this rule for do/while iterations that execute only one statement as shown below. A semi-colon (;) is needed after the while test condition.
	
	do
	
	bid_price += increment
	
	while(bid_price <= highest_bid);

for	for iteration structure.
	
	The for iteration structure handles all the details of counter-controlled iterations. It specifies each of the items needed for counter-controlled repetition with a control variable.
	
	*Usage:
	
	The general format of the for structure is similar to C/C++/Java:
	
	for(initialization; loopContinuationTest; increment)
	
	{
	
	statements(s)
	
	}
	
	Sometimes, the initialization and increment expressions are comma-separated lists of expressions. The three expressions in the for structure are optional. If the loopContinuationTest is omitted, JavaScript assumes that the loop-continuation condition is true, thus creating an infinite loop. One might omit the initialization expression if the control variable is initialized elsewhere in the program. One might omit the increment expression if the increment is calculated by statements in the body of the for structure or if no increment is needed. The increment expression in the for structure acts like a stand-alone statement at the end of the body of the for structure, since incrementing occurs after the body loop is executed. Therefore, pre-incrementing, post-incrementing, and all of the following expressions are all equivalent in the incrementing portion of the for structure: ++i or i++ or i += 1 or i = i + 1. The increment of a for structure can be negative as in --i, in which case it is really a decrement,

	and the loop actually counts downwards.
	
	Example:
	
	//Using a for loop, display incremental font sizes via CSS styles.
	
	for(var counter = 1; counter <= 7; ++counter)
	
	{
	
	document.writeln("<p style = \"font-size: " + counter + "pt\">" +
	
	"HTML font size " + counter + "pt</p>");
	
	}
	
	The curly braces ({}) must enclose the body of the loop. The exception to this rule for for iterations that execute only one statement as shown below:
	
	for(var bid_price = 100; bid_price <= highest_bid;)
	
	bid_price += increment;

for .. in	For … in iteration structure for XXXXiterals or arrays. XXXComplete. P.368.
	
	The foreach repetition structure allows a programmer to iterate over a list of values, accessing and manipulating each element. A benefit of this control structure is that the programmer is not required to maintain a separate counter-controlled variable, test condition, and increment to manage the repetition structure. As a result, the code is cleaner and more readable. The general format for the foreach structure in Perl is:
	
	foreach controlVariable (list)
	
	{
	
	statements(s)
	
	}
	
	where list can be any list, literal or array and controlVariable represents one item of list during each iteration. Changing the value of the controlVariable in the body of a foreach structure modifies the element in the list that the control variable is currently representing. If the body of the foreach structure does not access or modify the list, it is optional to provide the controlVariable. When no controlVariable is explicitly provided, the special Perl variable $_ will store the value of the current element of the list during each iteration of the loop.
	
	Examples:
	
	#Standard foreach structure. Prints the letters A-G.
	
	foreach $letter('A' .. 'G') #Or, foreach $_('A' .. 'G')

	{
	print “$letter”; #Or, print “$_”;
	}
	#Modify an array by squaring every element of the array.
	@array = (1 .. 10);
	foreach $number(@array) #Or, foreach $_(@array)
	{
	$number **= 2; #Or, $_ **= 2;
	}
	#This structure loops 10 times, not requiring a control variable.
	foreach(1 .. 10) #Or, foreach $_(1 .. 10)
	{
	print “*”;
	}
	As with every Perl control structure, curly braces ({}) must enclose the body of the loop.

break, and continue	Used for loop controls inside the body of a loop structure. The break and continue statements alter the flow of control in an iterative structure, such as a while or a for loop.
	*Usage:
	var counter = 1;
	while(true)
	{
	//Continue loop only if counter < 5.
	if(counter < 5)
	{
	continue;
	}

	//Exit loop only if counter == 5.
	
	if(counter == 5)
	
	{
	
	break;
	
	}
	
	++counter;
	
	}

labelled break, and continue	Used for loop controls inside the body of a loop structure to branch to an outer labelled loop structure. This is useful as loops can be nested infinitely deep. Inside of nested loops, break and continue apply to the innermost loop in which they appear. break and continue can also apply to a block other than the inner most one with the used of labelled break and continue statements. To do this the target block must have a name.
	
	*Usage:
	
	Any loop block can have a block label. A block label is can be any valid identifier followed by a colon (:). Block labels are traditionally written in uppercase letters, as this practice increases readability and avoids conflict between labels and JavaScript keywords. This label acts as a one-word description of the block of code, and can act as a target for the loop control commands break and continue. The target block label is then named as an argument to break or continue to refer to a specific block. When these commands execute without any arguments, they operate on the loop in which they appear.
	
	*labelled break and continue with block labels: The general format for a labelled next structure in JavaScript is shown below. When next OUTERBLOCK; is executed, control jumps out of the inner repetition block and immediately executes the next iteration of the OUTERBLOCK block repetition structure, ignoring the remaining iterations of the inner repetition structure. This gives us the ability to control loop iterations of repetition blocks in which we are not currently operating, from any inner repetition block.
	
	OUTERBLOCK: repetition block()
	
	{
	
	repetition block()
	
	{
	
	next OUTERBLOCK;
	
	}
	
	}
	
	Example:

	OUTERBLOCK: for(var row = 1; row <= 10; row++)
	{
	for(var column = 1; column <= 10; column ++)
	{
	document.writeln(“column = “ + column + “\””);
	
	next OUTERBLOCK if(row < column);
	#Which is equivalent to the following:
	#if(row < column)
	#{
	# next OUTERBLOCK;
	#}
	}
	}

goto with labels	Not available in JavaScript.

Table: JavaScript iteration (repetition) control structures.

JavaScript Functions:

*When the browser receives JavaScript code, either directly through a script element or indirectly through the module loader, it executes the statements it contains in the order in which they have been defined.

*You can also package statements into a function, which won’t be executed until the browser encounters a statement that invokes the function, as shown below:

```
//Defining a JavaScript function:
let myFunc = function ()
{
  console.log("This is a statement");
};
```

*Defining a function simple: use the let keyword followed by the name you want to give the function,

followed by the equal sign (=) and the function keyword, followed by parentheses, (and) characters.

*The statements you want the function to contain are enclosed between braces, the { and } characters.

*In the example, the function contains a single statement that writes a message to the JavaScript console.

*The statement in the function won't be executed until the browser reaches another statement that calls

the myFunc function, like this:

myFunc();

*Executing the statement in the function produces the following output:

This is a statement

*Functions are useful when they are invoked in response to some kind of change or event, such as user interaction.

*Two Ways to Define JavaScript Functions: Named Functions or Anonymous Functions: There are two ways in which you can define functions in JavaScript. A function in JavaScript can be either named or anonymous. A named function can be defined using function keyword as follows:

```
function named(){
  // do some stuff here
}
```

An anonymous function can be defined in similar way as a normal function but it would not have any name. An anonymous function can be assigned to a variable or passed to a method as shown below:

```
var handler = function (){
  // do some stuff here
}
```

*The approach used above is known as a function expression. The same function can also be defined like this:

```
function myFunc()
{
  console.log("This is a statement");
}
```

*This is known as a function declaration. The result is the same: a function called myFunc that writes

a message to the console.

*The difference is how the functions are processed by the browser when a JavaScript file is loaded.

*Function declarations are processed before the code in a JavaScript file is executed, which means you can use a statement that calls a function before it is defined, like this:

```
myFunc();

function myFunc()

{

  console.log("This is a statement");

}
```

*This works because the browser finds the function declaration when it parses the JavaScript file and sets up the function before the remaining statements are executed.

*Function expressions, however, are not processed before the code in a JavaScript file is executed, which means you cannot use a statement that calls a function before it is defined, like this, and this code will not work:

```
myFunc();

let myFunc = function()

{

  console.log("This is a statement");

};
```

*This code will generate an error reporting that myFunc is not a function. Developers who are new to JavaScript tend to prefer using function declarations because the syntax is more consistent with languages like C# or Java.

*The technique you use is entirely up to you, although you should aim to be consistent throughout your project to make your code easier to understand.

*Defining JavaScript Functions with Parameters: JavaScript allows you to define parameters for functions, as shown below:

```
//Defining functions with parameters:

let myFunc = function(name, weather)

{

  console.log("Hello " + name + ".");

  console.log("It is " + weather + " today");
```

```
};
```

*We added two parameters to the myFunc function, called name and weather. JavaScript is a dynamically typed language, which means you don't have to declare the data type of the parameters when you define the function.

*To invoke a function with parameters, you provide values as arguments when you invoke the function, like this:

```
myFunc("Adam", "sunny");
```

*The results from this listing are as follows:

```
Hello Adam.

It is sunny today
```

*Defining JavaScript Functions with Default Parameters: The number of arguments you provide when you invoke a function doesn't need to match the number of parameters in the function.

*If you call the function with fewer arguments than it has parameters, then the value of any parameters you have not supplied values for is undefined, which is a special JavaScript value.

*If you call the function with more arguments than there are parameters, then the additional arguments are ignored.

*The consequence of this is that you can't create two functions with the same name and different parameters and expect JavaScript to differentiate between them based on the arguments you provide when invoking the function. This is called function overloading, and although it is supported in languages such as Java and C#, it isn't available in JavaScript. Instead, if you define two functions with the same name, then the second definition replaces the first.

*There are two ways that you can modify a function to respond to a mismatch between the number of parameters it defines and the number of arguments used to invoke it. Default parameters deal with the situation where there are fewer arguments than parameters and allow you to provide a default value for the parameters for which there are no arguments, as shown below:

```
//Using a default parameter:

let myFunc = function (name, weather = "raining")

{

  console.log("Hello " + name + ".");

  console.log("It is " + weather + " today");

};
```

*In the following function call, the weather parameter in the function has been assigned a default value of raining, which will be used if the function is invoked with only one argument:

```
myFunc("Adam");
```

*Producing the following results:

Hello Adam.

It is raining today

*Defining JavaScript Functions with Rest Parameters: Rest parameters are used to capture any additional arguments when a function is invoked with additional arguments, as shown below:

```
//Using a rest parameter:

let myFunc = function (name, weather, ...extraArgs)

{

  console.log("Hello " + name + ".");

  console.log("It is " + weather + " today");

  for (let i = 0; i < extraArgs.length; i++)

  {

    console.log("Extra Arg: " + extraArgs[i]);

  }

};
```

*The rest parameter must be the last parameter defined by the function, and its name is prefixed with an ellipsis (three periods, ...). The rest parameter is an array to which any extra arguments will be assigned.

*When the above function is invoked as follows:

```
myFunc("Adam", "sunny", "one", "two", "three");
```

*The function prints out each extra argument to the console, producing the following results:

Hello Adam.

It is sunny today

Extra Arg: one

Extra Arg: two

Extra Arg: three

*Defining JavaScript Functions That Return Results: You can return results from functions using the return keyword. We don't have to declare that the function will return a result or denote the data type of the result.

*The following shows a function that returns a result:

```
//Returning a result from a function:

let myFunc = function(name)

{

  return ("Hello " + name + ".");

};
```

*This function defines one parameter and uses it to produce a result.

*When we invoke the function and pass the result as the argument to the console.log function, like this:

```
console.log(myFunc("Adam"));
```

*The result from this listing is as follows:

```
Hello Adam.
```

*Defining JavaScript Functions That Takes a Variable Number of Arguments: JavaScript variable arguments is a kind of array which has length property. Following example explains it very well:

```
function func(x){

  console.log(typeof x, arguments.length);

}

func();               //==> "undefined", 0

func(1);              //==> "number", 1

func("1", "2", "3");  //==> "string", 3
```

*The arguments object also has a callee property, which refers to the function you're inside of. For example:

```
function func() {

  return arguments.callee;

}

func();               // ==> func
```

*Defining JavaScript Functions That Takes Other Functions as Parameters: JavaScript functions can be passed around as objects, which means you can use one function as the argument to another, as shown below:

```
//Using a function as an argument to another function:

let myFunc = function (nameFunction)
```

```
{

  return ("Hello " + nameFunction() + ".");

};

console.log(myFunc(function () { return "Adam"; }));
```

*The myFunc function defines a parameter called nameFunction that it invokes to get the value to insert into the string that it returns. We pass a function that returns Adam as the argument to myFunc, which produces the output:

Hello Adam.

*Functions can be chained together, building up more complex functionality from small and easily tested pieces of code, as shown below:

```
//Chaining functions calls:

let myFunc = function (nameFunction)

{

  return ("Hello " + nameFunction() + ".");

};

let printName = function (nameFunction, printFunction) { printFunction(myFunc(nameFunction)); }

printName(function () { return "Adam" }, console.log);
```

*This example produces the same result as above.

*Defining JavaScript Lamda (Arrow) Functions That Takes Other Functions as Parameters: Lambda functions, also called as lambda expressions, or arrow functions or as fat arrow functions, are an alternative way of defining functions and are often used to define functions that are used only as arguments to other functions.

*The following replaces the functions from the previous example with arrow functions:

```
//Using arrow functions:

let myFunc = (nameFunction) => ("Hello " + nameFunction() + ".");

let printName = (nameFunction, printFunction) => printFunction(myFunc(nameFunction));

printName(function () { return "Adam" }, console.log);
```

*These functions perform the same work as the ones above.

*There are three parts to an arrow function: the input parameters, then an equal sign and a greater-than sign (the “arrow”), and finally the function result. The return keyword and curly braces are required only if the arrow function needs to execute more than one statement.

*Defining Nested JavaScript Functions Inside of Other JavaScript Functions: If you define a function inside another function, creating inner and outer functions, then the inner function is able to access the outer functions variables, using a feature called closure, like this:

```
let myGlobalVar = "apples";

let myFunc = function(name)

{

  let myLocalVar = "sunny";

  let innerFunction = function ()

  {

    return ("Hello " + name + ". Today is " + myLocalVar + ".");

  }

  return innerFunction();

};
```

*The inner function in this example is able to access the local variables of the outer function, including its parameter. This is a powerful feature that means you don't have to define parameters on inner functions to pass around data values, but caution is required because it is easy to get unexpected results when using common variable names like counter or index, where you may not realize that you are reusing a variable name from the outer function.

Handling JavaScript Errors: JavaScript Exception Handling with try, catch, and finally Clauses:
*JavaScript uses the try...catch statement to deal with errors.
*The following shows how to use the try…catch statement.
//Handling an exception:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Example</title>
  <script type="text/javascript">
    try
    {
      var myArray;
      for (var i = 0; i < myArray.length; i++)
      {
        console.log("Index " + i + ": " + myArray[i]);
      }
    }
    catch (e)
    {
      console.log("Error: " + e);
    }
  </script>
</head>
<body>
```

```
  This is a simple example
</body>
</html>
```

*The problem in this script is a common one. We're trying to use a variable that has not been initialized properly. We've wrapped the code that we suspect will cause an error in the try clause of the statement.
*If no problems arise, then the statements execute normally, and the catch clause is ignored.
*However, since there is an error in this code, then execution of the statements in the try clause stops immediately, and control passes to the catch clause, producing the following output on the console:
Error: TypeError: Cannot read property 'length' of undefined
*The error that we've encountered is described by an Error object, which is passed to the catch clause.
*The following table shows the properties defined by the Error object.

Property	Description	Returns
message	A description of the error condition.	string
Name	The name of the error. This is Error, by default.	string
number	The error number, if any, for this kind of error.	number

Table: The Error object.

*The catch clause is our opportunity to recover from or clear up after the error. If there are statements that need to be executed whether or not there has been an error, we can place them in the optional finally clause, as below:

```
//Using a finally clause:
<!DOCTYPE HTML>
<html>
<head>
  <title>Example</title>
  <script type="text/javascript">
    try
    {
      var myArray;
      for (var i = 0; i < myArray.length; i++)
      {
        console.log("Index " + i + ": " + myArray[i]);
      }
    }
    catch (e)
    {
      console.log("Error: " + e);
    }
    finally
    {
      console.log("Statements here are always executed");
    }
  </script>
</head>
<body>
  This is a simple example
</body>
</html>
```

*This example produces the following console output:
Error: TypeError: Cannot read property 'length' of undefined
Statements here are always executed

JavaScript Primitive Objects: JavaScript defines a basic set of primitive types: string, number, and boolean. This may seem like a short list, but JavaScript manages to fit a lot of flexibility into these three types.

*JavaScript comes with a useful set of built-in functions that can be used to manipulate Strings, Numbers and Dates.

*The JavaScript primitive objects are: string, Number, Boolean, Date, and are discussed below:

JavaScript Primitive Object: string Object:

*A string in JavaScript is an immutable object that contains none, one or many characters. Following are the valid examples of a JavaScript String:

"This is JavaScript String"

'This is JavaScript String'

'This is "really" a JavaScript String'

"This is 'really' a JavaScript String"

*A string is a series of unicode characters treated as a single unit. A string may include letters, digits, and various special characters, such as +, -, *, and $. A string is an object of type string.

*String literals or string constants are written as a sequence of characters in double quotation marks or single quotation marks. You define string values using either the double quote or single quote characters, as shown below. The quote characters you use must match. You can't start a string with a single quote and finish with a double quote, for example.

//Defining string variables:

let firstString = "This is a string";

let secondString = 'And so is this';

*If we want to use the double quote character inside a string created with double quotes, we must use the escape sequence for the double quote character (\"). Similarly, if we want to use the single quote character inside a string created with single quotes, we must use the escape sequence for the single quote character (\'). However, if we want to use double-quote character inside a string created with single quotes, we do not need to use the escape sequence (\"). Similarly, if we want to use a single quote character inside a string created with double quotes, we do not need to use the escape sequence (\'). Example of JavaScript string literals are:

"John Doe";

'212-555-1212';

*A string can be assigned to a variable in a declaration, such as:

var color = "blue";

*Individual white-space characters between words in a string are not ignored by the browser. However, if consecutive spaces appear in a string, browsers condense them to a single space. Also, in most cases, browsers ignore leading white-space characters at the beginning of a string.
*JavaScript Escape Sequences for String Outputs: are similar to C/C++/Java.
*Note that each of the last three are used to output a specific character (\, ", and '), as these characters have specific uses in JavaScript.

Escape Sequence	Description
\n	Newline. Move the screen cursor to the beginning of the next line.
\t	Horizontal tab. Move the screen cursor to the next tab stop.
\r	Carriage return. Move the cursor to the beginning of the current line; do not advance to the next line.
\\	Backslash. Used to insert a backslash character in a string.
\”	Double quote. Used to insert a double quote character in a double-quoted string.
\’	Single quote. Used to insert a single quote character in a single-quoted string.

*JavaScript String Useful Properties and Methods:

*Ref: https://www.w3schools.com/jsref/jsref_obj_string.asp

*JavaScript provides string objects with a basic set of properties and methods, the most useful of which are described below.
*JavaScript also has special operators for concatenating: string concatenation operator (+). These string operators are binary operators in that they take two operands.
*JavaScript string properties and methods:

Methods and properties	Description
+	*String concatenation operator (+): The string concatenation operator simply joins (concatenates) two or more strings together. *Example: After execution of the code name = “Emily”; greeting = “Hello, ” + name + “!”; *The variable greeting contains the string “Hello, Emily!” *The + operator for string concatenation can convert other variable types to string if necessary. Because string concatenation occurs between two strings, JavaScript must convert other variable types to strings before it can proceed with the operation. For example: if a variable age has an integer value equal to 21, then the expression “my age is “ + age evaluates to the string “my age is 21”. JavaScript converts the value of age to a string and concatenates with the existing string literal “my age is”. *Note: Using a comma instead of the concatenation operator (+) when trying to join two strings together is a logic error and will produce incorrect results.

${…}	*Formatted (template) strings: A common programming task is to combine static content with data values to produce a string that can be presented to the user.
	
	*The traditional way to do this is through string concatenation, which is the approach we saw above, as follows:
	
	let message = "It is " + weather + " today";
	
	*JavaScript also now supports template strings, which allow data values to be specified inline, which can help reduce errors and result in a more natural development experience.
	
	*The following shows the use of a template string:
	
	//Using a template string:
	
	let messageFunction = function (weather)
	
	{

	let message = `It is ${weather} today`; console.log(message); } *Template strings begin and end with backticks (the ` character), and data values are denoted by curly braces preceded by a dollar sign.  *This string, for example, incorporates the value of the weather variable into the template string:  let message = `It is ${weather} today`; *When this function is invoked with the method call: messageFunction("raining"); It generates the console output: It is raining today
length	This property returns the number of characters in the string (length of the string).
charAt(index)	This method returns a string containing the character at the specified index.
concat(string)	This method returns a new string that concatenates together the string on which the method is called and the string provided as an argument.
indexOf(term, start)	This method returns the first index at which term appears in the string or -1 if there is no match. The optional start argument specifies the start index for the search.
replace(term, newTerm)	This method returns a new string in which all instances of term are replaced with newTerm.
substr()	Returns the characters in a string beginning at the specified location through the specified number of characters.
slice(start, end)	This method returns a substring containing the characters between the start and end indices.
split(term)	This method splits up a string into an array of values that were separated by term.
toLowerCase()	Returns a new string with the calling string value converted to lowercase.
toUpperCase()	Returns a new string with the calling string value converted to uppercase.
trim()	This method returns a new string from which all the leading and trailing whitespace characters have been removed.

Table: JavaScript string useful properties and methods.

JavaScript Primitive Object: number Object:

*Numbers in JavaScript are double-precision 64-bit format IEEE 754 values. They are immutable, just as strings. Following are the valid examples of a JavaScript Numbers:

120.27

0.26

*JavaScript provides the number object as wrappers for numbers. The wrapper provides properties and methods for manipulating numbers.

*JavaScript automatically creates Number objects to store numeric values.

*Although you can explicitly create Number objects with the new operator, normally the JavaScript interpreter creates them as needed.

*The number type is used to represent both integer and floating-point numbers, also known as real numbers.

*The following provides a demonstration of number objects:

```
//Defining number values:

let daysInWeek = 7;

let pi = 3.14;

let hexValue = 0xFFFF;
```

*You don't have to specify which kind of number you are using. You just express the value you require, and JavaScript will act accordingly.

*In the example, we have defined an integer value, defined a floating-point value, and prefixed a value with 0x to denote a hexadecimal value.
*Converting JavaScript Numbers to Strings: If you are working with multiple number variables and want to concatenate them as strings, then you can convert the numbers to strings with the toString method, as shown below:

```
//Using the number.toString method:
let myData1 = (5).toString() + String(5);
console.log("Result: " + myData1);
```

*Notice that we placed the numeric value in parentheses, and then I called the toString method. This is because you have to allow JavaScript to convert the literal value into a number before you can call the methods that the number type defines.
*We've also shown an alternative approach to achieve the same effect, which is to call the String function and pass in the numeric value as an argument.
*Both of these techniques have the same effect, which is to convert a number to a string, meaning that the + operator is used for string concatenation and not addition.
*The output from this script is as follows:

```
Result: 55
```

*There are some other methods that allow you to exert more control over how a number is represented as a string. We briefly describe these methods below.
*All of the methods shown in the table are defined by the number type:
*Ref: https://www.w3schools.com/js/js_number_methods.asp

Method	Description
toString()	*This method returns a string that represents a number in base 10.
toString(2), toString(8), toString(16)	*This method returns a string that represents a number in binary, octal, or hexadecimal notation.
toFixed(n)	*This method returns a string representing a real number with the n digits

	after the decimal point.
toExponential(n)	*This method returns a string that represents a number using exponential notation with one digit before the decimal point and n digits after.
toPrecision(n)	*This method returns a string that represents a number with n significant digits, using exponential notation if required.

Table: Useful number-to-string methods of the number type.

*Converting JavaScript Strings to Numbers: The complementary technique is to convert strings to numbers so that you can perform addition rather than concatenation.

*You can do this with the Number function, as shown below:

```
//Converting strings to numbers:

let firstVal = "5";

let secondVal = "5";

let result = Number(firstVal) + Number(secondVal);

console.log("Result: " + result);
```

*The output from this script is as follows:

Result: 10

*The Number function is strict in the way that is parses string values, but there are two other functions you can use that are more flexible and will ignore trailing non-number characters. These functions are parseInt and parseFloat. All three methods in described in the table below.
*All of the methods shown in the table are defined by the number type:
*Ref: https://www.w3schools.com/js/js_number_methods.asp

Method	Description
Number(string)	*This method parses the specified string to create an integer or real value.
parseInt(string)	*This method parses the specified string to create an integer value. *Function converts its string argument to an integer. For example: var digit30 = parseInt(“30”);
parseFloat(string)	This method parses the specified string to create an integer or real value.

Table: Useful string-to-number methods of the number type.
*JavaScript Number NaN Property: The Number property NaN represents "Not-a-Number" value.
*This property indicates that a value is not a legal number.
*The NaN property is the same as the Number.Nan property.
*Note: Use the isNaN() global function to check if a value is a NaN value.

*Syntax: Number.NaN

JavaScript Primitive Object: boolean Object:

*A boolean in JavaScript can be either true or false. If a number is zero, it defaults to false. If an empty string defaults to false.

*Following are the valid examples of a JavaScript Boolean:

```
true      // true

false     // false

0         // false

1         // true

""        // false

"hello"   // true
```

*JavaScript provides the Boolean object as a wrapper for boolean true/false values. The wrapper defines methods and properties for manipulating boolean values. When JavaScript requires a boolean value, JavaScript automatically creates a Boolean object to store the value. JavaScript programmers may also explicitly create Boolean objects using the new operator.

*The boolean type has two values: true and false.

*The following shows both values being used, but this type is most useful when used in conditional statements, such as an if statement.

```
//Defining boolean values:

let firstBool = true;

let secondBool = false;
```

JavaScript Primitive Object: Date Object:

*JavaScript's Date object provides methods for date and time manipulations, in both the computer's local time zone or based on World Time Standards Coordinated Universal Time, abbreviated as UTC, formerly called Greenwich Mean Time (GMT). Some Date object methods are:

*Date(): default constructor creates the curent date and time.

*getHours(): Get the hour (0 – 23) in the Date object.

JavaScript Objects with Properties and Methods:

*JavaScript supports Object concept very well. We can create an object using the object literal as follows:

```
var emp = {
  name: "Zara",
  age: 10
};
```

*We can write and read properties of an object using the dot notation as follows:

```
//Getting object properties
emp.name  // ==> Zara
emp.age   // ==> 10
//Setting object properties
emp.name = "Daisy"  // <== Daisy
emp.age  =  20      // <== 20
```

*There are several ways to create objects in JavaScript.

*Creating a JavaScript Object with the new Operator and Assigning Properties to the Object: The following gives a simple example:

```
//Creating an Object:
let myData = new Object();
myData.name = "Adam";
myData.weather = "sunny";
console.log("Hello " + myData.name + ".");
console.log("Today is " + myData.weather + ".");
```

*We create an object by calling new Object(), and we assign the result, the newly created object, to a variable called myData. Once the object is created, we can define properties on the object just by assigning values, like this:

```
myData.name = "Adam";
```

*Prior to this statement, myData object doesn't have a property called name. When the statement has executed, the property does exist, and it has been assigned the value Adam.

*We can read the value of a property by combining the variable name and the property name with a period, like this:

```
console.log("Hello " + myData.name + ".");
```

*The above produces the following output:

```
Hello Adam.
Today is sunny.
```

*Creating a JavaScript Object as an Object Literal with Properties and Methods: We can define an object and its properties in a single step using the object literal format, as shown below:

```
//Using the Object literal format to create an Object:
let myData =
{
  name: "Adam",
  weather: "sunny"
};
console.log("Hello " + myData.name + ". ");
```

```
console.log("Today is " + myData.weather + ".");
```
*Each property that we want to define is separated from its value using a colon (:), and properties are separated using a comma (,). The effect is the same as in the previous example.
*The above produces the following output:
Hello Adam.
Today is sunny.
*JavaScript allows you to add methods to objects. A function defined on an object is called a method. The following shows how you can add methods in this manner:
```
//Adding methods to an Object:
let myData =
{
  name: "Adam",
  weather: "sunny",
  printMessages: function ()
  {
    console.log("Hello " + this.name + ". ");
    console.log("Today is " + this.weather + ".");
  }
};
myData.printMessages();
```
*Here, we've used a function to create a method called printMessages. Notice that to refer to the properties defined by the object, we've to use the this keyword. When a function is used as a method in the code segment, the function is implicitly passed the object on which the method has been called as an argument through the special variable this.
*The above produces the following output:
Hello Adam.
Today is sunny.
*Enumerating a JavaScript Object's Properties and Methods in a for … in Statement: We can enumerate the properties and methods that an object has using the for...in statement. The following shows how we can use this statement.
```
//Enumerating an Object's properties and methods:
<!DOCTYPE HTML>
<html>
<head>
  <title>Example</title>
  <script type="text/javascript">
    var myData =
    {
      name: "Adam",
      weather: "sunny",
      printMessages: function()
      {
        console.log("Hello " + this.name + ". ");
        console.log("Today is " + this.weather + ".");
      }
    };
    for (var prop in myData)
    {
      console.log("Name: " + prop + " Value: " + myData[prop]);
    }
  </script>
</head>
<body>
  This is a simple example
</body>
</html>
```

*The for...in loop performs the statement in the code block for each property in the myData object. The prop variable is assigned the name of the property being processed in each iteration. We use the array-index style to retrieve the value of the property from the object.
*The output from this example is as follows:
Name: name Value: Adam
Name: weather Value: sunny
Name: printMessages Value: function () {console.log("Hello " + this.name + ". "); console.log("Today is " + this.weather + "."); }
*From the result, we can see that the function defined as a method is also enumerated. This is as a result of the flexible way that JavaScript handles functions.
*The Difference Between JavaScript Arrays and JavaScript Objects: In JavaScript, arrays use numbered indexes, while objects use named indexes. Arrays are a special kind of objects, with numbered indexes.
*Associative Arrays Cannot be Created with JavaScript Arrays, but can be Created with JavaScript Objects: Many programming languages support arrays with named indexes. Arrays with named indexes are called associative arrays (or hashes). JavaScript does not support arrays with named indexes, but can be created with objects.
*Creating and Populating a JavaScript Indexable Object as a JavaScript Associative Array (Map): Indexable objects associate a key with a value, creating a map-like collection that can be used to gather related data items together.
*In the following, we've used an indexable object to collect together information about multiple cities.

```
//Using indexable arays as a map:
let cities: { [index: string]: [string, string] } = {};
cities["London"] = ["raining", "38"];
cities["Paris"] = ["sunny", "52"];
cities["Berlin"] = ["snowing", "23"];
for (let key in cities)
{
  console.log(`${key}: ${cities[key][0]}, ${cities[key][1]}`);
}
```

*The cities variable is defined as an indexable object, with the key as a string and the data value a [string, string] tuple. Values are assigned and read using array-style indexers, such as cities["London"].
*The collection of keys in an indexable type can be accessed using a for...in loop, as shown above, which produces the following output in the browser's JavaScript console:
London: raining, 38
Paris: sunny, 52
Berlin: snowing, 23
*Only number and string values can be used as the keys for indexable objects, and this is a helpful feature.

JavaScript Data Structures: JavaScript Arrays and JavaScript Maps:
*The built-in JavaScript data structures are Arrays and Maps. We explore both of these data structures.

JavaScript Data Structures: JavaScript Arrays:
*Ref: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
*JavaScript arrays work like arrays in most other programming languages.
*The Difference between JavaScript Arrays and JavaScript Objects: In JavaScript, arrays use numbered indexes, while objects use named indexes. Arrays are a special kind of objects, with numbered indexes. JavaScript does not support arrays with named indexes. In JavaScript, arrays always use numbered indexes.
*Associative Arrays cannot be Created with JavaScript Arrays, but can be Created with JavaScript Objects: Many programming languages support arrays with named indexes. Arrays with named indexes are called associative arrays (or hashes). JavaScript does not support arrays with named indexes.
*Creating and Populating a JavaScript Array Using an Array Literal: We can define arrays using the array literal, in a single statement, as follows:

```
//Using the array literal to create and populate an array:
var x = [];
var y = [1, 2, 3, 4, 5];
let myArray = [100, "Adam", true];
```

*In this example, we specified that the myArray variable should be assigned a new array by specifying the items we wanted in the array between square brackets ([and]).

*Creating and Populating a JavaScript Array by Creating an Array Object with the new Operator: An array can be created by creating an Array object with the new operator, and the array can be populated by assigning values at indices.
*The following listing shows how you can create and populate an array:

```
//Creating and populating an array:
let myArray = new Array();
myArray[0] = 100;
myArray[1] = "Adam";
myArray[2] = true;
```

*We've created a new array by calling new Array(). This creates an empty array, which we assign to the variable myArray. In the subsequent statements, we assign values to various index positions in the array.
*There are a couple of important things to note in this example:
*First, we didn't need to declare the number of items in the array when we created it. JavaScript arrays will resize themselves to hold any number of items.
*The second point is that we didn't have to declare the data types that the array will hold. Any JavaScript array can hold any mix of data types. In example, we've assigned three items to the array: a number, a string, and a boolean.
*Array elements don't all have to be the same type of value. Elements can be any kind of JavaScript value, even other arrays. For example, the following declares an array that holds another array as an element:

```
let hodgepodge = [100, "paint", [200, "brush"], false];
console.log(hodgepodge);
```

*The output from the listing is as follows:

```
[100, "paint", [200, "brush"], false]
```

*Creating and Populating a JavaScript Fixed-Length Arrays (JavaScript Tuples): Tuples are fixed-length arrays, where each item in the array is of a specified type. This is a vague-sounding description because tuples are so flexible. As an example, the following uses a tuple to represent a city and its current weather and temperature.

```
//Using a tuple:
let tuple: [string, string, number];
tuple = ["London", "raining", 38]
console.log(`It is ${tuple[2]} degrees C and ${tuple[1]} in ${tuple[0]}`);
```

*Tuples are defined as an array of types, and individual elements are accessed using array indexers.
*The output from the listing is as follows:

```
It is 38 degrees C and raining in London
```

*The JavaScript Array length Property: An array has a length property that is useful for iteration:

```
var x = [1, 2, 3, 4, 5];
for (var i = 0; i < x.length; i++) {
  // Do something with x[i]
}
```

*Reading and Modifying the Contents of a JavaScript Array: You read the value at a given index using square braces ([and]), placing the index you require between the braces, like this: myArray[3]. JavaScript arrays begin at 0, so the first element will always be inside [0].

```
//Reading the data from an Array index:
let myArray = [100, "Adam", true];
console.log("Index 0: " + myArray[0]);
```

*The output from the listing is as follows:

```
Index 0: 100
```

*To get the last element, you can use brackets and `1` less than the array's length property.

```
myArray[myArray.length - 1];
```

Which will hold the value true.
*This also works for setting an element's value. You can modify the data held in any position in a JavaScript array simply by assigning a new value to the index. Just as with regular variables, you can switch the data type at an index without any problems.

```
//Modifying the contents of an array.
let myArray = [100, "Adam", true];
myArray[0] = "Tuesday";
console.log("Index 0: " + myArray[0]);
```

*In this example, we've assigned a string to position 0 in the array, a position that was previously held by a number and produces this output:
Index 0: Tuesday
*Using the JavaScript Built-in Array Properties and Methods: The JavaScript Array object defines a number of methods that you can use to work with arrays, the most useful of which are described below.
*Ref: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Method	Description
pop()	This method removes the last element from an array and returns that element.
push(item)	This method appends one or more elements to the end of an array and returns the new length of the array. *This is equivalent to: myArray[myArray.length] = someValue;
shift()	This method removes and returns the first element in the array.
unshift(item)	This method inserts a new item at the start of the array.
concat(otherArray)	This method returns a new array that concatenates the array on which it has been called with the array specified as the argument. Multiple arrays can be specified.
join(separator)	This method joins all the elements in the array to form a string. The argument specifies the character used to delimit the items.
reverse()	This method returns a new array that contains the elements in reverse order: the first becomes the last, and the last becomes the first.
slice(start, end)	This method returns a section of the array.
sort()	This method sorts the elements of an array. An optional comparison function can be used to perform custom comparisons.
splice(index, count)	This method removes count items from the array, starting at the specified index. The removed items are returned as the result of the method.
every(test)	This method calls the test function for each item in the array and returns true if the function returns true for all of them and false otherwise.
some(test)	This method returns true if calling the test function for each item in the array returns true at least once.
filter(test)	This method returns a new array containing the items for which the test function returns true.
find(test)	This method returns first item in the array for which the test function returns true.
findIndex(test)	This method returns the index of the first item in the array for which the test function returns true.
forEach(function)	This method calls the function for each item in the array.
includes(value)	This method returns true if the array contains the specified value.
map(function)	This method returns a new array containing the result of invoking the function for every item in the array.
reduce(function)	This method returns the accumulated value produced by invoking the function for every item in the array.

Table: Useful Array properties and methods.
*Example: Using the Array object: Since many of the methods in the table above return a new array, these methods can be chained together to process a data array, as shown below:

```
//Processing a data Array:
let products = [ { name: "Hat", price: 24.5, stock: 10 }, { name: "Kayak", price: 289.99, stock: 1 },
                 { name: "Soccer Ball", price: 10, stock: 0 }, { name: "Running Shoes", price: 116.50, stock: 20 } ];
let totalValue = products.filter(item => item.stock > 0).reduce((prev, item) => prev + (item.price * item.stock), 0);
console.log("Total value: $" + totalValue.toFixed(2));
```

*We use the filter method to select the items in the array whose stock value is greater than zero and use the reduce method to determine the total value of those items, producing the following output:
Total value: $2864.99
*Enumerating the Contents of a JavaScript Array: You enumerate the content of an array using a for loop or using the forEach method, which receives a function that is called to process each element in the array. Both approaches are shown below.

*The JavaScript for loop works just the same way as loops in many other languages. You determine how many elements there are in the array by using the length property.

```
//Enumerating the contents of an array:
let myArray = [100, "Adam", true];
for (let i = 0; i < myArray.length; i++)
{
  console.log("Index " + i + ": " + myArray[i]);
}
```

*The function passed to the forEach method is given two arguments: the value of the current item to be processed and the position of that item in the array.

```
myArray.forEach(function(value, index) { console.log("Index " + index + ": " + value);
```

*In this listing, we've used an arrow function as the argument to the forEach method and is the kind of use for which they excel, and we'll see it used many times.

```
myArray.forEach((value, index) => console.log("Index " + index + ": " + value));
```

*The output from each of the listing is the same, and is as follows:

```
Index 0: 100
Index 1: Adam
Index 2: true
```

JavaScript Data Structures: JavaScript Maps:

*Ref: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

*The Map object holds key-value pairs. Any value, both objects and primitive values, may be used as either a key or a value. The keys of a Map can be any value, including functions, objects, and any primitive.

new Map([iterable])

*Parameters: iterable: An Array or other iterable object whose elements are key-value pairs, which are arrays with two elements of the form [key, value], such as for example [[1, 'one'], [2, 'two']]. Each key-value pair is added to the new Map; null values are treated as undefined.

*A Map object iterates its elements in insertion order; a for...of loop returns an array of [key, value] for each iteration.

*It should be noted that a Map which is a map of an object, especially a dictionary of dictionaries, will only map to the object's insertion order, which is random and not ordered.

*Key equality: Key equality is based on the "SameValueZero" algorithm: NaN is considered the same as NaN (even though NaN !== NaN) and all other values are considered equal according to the semantics of the === operator (the identity operator).

*Using the JavaScript Built-in Map Properties and Methods: The JavaScript Map object defines a number of methods that you can use to work with maps, the most useful of which are described below.

*Ref: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Method	Description
Size	Returns the number of key/value pairs in the Map object.
clear()	Removes all key/value pairs from the Map object.
set(key, value)	Sets the value for the key in the Map object. Returns the Map object.
get(key)	Returns the value associated to the key, or undefined if there is none.
has(key)	Returns a boolean asserting whether a value has been associated to the key in the Map object or not.
keys()	Returns a new Iterator object that contains the keys for each element in the Map object in insertion order.
values()	Returns a new Iterator object that contains the values for each element in the Map object in insertion order.
delete(key)	Returns true if an element in the Map object existed and has been removed, or false if the element does not exist. Map.prototype.has(key) will return false afterwards.
entries()	Returns a new Iterator object that contains an array of [key, value] for each element in the Map object in insertion order.
forEach(callbackFn[, thisArg])	Calls callbackFn once for each key-value pair present in the Map object, in insertion order. If a thisArg parameter is provided to forEach, it will be used as the this value for each callback.

Table: Useful Map properties and methods.

*Example: Using the JavaScript Map object:

```
let myMap = new Map();
let keyString = 'a string', keyObj = {}, keyFunc = function() {};
//Setting the values:
myMap.set(keyString, "value associated with 'a string'");
myMap.set(keyObj, 'value associated with keyObj');
myMap.set(keyFunc, 'value associated with keyFunc');
myMap.size; //3
//Getting the values:
myMap.get(keyString);      //"value associated with 'a string'"
myMap.get(keyObj);         //"value associated with keyObj"
myMap.get(keyFunc);        //"value associated with keyFunc"
myMap.get('a string');     //"value associated with 'a string'", because keyString === 'a string'
myMap.get({});             //undefined, because keyObj !== {}
myMap.get(function() {}) //undefined, because keyFunc !== function () {}
```

*Creating JavaScript Maps from an Array of Key-Value Pairs: Key-value pairs are arrays with two elements of the form [key, value]. The following creates a map from an array of key-value pairs:

```
let kvArray = [['key1', 'value1'], ['key2', 'value2']];
//Use the regular Map constructor to transform a 2D key-value Array into a map
let myMap = new Map(kvArray);
myMap.get('key1');  //Returns "value1"
//Use the Array.from function to transform a map into a 2D key-value Array:
console.log(Array.from(myMap));  //Will show you exactly the same Array as kvArray
//Or, use the keys or values iterators and convert them to an array:
console.log(Array.from(myMap.keys()));  //Will show ["key1", "key2"]
```

*Using NaN as JavaScript Map Keys: NaN can also be used as a key. Even though every NaN is not equal to itself (NaN !== NaN is true), the following example works because NaNs are indistinguishable from each other:

```
var myMap = new Map();
myMap.set(NaN, 'not a number');
myMap.get(NaN);          //"not a number"
var otherNaN = Number('foo');
myMap.get(otherNaN);  //"not a number"
```

*Iterating JavaScript Maps with for..of: Maps can be iterated using a for..of loop:

```
let myMap = new Map();
myMap.set(0, 'zero');
myMap.set(1, 'one');
for (let [key, value] of myMap)
{
  console.log(key + ' = ' + value);
}
```

*Produces the following output:

```
0 = zero
1 = one
```

*The following iterates through all the keys in the map:

```
for (var key of myMap.keys())
{
  console.log(key);
}
```

*Produces the following output:
0
1
*The following iterates through all the values in the map:
```
for (var value of myMap.values())
{
  console.log(value);
}
```
*Produces the following output:
zero
one
*The following iterates through all the entries in the map:
```
for (var [key, value] of myMap.entries())
{
  console.log(key + ' = ' + value);
}
```
*Produces the following output:
0 = zero
1 = one
*Iterating JavaScript Maps with forEach(): Maps can be iterated using the forEach() method:
```
myMap.forEach(function(value, key)
{
  console.log(key + ' = ' + value);
});
```
*Produces the following output:
0 = zero
1 = one

JavaScript Classes with Properties and Methods:

*Classes are templates that are used to create objects that have identical functionality.
*Support for classes is a recent addition to the JavaScript specification intended to make working with JavaScript more consistent with other mainstream programming languages, and they are used throughout Angular development.
*The following shows how the object defined in the previous section can be expressed using a class.
```
//Defining a class:
class MyClass
{
  constructor(name, weather)
  {
    this.name = name;
    this.weather = weather;
  }
  printMessages()
  {
    console.log("Hello " + this.name + ". ");
    console.log("Today is " + this.weather + ".");
  }
}
let myData = new MyClass("Adam", "sunny");
myData.printMessages();
```
*The class keyword is used to declare a class, followed by the name of the class, which is MyClass in this case.
*The constructor function, which can receive parameters, is invoked when a new object is created using the class, and it provides an opportunity to receive data values and do any initial setup that the class requires. In the example, the constructor defines name and weather parameters that are used to create variables with the same names.

*Variables defined like this are known as properties. Classes can have methods, which defined as functions, albeit without needing to use the function keyword. There is one method in the example, called printMessages, and it uses the values of the name and weather properties to write messages to the browser's JavaScript console.
*Classes can also have static methods, denoted by the static keyword. Static methods belong to the class rather than the objects they create. We illustrate static methods later.
*The new keyword is used to create an object from a class, like this:
let myData = new MyClass("Adam", "sunny");
*This statement creates a new object using the MyClass class as its template. MyClass is used like a function in this situation, and the arguments passed to it will be received by the constructor function defined by the class. The result of this expression is a new object that is assigned to a variable called myData.
*Once you have created an object, you can access its properties and methods through the variable to which it has been assigned, like this:
myData.printMessages();
*The result from this listing is as follows:
Hello Adam.
Today is sunny.
*Defining JavaScript Class Properties and Properties Setter and Getter Methods: JavaScript classes can define properties in their constructor, resulting in a variable that can be read and modified elsewhere in the application.
*Getters and setters appear as regular properties outside of the class, but they allow the introduction of additional logic, which is useful for validating or transforming new values or generating values programmatically, as below:

```
//Using properties getters and setters:
class MyClass
{
  constructor(name, weather)
  {
    this.name = name;
    this._weather = weather;
  }
  set weather(value)
  {
    this._weather = value;
  }
  get weather()
  {
    return `Today is ${this._weather}`;
  }
  printMessages()
  {
    console.log("Hello " + this.name + ". ");
    console.log(this._weather);
  }
}
let myData = new MyClass("Adam", "sunny");
myData.printMessages();
```

*The getter and setter are implemented as functions preceded by the get or set keyword.
*There is no notion of access control in JavaScript classes. There is a convention of prefixing the names of internal properties and methods with an underscore (the _ character), but this is just a warning to other developers and is not enforced. However, TypeScript provides access modifiers, as shown in the next section.
*In the above listing, the weather property is implemented with a setter that updates a property called _weather and a getter that incorporates the _weather value in a template string.
*The result from this listing is as follows:
Hello Adam.
Today is sunny
*Defining TypeScript Class Properties and Methods Access Modifiers: As noted above, JavaScript doesn't support access protection, which means that classes, their properties, and their methods can all be accessed from any part of

the application. There is a convention of prefixing the name of internal properties and methods with an underscore (the _ character), but this is just a warning to other developers and is not enforced.
*TypeScript provides three keywords that are used to manage access and that are enforced by the compiler at runtime in the browser, and not at compile time and are not so useful during development. They become more important when you come to deploy the application, and it is important to ensure that any property or method that is accessed in a data binding expression is marked as public or has no access modified, which has the same effect as using the public keyword.
*These TypeScript access modifier keywords are described in the table below:

Keyword	Description
public	*This keyword is used to denote a property or method that can be accessed anywhere. *This is the default access protection if no keyword is used.
private	*This keyword is used to denote a property or method that can be accessed only within the class that defines it.
protected	*This keyword is used to denote a property or method that can be accessed only within the class that defines it or by classes that extend that class.

Table: The TypeScript access modifier keywords.
*Example: The following adds a private method to the TempConverter class.

```
//Using an access modifier in the tempConverter class:
class TempConverter
{
  static convertFtoC(temp: any): string
  {
    let value: number;
    if ((temp as number).toPrecision)
    {
      value = temp;
    }
    else if ((temp as string).indexOf)
    {
      value = parseFloat(<string>temp);
    }
    else
    {
      value = 0;
    }
    return TempConverter.performCalculation(value).toFixed(1);
  }
  private static performCalculation(value: number): number
  {
    return (parseFloat(value.toPrecision(2)) - 32) / 1.8;
  }
}
```

*The performCalculation method is marked as private, which means that the TypeScript compiler will report an error code if any other part of the application tries to invoke the method.
*Defining JavaScript Class Static Methods: Classes can also have static methods, denoted by the static keyword. Static methods belong to the class rather than the objects they create.
*The following shows a class called as TempConverter with a static method called convertFtoC():

```
//Class TempConverter:
class TempConverter
{
  static convertFtoC(temp)
  {
    return ((parseFloat(temp.toPrecision(2)) - 32) / 1.8).toFixed(1);
  }
}
```

*The TempConverter class contains a simple static method called convertFtoC that accepts a temperature value expressed in degrees Fahrenheit and returns the same temperature expressed in degrees Celsius.
*Defining TypeScript Class Properties, Methods and Variables with TypeScript Type Annotations: The headline TypeScript feature is support for type annotations, which can help reduce common JavaScript errors by applying type checking when the code is compiled, in a way that is reminiscent of strongly-typed languages like C# or Java. Type annotations can go a long way to preventing the most common errors.
*To show the kind of problem that type annotations solve, we create a class called TempConverter with codes:

```
//Class TempConverter:
class TempConverter
{
  static convertFtoC(temp)
  {
    return ((parseFloat(temp.toPrecision(2)) - 32) / 1.8).toFixed(1);
  }
}
```

*The TempConverter class contains a simple static method called convertFtoC that accepts a temperature value expressed in degrees Fahrenheit and returns the same temperature expressed in degrees Celsius.
*The problem is that there are some assumptions in this code that are not explicit. The convertFtoC method expected to receive a number value, on which the toPrecision method is called to set the number of floating-point digits. The method returns a string, since the result of the toFixed method is a string, although that is difficult to tell without inspecting the code carefully.
*These implicit assumptions lead to problems, especially when one developer is using JavaScript code written by another developer.
*In the code below, we've deliberately created an error by passing the temperature as a string value, instead of the number that the method expects.

```
//Using the wrong type:
let cTemp = TempConverter.convertFtoC("38");
console.log(`The temp is ${cTemp}C`);
```

*When the code is executed by the browser, you will see the following message in the browser's JavaScript console:
temp.toPrecision is not a function
*This kind of issue can be fixed without using TypeScript, of course, but it does mean that a substantial amount of the code in any JavaScript application is given over to checking the types that are being used.
*The TypeScript solution is to make type enforcement the job of the compiler, using type annotations that are added to the JavaScript code. In the listing below, we've added type annotations to the TempConverter class:

```
//Class TempConverter with type annotations:
class TempConverter
{
  static convertFtoC(temp: number) : string
  {
    return ((parseFloat(temp.toPrecision(2)) - 32) / 1.8).toFixed(1);
  }
}
```

*Type annotations are expressed using a colon (the : character) followed by the type. There are two annotations in the example. The first specifies that the parameter to the convertFtoC method should be a number, as in:

```
static convertFtoC(temp: number) : string { //Omitted }
```

*The other annotation specifies that the result of the method is a string, as in:

```
static convertFtoC(temp: number) : string { //Omitted }
```

*When we save this file, the TypeScript compiler will run. Among the errors that are reported will be this one:
error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.
*The TypeScript compiler has examined that the type of the value passed to the convertFtoC method doesn't match the type annotation and has reported an error. This is the core of the TypeScript type system, and it means we don't have to write additional code in your classes to check that you have received the expected types, and it also makes it easy to determine the type of a method result.
*To resolve the error reported to the compiler, the following updates the statement that invokes the convertFtoC method so that it uses a number.

```
//Using the correct type:
```

```
let cTemp = TempConverter.convertFtoC(38);
console.log(`The temp is ${cTemp}C`);
```
*When you save the changes, you will see the following messages displayed in the browser's JavaScript console:
The temp is 3.3C
*TypeScript Type Annotating Properties and Variables: It isn't just methods that type annotations can be applied to. They can also be applied to properties and variables, ensuring that all of the types used in an application can be verified by the compiler.
*The following shows type annotations to the class Name:
```
//Class Name with type annotations:
class Name
{
  first: string;
  second: string;
  constructor(first: string, second: string)
  {
    this.first = first;
    this.second = second;
  }
  get nameMessage() : string
  {
    return `Hello ${this.first} ${this.second}`;
  }
}
```
*Properties are declared with a type annotation, following the same pattern as for parameter and result annotations.
*The pattern of receiving constructor parameters and assigning their values to variables is so common that TypeScript includes an optimization, as shown below:
```
//Creating properties from class constructor parameters:
class Name
{
  constructor(private first: string, private second: string)
  {
  }
  get nameMessage() : string
  {
    return `Hello ${this.first} ${this.second}`;
  }
}
```
*The keyword private is an example of an access control modifier, which we describe later. Applying the keyword to the constructor parameter has the effect of automatically defining the class property and assigning it the parameter value. The code above has the same effect as the previous listing, but is more concise.
*Note: Type Definitions for JavaScript Libraries: TypeScript works best when it has type information for all the packages that you are working with, including the ones from third parties. Some of the packages required for Angular (including the Angular framework itself) include the information that TypeScript needs. For other packages, type information must be downloaded and added to the package. This is done using the typings tool.
*Defining TypeScript Multiple Types or Any Type Class Properties, Methods and Variables with TypeScript Type Annotations: TypeScript allows multiple types to be specified, separated using a bar (the | character). This can be useful when a method can accept or return multiple types or when a variable can be assigned values of different types. The following shows modifies the convertFtoC method so that it will accept number or string values:
```
//Accepting multiple values in the convertFtoC method:
class TempConverter
{
  static convertFtoC(temp: number | string): string
  {
    let value: number = (<number>temp).toPrecision ? <number>temp : parseFloat(<string>temp);
    return ((parseFloat(value.toPrecision(2)) - 32) / 1.8).toFixed(1);
  }
```

}
*The type declaration for the temp parameter has changes to number | string, which means that the method can accept either value. This is called a union type.
*Within the method, a type assertion is used to work out which type has been received. This is a slightly awkward process, but the parameter value is cast to a number value in order to check is there is a toPrecision method defined on the result, like this:
(<number>temp).toPrecision
*The angle brackets (the < and > characters) are to declare a type assertion, which will attempt to convert an object to the specified type.
*You can also achieve the same result using the as keyword, as shown below:

```
//Using the as keyword in the convertFtoC method:
class TempConverter
{
  static convertFtoC(temp: number | string): string
  {
    let value: number = (temp as number).toPrecision ? temp as number : parseFloat(<string>temp);
    return ((parseFloat(value.toPrecision(2)) - 32) / 1.8).toFixed(1);
  }
}
```

*An alternative to specifying a union type is to use the any keyword, which allows any type to be assigned to a variable, used as an argument, or returned from a method. Note that the TypeScript compiler will implicitly apply the any keyword when you omit a type annotation. Also, at the opposite end of the spectrum is void, which is used to indicate that a method returns no result. You don't have to use void, but it makes the fact that there is no result obvious. The following replaces the union type in the convertFtoC method with the any keyword:

```
//Specifying Any Type in the convertFtoC method:
class TempConverter
{
  static convertFtoC(temp: any): string
  {
    let value: number;
    if ((temp as number).toPrecision)
    {
      value = temp;
    }
    else if ((temp as string).indexOf)
    {
      value = parseFloat(<string>temp);
    }
    else
    {
      value = 0;
    }
    return ((parseFloat(value.toPrecision(2)) - 32) / 1.8).toFixed(1);
  }
}
```

*Defining JavaScript Class Inheritance: Classes can inherit behavior from other classes using the extends keyword, as shown below:

```
//Using class inheritance:
class MyClass
{
  constructor(name, weather)
  {
    this.name = name;
    this._weather = weather;
  }
  set weather(value)
```

```
  {
    this._weather = value;
  }
  get weather()
  {
    return `Today is ${this._weather}`;
  }
  printMessages()
  {
    console.log("Hello " + this.name + ". ");
    console.log(this._weather);
  }
}
//Define class MySubClass by extending class MyClass:
class MySubClass extends MyClass
{
  constructor(name, weather, city)
  {
    super(name, weather);
    this.city = city;
  }
  printMessages()
  {
    super.printMessages();
    console.log(`You are in ${this.city}`);
  }
}
let myData = new MySubClass("Adam", "sunny", "London");
myData.printMessages();
```

*The extends keyword is used to declare the class that will be inherited from, known as the super class or base class. In the listing, the MySubClass inherits from MyClass.
*The super keyword is used to invoke the super class's constructor and methods. The MySubClass builds on the MyClass functionality to add support for a city.
*The result from this listing is as follows:

```
Hello Adam.
Today is sunny
You are in London
```

*JavaScript Classes Vs. JavaScript Prototypes: The class feature doesn't change the underlying way that JavaScript handles types. Instead, it simply provides a way to use them that is more familiar to the majority of programmers.
*Behind the scenes, JavaScript still uses its traditional type system, which is based on prototypes.
*As an example, the above code can also be written like this prototypes, as follows:

```
var MyClass = function MyClass(name, weather)
{
  this.name = name;
  this.weather = weather;
}
MyClass.prototype.printMessages = function ()
{
  console.log("Hello " + this.name + ". ");
  console.log("Today is " + this.weather + ".");
};
var myData = new MyClass("Adam", "sunny");
myData.printMessages();
```

*Angular development is easier when using classes, which is the approach that we take. A lot of the features introduced in ES6 are classified as syntactic sugar, which means that they make aspects of JavaScript easier to understand and use.

JavaScript Modules (JavaScript Files):
*JavaScript modules are used to manage the dependencies in a web application, which means you don't need to manage a set of script elements in the HTML document. Instead, a module loader is responsible for figuring out which files are required to run an application, loading those files and executing them in the right order.
*In a complex application, this is a difficult task to perform manually, and it is well-suited to automation.
*Creating JavaScript Modules: Creating modules is simple and is performed automatically when TypeScript compiles a file, because each file is treated as a module.
*The export keyword is used to denote variables and classes that can be used outside of the file, which means that any other variables or classes can be used only within the file.
*To demonstrate how modules work, we create a TypeScript file called NameAndWeather.ts, and use it to define the classes shown below:

```
//File: TypeScript file NameAndWeather.ts:
export class Name
{
  constructor(first, second)
  {
    this.first = first;
    this.second = second;
  }
  get nameMessage()
  {
    return `Hello ${this.first} ${this.second}`;
  }
}
export class WeatherLocation
{
  constructor(weather, city)
  {
    this.weather = weather;
    this.city = city;
  }
  get weatherMessage()
  {
    return `It is ${this.weather} in ${this.city}`;
  }
}
```

*The export keyword has been applied to each of the classes in the new file, which means they can be used elsewhere in the application.
*Note: Angular has a convention of defining one class per file, which means that every class in an Angular project is contained in a separate module (separate file).
*JavaScript Different Module Formats and Specifying a Module Format: There are different types of JavaScript module, each of which is supported by a different set of module loaders.
*The TypeScript compiler can be configured to produce the most popular formats, specified through the module property in the tsconfig.json file.
*The commonjs format is used for Angular development, which can be seen in the tsconfig.json file we use:

```
//File: tsconfig.json
{
  "compilerOptions":
  {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true
  },
```

```
  "exclude": [ "node_modules" ]
}
```
*This format can be handled by the SystemJS module loader that we used for the examples here.
*Importing Types from JavaScript Modules: The import keyword is used to declare dependencies on the contents of a module. There are different ways to use the import keyword, as described below:
*Importing Specific Types from JavaScript Modules: The conventional way to use the import keyword is to declare dependencies on specific types, as shown below.
*This has the advantage of minimizing the dependencies in the application, and leads to optimization for application deployment:
```
//File: Importing specific types:
import { Name, WeatherLocation } from "./modules/NameAndWeather";
let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
console.log(name.nameMessage);
console.log(loc.weatherMessage);
```
*The import keyword is followed by curly braces that contain a comma-separated list of the classes that the code in the current files depends on, followed by the from keyword, followed by the module name. Here, we've imported the Name and WeatherLocation classes from the NameAndWeather module in the modules folder. Notice that the file extension is not included when specifying the module.
*The result from this listing is as follows:
```
Hello Adam Freeman
It is raining in London
```
*Specifying Relative Module Path and Nonrelative Module Path of JavaScript Modules in the import Statement:
*We'll see two different ways of specifying modules in the import statements in this book.
*The first is a relative module, in which the name of the module is prefixed with ./, like in the above example:
```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
```
*This kind of import tells the TypeScript compiler that the module is located relative to the file that contains the import statement. In this case, the NameAndWeather.ts file is in the Angular modules directory.
*The other type of import is nonrelative. Here is an example of a nonrelative import:
```
import { Component } from "@angular/core";
```
*This import statement doesn't begin with ./ and so the TypeScript compiler resolves it using the NPM packages in the node_modules folder.
*The JavaScript module loader must also be configured so that it knows how to resolve both relative and nonrelative modules using HTTP requests.
*Importing Specific Types and Renaming Import Types (Type Aliases) from JavaScript Modules: In complex projects that have lots of dependencies, it is possible that you will need to use two classes with the same name from different modules.
*To re-create this problem, we created a file called DuplicateName.ts and defined the class shown below:
```
//File: DuplicateName.ts:
export class Name
{
  get message()
  {
    return "Other Name";
  }
}
```
*This class is called Name, which means that importing it using the approach above will cause a conflict because the compiler won't be able to differentiate between both the classes named as Name.
*The solution is to use the as keyword, which allows an alias to be created for a class when it is important from a module, as shown below:
```
//Using a module alias:
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
let other = new OtherName();
```

```
console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(other.message);
```
*The Name class in the DupliateName module is imported as OtherName, which allows it to be used without conflicting with the Name class in the NameAndWeather module.
*The result from this listing is as follows:
```
Hello Adam Freeman
It is raining in London
Other Name
```
*Importing All of the Types from a JavaScript Module: An alternative approach is to import the module as an object that has properties for each of the types it contains, as shown below:
```
//Importing a Module as an Object in the primer.ts File
import * as NameAndWeatherLocation from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
let name = new NameAndWeatherLocation.Name("Adam", "Freeman");
let loc = new NameAndWeatherLocation.WeatherLocation("raining", "London");
let other = new OtherName();
console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(other.message);
```
*The import statement in this example imports the contents of the NameAndWeather module and creates an object called NameAndWeatherLocation. This object has Name and Weather properties that correspond to the classes defined in the module.
*This example produces the same output as the previous listing.

XXXXXXXXStart

XXXXXXXXEnd

JavaScript Global Objects in the Document Object Model (DOM), and their Properties, Methods, and Events: When the browser loads and processes an HTML document, it creates the Document Object Model (DOM).

*The Document Object Model is a tree structure of various HTML elements, as follows:

```
<html>

  <head>

    <title>The jQuery Example</title>

  </head>

  <body>

    <div>

      <p>This is a paragraph.</p>

      <p>This is second paragraph.</p>

      <p>This is third paragraph.</p>
```

```
    </div>

  </body>

</html>
```

*Following are the important points about the above DOM tree structure:

*(1) The <html> is the ancestor of all the other elements; in other words, all the other elements are descendants of <html>.

*(2) The <head> and <body> elements are not only descendants, but children of <html>, as well.

*(3) Likewise, in addition to being the ancestor of <head> and <body>, <html> is also their parent.

*(4) The <p> elements are children (and descendants) of , descendants of <body> and <html>, and siblings of each other <p> elements.

*The DOM is a model where JavaScript objects are used to represent each element in the document, and the DOM is the mechanism by which you can programmatically engage with the content of an HTML document.

*Note that in principle, the DOM can be used with any programming language that the browser cares to implement. In practice, JavaScript dominates the mainstream browsers, so we're not going to differentiate between the DOM as an abstract idea and the DOM as a collection of related JavaScript objects. Thus, using the DOM means using JavaScript.

*We can use JavaScript to traverse the network of objects to learn about the nature and structure of the document that has been represented. We can also use jQuery, a JavaScript library to access the DOM.

*The browser contains a complete set of global objects that allow script programmers to access and manipulate every element of an XHTML document. These objects are part of the JavaScript Document Object Model (DOM). Each JavaScript DOM object has properties, methods and events.

*JavaScript window DOM Object: The window Object: The global window object provides properties, methods and events for manipulating browser windows. Some of the methods of the window object can be used to show message dialogs, input prompt dialogs, re-sizing and re-positioning the browser, creating fully customizable child popup browser windows so as to create websites that span multiple browser windows. The window object has many properties, methods and events, such as:

*(1) window.alert(): To display text in a pre-defined dialog in the window object called an alert dialog. The argument to this method is the string to display. The dialog provides an OK button that allows the user to close the dialog by clicking the button. Note: Dialogs display plain text; they do not render XHTML. Therefore, specifying XHTML elements as part of a string to be displayed in a dialog results in the actual characters of the tags being displayed.

*(2) window.prompt(): To display text and get an input string value back from a user in a pre-defined dialog in the window object called a prompt dialog. The argument to this method is the string to display, called as a prompt because it directs the user to take a specific action. An optional second argument, separated from the first by a comma, may specify the default string displayed in the text field. The user types characters in the text field, then clicks the OK button to submit the string in the text field to the program. If the user clicks the CANCEL button, no string value is sent to the program. Instead the program submits the value null, a JavaScript keyword signifying that a variable has no value. Note: (*) Dialogs display plain text; they do not render XHTML. Therefore, specifying

XHTML elements as part of a string to be displayed in a dialog results in the actual characters of the tags being displayed. (*) We can also get input via GUI components in XHTML forms.

*JavaScript document DOM Object: The document Object: The global document object represents the XHTML document the browser is currently displaying. The document object is used to manipulate the document that is currently visible in the browser window. The document object allows you to specify text to display in the XHTML document. The document object has many properties, methods and events, such as the following:

*(1) document.write(), document.writeln(): display in the XHTML document a string of XHTML markup characters, including CSS style attributes, contained between double quotation marks or single quotation marks. writeln() also also writes a new-line. Note: The script that is used to write XHTML markup in the XHTML document may also be encapsulated inside of a Javscript function that can be invoked as an event handler for the onload event of the <body> element of the XHTML document, which is called when the body is completely loaded.

*(2) document.getElementbyId():

*(3) document.cookie: a property that holds a string containing the values of all the cookies stored on the user's computer for the current document.

*JavaScript HTMLElement Object and Subclasses DOM Objects: HTMLElement Object and Subclasses:

*The JavaScript object that defines the basic functionality that is available in the DOM for all types of elements is called HTMLElement.

*Ref: https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement

*The HTMLElement object defines properties and methods that are common to all HTML element types, including the properties shown below:

Property	Description	Returns
className	Gets or sets the list of classes that the element belongs to.	string
Id	Gets or sets the value of the id attribute.	string
Lang	Gets or sets the value of the lang attribute.	string
tagName	Returns the tag name (indicating the element type).	string

Table: Basic HTMLElement properties.

*Many more properties are available. The exact set depends on the version of HTML you are working with. But these four are sufficient for our work.

*The DOM uses objects that are derived from HTMLElement to represent the unique characteristics of each element type. For example, the HTMLImageElement object is used to represent img elements in the DOM, and this object defines the src property, which corresponds to the src attribute of the img element.

*Without going to the element-specific objects, as a rule, we can rely on properties being available that correspond to an element's attributes.

*We can access the DOM through the global document variable, which returns a Document object. The Document object represents the HTML document that is being displayed by the browser and defines some methods that allow you to locate objects in the DOM, as described below:

Property	Description	Returns

getElementById(<id>)	Returns the element with the specified id value.	HTMLElement
getElementsByClassName(<class>)	Returns the elements with the specified class value.	HTMLElement[]
getElementsByTagName(<tag>)	Returns the elements of the specified type. *Example: var labelElems = document. getElementsByTagName("label");	HTMLElement[]
querySelector(<selector>)	Returns the first element that matches the specified CSS selector.	HTMLElement
querySelectorAll(<selector>)	Returns all of the elements that match the specified CSS selector.	HTMLElement[]

Table: Document methods to find HTMLElement elements.

*There are other methods too, but these methods are sufficient for us.

*The last two methods described in the table use CSS selectors, described later.

*The following shows how you can use the Document object to search for elements of a specific type in the document. In the script, we use the getElementsByTagName method to find all of the img elements in the document. This method returns an array of objects that we enumerate to print out the value of the tagName and src properties for each object to the console.

```
<script>

  var elements = document.getElementsByTagName("img");

  for (var i = 0; i < elements.length; i++)

  {

    console.log("Element: " + elements[i].tagName + " " + elements[i].src);

  }

</script>
```

***Modifying the DOM Objects to Dynamically Modify the HTML Document**: The objects in the DOM are live, meaning that changing the value of a DOM object property changes the document that the browser is displaying.

*The following shows a script that has this effect. We're only just showing the script element here. The rest of the HTML document is omitted:

```
//Modifying a DOM object property:
```

```
<script>

  var elements = document.getElementsByTagName("img");

  for (var i = 0; i < elements.length; i++)

  {

    elements[i].src = "snowdrop.png";

  }

</script>
```

*In this script, we set the value of the src attribute to be snowdrop.png for all of the img elements. All the img elements will now show the snowdrop.png image.

***Modifying the DOM Objects CSS Property Values Styles to Dynamically Modify the HTML Document Styles**: We can use the DOM to change the values for CSS properties.

*The DOM API support for CSS is comprehensive, but the simplest technique is to use the style property that is defined by the HTMLElement object.

*The object returned by the style property defines properties that correspond to CSS properties.

*The naming scheme of properties as defined by CSS and by the object that style returns is slightly different. For example, the background-color CSS property becomes the style.backgroundColor object property.

*The following demonstrates the use of the DOM to manage styles. We're only just showing the script element here. The rest of the HTML document is omitted:

```
//Using the DOM to modify element styles:

<script>

   var elements = document.getElementsByTagName("img");

  for (var i = 0; i < elements.length; i++)

  {

    if (i > 0)

    {

      elements[i].style.opacity = 0.5;

    }

  }
```

</script>

*In this script, we change the value of the opacity property for all but the first of the img elements in the document. We left one element unaltered so that we can see the difference.

***Handling the DOM Objects Events Sent by the Browser by Registering JavaScript Handler Functions to be Called**: Events are signals sent by the browser to indicate a change in status of one or more elements in the DOM.
*There is a range of events to represent different kinds of state change. For example, the click event is triggered when the user clicks an element in the document and the submit element is triggered when the user submits a form.
*Many events are related. For example, the mouseover event is triggered when the user moves the mouse over an element, and the mouseout event is triggered when the user moves the mouse out again.
*We can respond to an event by associating a JavaScript handler function with an event for a DOM element. The statements in the handler function will be executed each time the event is triggered. In addition to the DOM API support for event handling, jQuery also provides strong support for events.
*The following gives an example. We're only just showing the script element here. The rest of the HTML document is omitted:

```
//Handling a DOM element object event:
<script>
  var elements = document.getElementsByTagName("img");
  for (var i = 0; i < elements.length; i++)
  {
    elements[i].onmouseover = handleMouseOver;
    elements[i].onmouseout = handleMouseOut;
  }
  function handleMouseOver(e)
  {
    e.target.style.opacity = 0.5;
  }
  function handleMouseOut(e)
  {
    e.target.style.opacity = 1;
  }
</script>
```

*This script defines two handler functions, which we assign as the values for the onmouseover and onmouseout properties on the img DOM objects.
*The effect of this script is that the images become partially transparent when the mouse is over them and return to normal when the mouse exits.
*We now look at the object that is passed to the event handler functions: the Event object.
*The following table shows the most important members of the Event object.

Name	Description	Returns
type	The name of the event, such as mouseover.	string
target	The element at which the event is targeted.	HTMLElement
currentTarget	The element whose event listeners are currently being invoked.	HTMLElement
eventPhase	The phase in the event life cycle.	number
Bubbles	Returns true if the event will bubble through the document; returns false otherwise.	boolean
Cancelable	Returns true if the event has a default action that can be canceled; returns false otherwise.	boolean
stopPropagation()	Halts the flow of the event through the element tree after the event listeners for the current element have been triggered.	void
stopImmediatePropagation()	Immediately halts the flow of the event through the element tree; untriggered event listeners for the current element will be ignored.	void

preventDefault()	Prevents the browser from performing the default action associated with the event.	void
defaultPrevented	Returns true if preventDefault() has been called.	boolean

Table: Functions and properties of the Event object.

*In the previous example, we used the target property to get hold of the element for which the event was triggered.

*Some of the other members relate to event flow and to default actions, which is discussed now.

*DOM Event Flow from Element to Ancestor Elements: An event has three phases to its life cycle: capture, target, and bubbling. When an event is triggered, the browser identifies the element that the event relates to, which is referred to as the target for the event. The browser identifies all of the elements between the body element and the target and checks each of them to see whether they have any event handlers that have asked to be notified of events of their descendants. The browser triggers any such handler before triggering the handlers on the target itself. Once the capture phase is complete, the browser moves to the target phase, which is the simplest of the three phases. When the capture phase has finished, the browser triggers any listeners for the event type that have been added to the target element. Once the target phase has been completed, the browser starts working its way up the chain of ancestor elements back toward the body element. At each element, the browser checks to see whether there are listeners for the event type that are not capture-enabled. Not all events support bubbling. We can check to see whether an event will bubble using the bubbles property. A value of true indicates that the event will bubble, and false means that it won't.

*Default Action of a DOM Event on Element: Some events define a default action that will be performed when an event is triggered. As an example, the default action for the click event on the a element is that the browser will load the content at the URL specified in the href attribute. When an event has a default action, the value of its cancelable property will be true. You can stop the default action from being performed by calling the preventDefault method. Note that calling the preventDefault function doesn't stop the event flowing through the capture, target, and bubble phases. These phases will still be performed, but the browser won't perform the default action at the end of the bubble phase. You can test to see whether the preventDefault function has been called on an event by an earlier event handler by reading the defaultPrevented property. If it returns true, then the preventDefault function has been called.

XXXXXXEnd

XXXXXXStart

XXXXXXEnd

***Example: Accessing and Modifying the DOM Objects Dynamically**: Embedded JavaScript inside the XHTML documents <head> section, <script> element. JavaScript comments may be placed anywhere inside the <script> element, and single-line comments start with a //, while multi-line comments start with a /* and end with a */.

```
<html xmlns = "http://www.w3.org/1999/xhtml">

  <head>

    <title>Emedded JavaScript Example</title>

    <!-- The script element declares embedded JavaScript -->

    <script type = "text/javascript">

      <!--
```

```
//Declare a JavaScript variable to hold a string entered by a user.

var name;

//Display in the XHTML document a string of XHTML markup characters,

//including CSS style attributes.

document.writeln( "<h1 style = \"color: magenta\">Welcome to JavaScript " +

                  "Programming!</h1>" );

//Display text in an alert dialog.

window.alert( "Welcome to JavaScript " + "Programming!" );

//Display text in a prompt dialog and get user input as a string.

name = window.prompt( "Please enter your name:" );

//Display in the XHTML document a string of XHTML markup characters in multiple

//lines, includes CSS styles.

document.writeln( "<h1 style = \"color: magenta\">Hello<br/> " + name + "<br/>Great day!</h1>" );

//Get two integer strings from user. Display text in a prompt dialog and get user input as a string.

var firstNumber = window.prompt( "Enter first integer:" );

var secondNumber = window.prompt( "Enter second integer:" );

//Convert the integer strings to integers. Call parseInt(string): function converts its string

//argument to an integer.
var number1 = parseInt(firstNumber );

var number2 = parseInt(secondNumber );

//Add the two integers ented by the user.

var sum = number1 + number2;

//Display in the XHTML document a string of XHTML markup characters in multiple lines,

//includes CSS styles.

document.writeln( "<h1 style = \"color: green\">The sum is: " + sum + "</h1>" );

//Get the current date and time, and display Good Morning/Afternoon/Evening.
```

```
var now = new Date();  //Date default constructor creates the curent date and time.

//Call Date.getHours() to get the hour (0 – 23) in the Date object.

var hour = now.getHours(); //Date.getHours() gets the hour (0 – 23) in the Date object.

//Determine whether it is morning, if so print “Good Morning!”

if( hour < 12 )

{

  document.writeln( “<h1 style = \”color: blue\”>Good Morning!</h1>” );

}

//Determine whether it is afternoon, if so print “Good Afternoon!”

else if( (hour >= 12) && (hour < 6) )

{

  document.writeln( “<h1 style = \”color: blue\”>Good Afternoon!</h1>” );

}

//Determine whether it is evening, if so print “Good Evening!”

else //if( hour >= 6 )

{

  document.writeln( “<h1 style = \”color: blue\”>Good Evening!</h1>” );

}

//Using a for loop, display incremental font sizes via CSS styles.

for( var counter = 1; counter <= 7; ++counter )

{

    document.writeln( “<p style = \”font-size: “ + counter + “pt\”> HTML font size “ +

                                counter + “pt</p>” );

}

//Ask a user to select a list item style:
```

```
var choice = window.prompt( “Select a list items style: \n” +
  “1 (numbered), 2 (lettered), 3 (roman)”, “1” );
var validInput = true; //Indicates if user input is valid.
var listType; //Type of list item as a string.
var startTag, endTag; //Start list item tag and end list item tag.
switch( choice )
{
  case “1”: //Numbered List Items
    startTag = “<o1>”;
    endTag = “</o1>”;
    listType = “<h1>Numbered List Items</h1>”;
    break;
  case “2”: //Lettered List Items
    startTag = “<o1 style = \”list-style-type: upper-alpha\”>”;
    endTag = “</o1>”;
    listType = “<h1>Lettered List Items</h1>”;
    break;
  case “3”: //Roman Numbered List Items
    startTag = “<o1 style = \”list-style-type: upper-roman\”>”;
    endTag = “</o1>”;
    listType = “<h1>Roman Numbered List Items</h1>”;
    break;
  default: //If user input is not valid.
    validInput = false; //Indicates user input is not valid.
    break;
```

```
    } //switch( choice )
    //Display list items in the user selected list item style:
    if( validInput == true )
    {
      document.writeln( listType + startTag );
      for( var i = 1; i <=3; ++i )
      {
        document.writeln( "<li>List item " + i + "</li>" );
      } //for( var i = 1; i <=3; ++i )
      document.writeln( listType + startTag );
    } //if( validInput == true )
    else //if( validInput == false )
    {
      document.writeln( "Invalid user selection for list items:" + choice );
    } // else if( validInput == false )
    // -->
   </script>
  </head>
  <body>
   <!-- Enter body here -->
  <p>Click refresh to run he script again.</p>
  </body>
</html>
```

XXXXXXXXStart

XXXXXXXXEnd

XXXXXXXXStart

XXXXXXXXEnd

JSON (JavaScript Object Notation): http://json.org/, is a lightweight data-interchange format that is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

*JSON is built on two structures: (i) A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array. (ii) An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

*These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures. In JSON, they take on these forms:

(*) An object is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma).

object = { name1 : value1, name2 : value2, name3 : value3 }

(*) An array is an ordered collection of values. An array begins with [(left bracket) and ends with] (right bracket). Values are separated by , (comma).

array = [value1, value2, value3, value4, value5, value1, value1, value1,]

(*) A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested. The following is the format for the values:

A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string.

A number is very much like a C or Java number, except that the octal and hexadecimal formats are not used.

Whitespace can be inserted between any pair of tokens. Excepting a few encoding details, that completely describes the language.

*To Validate Json: https://jsonlint.com/

*RapidJSON: http://rapidjson.org/index.html. Tutorials: http://rapidjson.org/md_doc_tutorial.html

RapidJSON is a JSON parser and generator for C++. Rapid-JSON is small but complete. It supports both SAX and DOM style API. RapidJSON is memory-friendly. Each JSON value occupies exactly 16/20 bytes for most 32/64-bit

machines (excluding text string). RapidJSON is Unicode-friendly. It supports UTF-8, UTF-16, UTF-32, and their detection, validation and transcoding internally. For example, you can read a UTF-8 file and let RapidJSON transcode the JSON strings into UTF-16 in the DOM. It also supports "\u0000" (null character).

---

jQuery for Web Development Tutorials:
XXXStart jQuery - DOM Traversing https://www.tutorialspoint.com/jquery/jquery-traversing.htm

*Ref: https://www.tutorialspoint.com/jquery/jquery-overview.htm
*Ref: https://www.tutorialrepublic.com/jquery-tutorial/
*jQuery is a fast and concise JavaScript Library created by John Resig in 2006 with a nice motto: Write less, do more. jQuery simplifies HTML and XHTML document traversing, event handling, animating, and Ajax interactions for rapid web development.
*jQuery is a JavaScript toolkit designed to simplify various tasks by writing less code. Here is the list of important core features supported by jQuery:
*(1) DOM manipulation: The jQuery made it easy to select DOM elements, negotiate them and modifying their attributes and contents by using cross-browser open source selector engine called Sizzle.
*(2) Event handling: The jQuery offers an elegant way to capture a wide variety of events, such as a user clicking on a link, without the need to clutter the HTML code itself with event handlers.
*(3) AJAX Support: The jQuery helps to develop a responsive and feature-rich site using AJAX technology.
*(4) Animations: The jQuery comes with plenty of built-in animation effects which you can use in your websites.
*(5) Lightweight: The jQuery is very lightweight library - about 19KB in size (Minified and gzipped).
*(6) Cross Browser Support: The jQuery has cross-browser support, and works well in IE 6.0+, FF 2.0+, Safari 3.0+, Chrome and Opera 9.0+.
*(7) Latest Technology: The jQuery supports CSS3 selectors and basic XPath syntax.

How to Install and use jQuery: There are two ways to install and use jQuery.
*(1) jQuery Local Installation: We can download jQuery library on our local machine and include it in our HTML code in a <script> tag.
*To download jQuery, go to the https://jquery.com/download/ to download the latest version available.
*Put the downloaded jquery-2.1.3.min.js file in a directory of your website, e.g. /jquery.
*Example: We can include jquery library in your HTML file as follows:

```
//File: helloworld.htm
<html>
  <head>
    <title>The jQuery Example</title>
    <script type = "text/javascript" src = "/jquery/jquery-2.1.3.min.js"> </script>
    <script type = "text/javascript">
      $(document).ready(function() {
        document.write("Hello, World!");
      });
    </script>
  </head>
  <body>
    <h1>Hello</h1>
  </body>
</html>
```

This will produce following result:
Hello, World!
*(2) jQuery CDN Based Version: We can include jQuery library into our HTML code directly from Content Delivery Network (CDN). Google and Microsoft provides content deliver for the latest version.
*We use Google CDN version of the library throughout this tutorial.
*Example: Let us rewrite above example using jQuery library from Google CDN.

```
//File: helloworld.htm
<html>
```

```
  <head>
    <title>The jQuery Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript">
      $(document).ready(function() {
        document.write("Hello, World!");
      });
    </script>
  </head>
  <body>
    <h1>Hello</h1>
  </body>
</html>
```

This will produce following result:

Hello, World!

*Using Multiple External Libraries: We can also use multiple JavaScript libraries without conflicting each other, impoerting each with a <script> tag. For example, you can use jQuery and MooTool javascript libraries together.

How to Call a jQuery Library Functions: In almost everything we do when using jQuery reads or manipulates the document object model (DOM), adding events etc., should only be done as soon as the DOM is ready.

*This is done by putting the jQuery codes inside the $(document).ready() function. Everything inside it will load as soon as the DOM is loaded and before the page contents are loaded.

*To do this, we register a ready event for the document as follows:

```
$(document).ready(function() {
  //do stuff when DOM is ready
});
```

*Example: To call upon any jQuery library function, use HTML script tags as shown below:

```
//File: helloworld-alertbox.htm
<html>
  <head>
    <title>The jQuery Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        $("div").click(function() {alert("Hello, world!");});
      });
    </script>
  </head>
  <body>
    <div id = "mydiv">
      Click on this to see an alert box.
    </div>
  </body>
</html>
```

This will produce following result:

Click on this to see an alert box.

How to Use jQuery External Scripts: It is better to write an external jQuery JavaScript file, such as external.js as below, and import that it our HTML page with the <script> tag:

*Example: Importing an external jQuery JavaScript file:

```
//File: externalscript.js
$(document).ready(function() {
```

```
  $("div").click(function() {
    alert("Hello, world!");
  });
});
```
*Now we can import custom.js file in our HTML file as follows:
```
//File: helloworld-external-script-alertbox.htm
<html>
  <head>
    <title>The jQuery Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" src = "/jquery/externalscript.js">
    </script>
  </head>
  <body>
    <div id = "mydiv">
      Click on this to see an alert box.
    </div>
  </body>
</html>
```
This will produce following result:
Click on this to see an alert box.

jQuery Basics: jQuery is a framework built using JavaScript capabilities. So, we can use all the functions and other capabilities available in JavaScript. WE explain most basic concepts but frequently used in jQuery.
*jQuery makes a use of anonymous functions very frequently as follows:
```
$(document).ready(function(){
  // do some stuff here
});
```
*JavaScript Context: The JavaScript keyword this always refers to the current context. Within a function this context can change, depending on how the function is called:
```
$(document).ready(function() {
  //this refers to window.document
});
$("div").click(function() {
  //this refers to a div DOM element
});
```
*JavaScript CallbackS: A callback is a plain JavaScript function passed to some method as an argument. Some callbacks are just events, called to give the user a chance to react when a certain state is triggered. jQuery's event system uses such callbacks everywhere for example:
```
$("body").click(function(event) {
  console.log("clicked: " + event.target);
});
```
Most callbacks provide arguments and a context. In the event-handler example, the callback is called with one argument, an Event. Some callbacks are required to return something, others make that return value optional. To prevent a form submission, a submit event handler can return false as follows:
```
$("#myform").submit(function() {
  return false;
});
```
*jQuery and the Document Object Model: The Document Object Model is a tree structure of various elements of HTML. jQuery is most often used to traverse, find and manipulate DOM elements contents, attributes, and events.

jQuery Selectors: The jQuery library harnesses the power of Cascading Style Sheets (CSS) selectors to let us quickly and easily access elements or groups of elements in the Document Object Model (DOM).

*jQuery Selectors are used to select one or more HTML elements using jQuery. A jQuery Selector is a function which makes use of expressions to find out matching elements from a DOM based on the given criteria, such as element name, element ids, and element class attributes.
*Once element(s) are selected, we can perform various operations on the selected element(s).
*The jQuery $() factory function: jQuery selectors start with the dollar sign and parentheses: $().
*The factory function $() is a synonym of jQuery() function. So in case we're using any other JavaScript library where $ sign is conflicting with some thing else, then we can use the function jQuery() instead of $().
*The selectors are very useful and would be required at every step while using jQuery. They get the exact element that we want from an HTML document.
*The factory function $() makes use of following three building blocks while selecting elements in a document:

jQuery Selector	Description
Universal Selector: $(*)	*Selects all elements available in a DOM.
Tag Name Selector: $('tagName')	*Selects all elements which match with the given element Name. For example $('p') selects all paragraphs <p> in the document.
Tag ID Selector: $('#id')	*Selects a single element which matches with the given id attribute in the DOM. For example $('#some-id') selects the single element in the document that has an ID of some-id.
Tag Class Selector: $('.class')	*Selects all elements which match with the given class attribute in the DOM. For example $('.some-class') selects all elements in the document that have a class of some-class.
Multiple Elements E, F, G Selector:	*Selects the combined results of all the specified selectors E, F or G.

Table: jQuery HTML element selectors.
*All the above items can be used either on their own or in combination with other selectors. All the jQuery selectors are based on the same principle.
*We have different type of useful selectors: We can use all the above selectors with any HTML element in generic way. For example if selector $("li:first") works for <li> element then $("p:first") would also work for <p> element.
*Example: The following is a simple example which makes use of Tag Selector. This would select all the elements with a tag name p.

```
//File: tagNameSelector.htm
<html>
  <head>
    <title>The jQuery Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        $("p").css("background-color", "yellow");
      });
    </script>
  </head>
  <body>
    <div>
      <p class = "myclass">This is a paragraph.</p>
      <p id = "myid">This is second paragraph.</p>
      <p>This is third paragraph.</p>
    </div>
  </body>
</html>
```

*This will produce following result, with each <p> element in a yellow background color:
This is a paragraph.

This is second paragraph.
This is third paragraph.

Using jQuery to Manipulate HTML DOM Element Attributes: We can manipulate the DOM elements attributes assigned to those elements.
*Most of these attributes are available through JavaScript as DOM node properties. Some of the more common properties are: className, tagName, id, href, title, rel, and src.
*Example: Consider the following HTML markup for an image element:

In this element's markup, the tag name is img, and the markup for id, src, alt, class, and title represents the element's attributes, each of which consists of a name and a value pair.
*jQuery gives us the means to easily manipulate an element's attributes and gives us access to the element so that we can also change its attribute values.
***jQuery HTML Element Attribute Manipulation Methods**: The following lists some useful methods which we can use to manipulate attributes of HTML elements:
***jQuery Get Attribute Value Method attr()**: The attr() method can be used to either fetch the value of an attribute from the first element in the matched set or set attribute values, see below, onto all matched elements.
*Example: The following is a simple example which fetches title attribute of tag and set value with the same value:
//File: getTagAttribute.htm

```
<html>
  <head>
    <title>The jQuery Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        var title = $("em").attr("title");
        $("#divid").text(title);
      });
    </script>
  </head>
  <body>
    <div>
      <em title = "Bold and Brave">This is first paragraph.</em>
      <p id = "myid">This is second paragraph.</p>
      <div id = "divid"></div>
    </div>
  </body>
</html>
```

*This will produce following result:
This is first paragraph.
This is second paragraph.
Bold and Brave
***jQuery Set an Attribute Value Method attr(name, value)**: The attr(name, value) method can be used to set a named attribute onto all elements in the wrapped set using the passed value.
*Example:
Following is a simple example which set src attribute of an image tag to a correct location:
//File: setTagAttribute.htm

```
<html>
  <head>
    <title>The jQuery Example</title>
    <base href="https://www.tutorialspoint.com" />
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
```

```
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        $("#myimg").attr("src", "/jquery/images/jquery.jpg");
      });
    </script>
  </head>
  <body>
    <div>
      <img id = "myimg" src = "/images/jquery.jpg" alt = "Sample image" />
    </div>
  </body>
</html>
```

*This will produce following result:

***jQuery Set Multiple Attributes Values Method attr({attribute1:value1, attribute2:value2, ...})**: The method attr({attribute1:value1, attribute2:value2, ...}) can be used to set key/value object pairs as attributes to all matched elements. Parameters: (*) attribute1, attribute2: This is the attribute of the matched element. (*) value1, value2: This is the value of the attribute to be set.

*Example: The following example would change the attributes of an image tag:

```
<html>
  <head>
    <title>The Selecter Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        $("img").attr({
          src: "/images/jquery.jpg",
          title: "jQuery",
          alt: "jQuery Logo"
        });
      });
    </script>
  </head>
  <body>
    <div class = "division" id = "divid">
      <p>Following is the logo of jQuery</p>
      <img src="/jquery/images/jquery-mini-logo.jpg" title="None" alt="None" />
    </div>
  </body>
</html>
```

*This will produce following result:

***jQuery Set an Attributes Value to a Computed Value with Method attr(attribute, fn)**: The function attr(attribute, fn) sets a single attribute to a computed value, on all matched elements. Parameters: (*) attribute: the name of the attribute to set. (*) func: A function returning the value to set. This function would have one argument which is index of current element.

*Example: The following example would create border for each table:

```
<html>
  <head>
    <title>The Selecter Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function(index) {
```

```
          $("table").attr("border", function() {
            return "4px";
          })
        });
      </script>
    </head>
    <body>
      <table>
        <tr><td>This is first table</td></tr>
      </table>
      <table>
        <tr><td>This is second table</td></tr>
      </table>
      <table>
        <tr><td>This is third table</td></tr>
      </table>
    </body>
</html>
```

*This will produce following result:

*__jQuery Remove an Attributes Value with Method removeAttr(attribute)__: The removeAttr(attribute) method removes an attribute from each of the matched elements. Parameters: attribute: The name of the attribute to be removed.

*Example: The following example would remove border from each table:

```
<html>
  <head>
    <title>The Selecter Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        $("table").removeAttr("border");
      });
    </script>
  </head>
  <body>
    <table border = "2">
      <tr><td>This is first table</td></tr>
    </table>
    <table border = "3">
      <tr><td>This is second table</td></tr>
    </table>
    <table border = "4">
      <tr><td>This is third table</td></tr>
    </table>
  </body>
</html>
```

*This will produce following result:

*__jQuery Add CSS Styles with the Method addClass(classes)__: The addClass(classes) method can be used to add defined CSS style sheets onto all the matched elements. You can specify multiple classes separated by space.

*Example: The following is a simple example which sets class attribute of a para <p> tag:

```
//File: setTagStyles.htm
<html>
  <head>
    <title>The jQuery Example</title>
    <script type = "text/javascript"
```

```
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        $("em").addClass("selected");
        $("#myid").addClass("highlight");
      });
    </script>
    <style>
      .selected { color:red; }
      .highlight { background:yellow; }
    </style>
  </head>
  <body>
    <em title = "Bold and Brave">This is first paragraph.</em>
    <p id = "myid">This is second paragraph.</p>
  </body>
</html>
```

*This will produce following result:

This is first paragraph.

This is second paragraph.

***jQuery Remove CSS Styles with the Method removeClass(classes)**: The removeClass(classes) method removes all or the specified class(es) from the set of matched elements. Parameters: class: The name of CSS class.

*Example: The following example would remove class red from the first paragraph <p> element:

```
<html>
  <head>
    <title>The Selecter Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        $("p#pid1").removeClass("red");
      });
    </script>
    <style>
      .red { color:red; }
      .green { color:green; }
    </style>
  </head>
  <body>
    <p class = "red" id = "pid1">This is first paragraph.</p>
    <p class = "green" id = "pid2">This is second paragraph.</p>
  </body>
</html>
```

*This will produce following result:

This is first paragraph.

This is second paragraph.

***jQuery Toggle CSS Style with the Method toggleClass(class)**: The toggleClass(class) method adds the specified class if it is not present in the set of matched elements, or removes the specified class if it is present in the set of matched elements. Parameters: class: The name of CSS class.

*Example: The following example would remove a class with one click and in second click it would again add the same class:

```
<html>
  <head>
    <title>The Selecter Example</title>
```

```
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        $("p#pid").click(function () {
          $(this).toggleClass("red");
        });
      });
    </script>
    <style>
      .red { color:red; }
    </style>
  </head>
  <body>
    <p class = "green">Click following line to see the result</p>
    <p class = "red" id = "pid">This is first paragraph.</p>
  </body>
</html>
```

*This will produce following result:

Click following line to see the result

This is first paragraph.

***jQuery Test if a CSS Style is Present with the Method hasClass(class)**: The hasClass(class) returns true if the specified class is present on at least one of the set of matched elements, otherwise it returns false. Parameters: class: The name of CSS class.

*Example: The following example would check which selected elements has class red:

```
<html>
  <head>
    <title>The Selecter Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        $("#result1").text( $("p#pid1").hasClass("red") );
        $("#result2").text( $("p#pid2").hasClass("red") );
      });
    </script>
    <style>
      .red { color:red; }
      .green { color:green; }
    </style>
  </head>
  <body>
    <p class = "red" id = "pid1">This is first paragraph.</p>
    <p class = "green" id = "pid2">This is second paragraph.</p>
    <div id = "result1"></div>
    <div id = "result2"></div>
  </body>
</html>
```

*This will produce following result:

This is first paragraph.

This is second paragraph.

true

false

*__jQuery Get HTML Contents of First Matched Element with the Method html()__: The html() method get the html contents (innerHTML) of the first matched element. Parameters: None.
*Example: The following example would get HTML content of first paragraph and would display it in second paragraph. Please check description of html(val) method as well, described later.

```
<html>
  <head>
    <title>The Selecter Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language="javascript">
      $(document).ready(function() {
        var content = $("p").html();
        $("#pid2").html( content );
      });
    </script>
    <style>
      .red { color:red; }
      .green { color:green; }
    </style>
  </head>
  <body>
    <p class = "green" id = "pid1">This is first paragraph.</p>
    <p class = "red" id = "pid2">This is second paragraph.</p>
  </body>
</html>
```

*This will produce following result:
This is first paragraph.
This is first paragraph.
*__jQuery Set HTML Contents of Every Matched Element with the Method html(val)__: The html(val) method sets the html contents of every matched element. Parameters: val: Any string.
*Example: The following example would get HTML content of first paragraph and would display it in second paragraph. Please check description of html() method as well, described earlier.

```
<html>
  <head>
    <title>The Selecter Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        var content = $("p").html();
        $("#pid2").html( content );
      });
    </script>
    <style>
      .red { color:red; }
      .green { color:green; }
    </style>
  </head>
  <body>
    <p class = "green" id = "pid1">This is first paragraph.</p>
    <p class = "red" id = "pid2">This is second paragraph.</p>
  </body>
</html>
```

*This will produce following result;

This is first paragraph.
This is first paragraph.
*__jQuery Get HTML Text Contents of All Matched Element with the Method text()__: The text() method gets the combined concatenated text contents of all matched elements. Parameters: None.
*Example: The following example would find the text in the first paragraph stripping out the html, then set the html of the second paragraph to show it is just text.

```
<html>
  <head>
    <title>The Selecter Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language="javascript">
      $(document).ready(function() {
        var content = $("p#pid1").text();
        $("#pid2").html(content);
      });
    </script>
    <style>
      .red { color:red; }
      .green { color:green; }
    </style>
  </head>
  <body>
    <p class = "green" id = "pid1">This is <i>first paragraph</i>.</p>
    <p class = "red" id = "pid2">This is second paragraph.</p>
  </body>
</html>
```

*This will produce following result:
This is first paragraph.
This is first paragraph.
*__jQuery Set HTML Text Contents of All Matched Element with the Method text(val)__: The text(val) method sets the text contents of all matched elements. This method is similar to html(val) but escapes all HTML entities. Parameters: val: Any string.
*Example: The following example would set the HTML content of the first paragraph in the second paragram but it escapes all the HTML tag.

```
<html>
  <head>
    <title>The Selecter Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        var content = $("p#pid1").html();
        $("#pid2").text(content);
      });
    </script>
    <style>
      .red { color:red; }
      .green { color:green; }
    </style>
  </head>
  <body>
    <p class = "green" id = "pid1">This is <i>first paragraph</i>.</p>
    <p class = "red" id = "pid2">This is second paragraph.</p>
```

```
  </body>
</html>
```
*This will produce following result:
This is first paragraph.
This is first paragraph.
*__jQuery Get HTML Input Value of the First Matched <input> Element with Method val()__: The val() method gets the input value of the first matched <input> element. Parameters: None.
*Example: The following example would set the HTML content of the first input box in the second paragraph:
```
<html>
  <head>
    <title>The Selecter Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        var content = $("input").val();
        $("p#pid2").text(content);
      });
    </script>
    <style>
      .red { color:red; }
      .green { color:green; }
    </style>
  </head>
  <body>
    <input type = "text" value = "First Input Box"/>
    <input type = "text" value = "Second Input Box"/>
    <p class = "green" id = "pid1">This is <i>first paragraph</i>.</p>
    <p class = "red" id = "pid2">This is second paragraph.</p>
  </body>
</html>
```
*This will produce following result:
First Input Box Second Input Box
This is first paragraph.
First Input Box
*__jQuery Set HTML Input Value of Every Matched <input> Element with Method val(val)__: The val(val) method sets the value attribute of every matched element if it is called on <input> element. If this method is called on radio buttons, checkboxes, or <select> with passed <option> values that provides a drop-down list with the specified options, then it would checks, or selects them at the passed value. Parameters: val: If it is called on <input> element, sets the value attribute of every matched element; but if it is called on <select> with the passed <option> value then passed option would be selected; if it is called on check box or radio box then all the matching check box and radiobox would be checked.
*Example: The following example would set the value attribute of the second input with the value content of the first input:
```
<html>
  <head>
    <title>The Selecter Example</title>
    <script type = "text/javascript"
      src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
    </script>
    <script type = "text/javascript" language = "javascript">
      $(document).ready(function() {
        var content = $("input").val();
        $("input").val( content );
      });
```

```
    </script>
  </head>
  <body>
    <input type = "text" value = "First Input Box"/><br/>
    <input type = "text" value = "Second Input Box"/>
  </body>
</html>
```

*This will produce following result:
First Input Box
First Input Box

XXXXXXStart
XXXStart jQuery - DOM Traversing https://www.tutorialspoint.com/jquery/jquery-traversing.htm
XXXXXXXXEnd

jQuery 2.0 for Web Development:

XXXStart Iterate a Function over DOM Objects, p.110.

*Ref: Freeman, "Pro jQuery", Second Edition, 2013, Apress.

*Ref: jQuery API: https://api.jquery.com/

*jQuery lets you modify the contents of web pages by manipulating the model that the browser creates when it processes the HTML, a process known as DOM (Document Object Model) manipulation.

*jQuery doesn't replace the DOM; it just makes it a lot easier to work with. The HTMLElement objects are still used, and the jQuery library makes it easy to switch between jQuery objects and DOM objects. The ease with which we can move from the traditional DOM to jQuery and back is part of the elegance of jQuery and helps us maintain compatibility with non-jQuery scripts and libraries.

*It is also possible to do DOM manipulation, using either another JavaScript library or the built-in web browser API (application programming interface).

*jQuery does this better. It makes DOM manipulation easier. JavaScript is the means by which jQuery functionality is defined and applied to HTML content. jQuery is a JavaScript library that we add to our HTML documents and that is executed by a browser. We employ the features of the jQuery library by adding our own code to the document as well.

*These are the top reasons to use jQuery in projects:

*(1) jQuery is expressive, and you can do more work with less code.

*(2) jQuery methods apply to multiple elements. The DOM API approach of select-iterate-modify is gone, meaning fewer for loops to iterate through elements and fewer mistakes.

*(3) jQuery deals with implementation differences between browsers. You don't have to worry about whether Internet Explorer (IE) supports a feature in an odd way, for example; we just tell jQuery what we want, and it works around the implementation differences.

*(4) jQuery is open source. If we don't understand how something works or we don't quite get the result expected, we can read through the JavaScript code and, if needed, make changes.

*(5) One of the nice aspects of jQuery is that almost any task can be performed in several different ways, allowing you to develop a personal jQuery style.

jQuery UI and jQuery Mobile: In addition to the core jQuery library, there is also jQuery UI and jQuery Mobile, which are user interface (UI) libraries built on top of jQuery.

*jQuery UI is a general-purpose UI toolkit intended to be used on any device.

*jQuery Mobile is designed for use with touch-enabled devices such as smartphones and tablets.

Downloading the jQuery Library: The very first thing you need is the jQuery library, which is available from http://jquery.com. There is a download button right on the front page of the web site and an option to choose either the production or development release.

*We'll download jQuery 2.x. We'll be using the development versions for this book.

Setting Up the jQuery Development Environment: The following sets out the software that is required for jQuery web development, all of which can be obtained free of charge.

***HTML Editors for jQuery Development**: One of the most important tools for web development is an editor with which you can create HTML documents. HTML is just text, so you can use a very basic editor, but there are some dedicated packages available that make the development smoother and simpler, and many of them are available without charge. Good HTML edit tools are:

*(1) Komodo Edit from Active State, http://activestate.com. It is free; it is simple; it has good support for HTML, JavaScript, and jQuery; and there are versions for Windows, Mac, and Linux.

*(2) Microsoft's Visual Studio. The recent versions of Visual Studio have outstanding support for HTML. Editing and can be used without any dependency of the Microsoft web stack. Visual Studio 2012 Express is available for free, see http://www.microsoft.com/visualstudio. There are also paid versions of Visual Studio, but you don't need the extra features for jQuery development.

*(3) JsFiddle is a popular online editor that provides support for working with jQuery. It is pretty flexible and powerful. It is free to use and is available at http://jsfiddle.net.

***Web Browser for jQuery Development**: We need a web browser to view your HTML documents and test your jQuery and JavaScript code. We need to use a browser that has good developer tools.

*Google Chrome is a good choice: the developer tools are pretty good.

*Another good option is Mozilla Firefox has some excellent JavaScript tools available through the Firebug extension, which you can get athttp://getfirebug.com.

*The next best option is Internet Explorer. It can also be used to do a quick sanity check when Chrome behaves in an unexpected manner.

*__Web Servers for jQuery Development__: We'll also need a web server so that the browser has somewhere from which to load the HTML documents and related resources, such as images and JavaScript files.

*A lot of web servers are available, and most of them are open source and free of charge. It doesn't matter which web server we use.

*Many web servers can be used, including Microsoft's Internet Information Services (IIS).

*__Installing Node.js for jQuery Development__: We'll use Node.js in addition to a regular web server.

*Node.js is very popular at the moment, but I have used it for the simple reason that it is based on JavaScript, so you don't have to deal with a separate web application framework.

*Node.js can be downloaded from Node.js from http://nodejs.org. There is a precompiled binary for Windows and source code that you can build for other platforms.

*Here, we'll use version 0.10.13.

*Setting Up and Testing Node.js Installation: The simplest way to test Node.js is with a simple script.

*Save the following into a file called NodeTest.js, may be in the same directory as the Node.js binary.

```
//File: Node.js: Test script:

var http = require('http');

var url = require('url');

http.createServer(function (req, res)

{

  console.log("Request: " + req.method + " to " + req.url);

  res.writeHead(200, "OK");

  res.write("<h1>Hello</h1>Node.js is working");

  res.end();

}).listen(80);

console.log("Ready on port 80");
```

*This is a simple test script that returns a fragment of HTML when it receives an HTTP GET request.

*To test Node.js, run the binary specifying the file you just created as an argument. Type he following at the command prompt:

```
node NodeTest.js
```

*To make sure everything is working, navigate to port 80 on the machine that is running Node.js. We should see a screen that says:

Hello

Node.js is working

*Which indicates that everything is working as expected.

Importing the jQuery JavaScript Library in a HTML Document Using the <script> Element: The first thing we need to do with jQuery is add it to the HTML document we want to work with. The script element lets you include JavaScript in your code.

*The following <script> element can be used to import the main jQuery library into a HTML document:

```
<script src="jquery-2.0.2.js" type="text/javascript"></script>
```

*When you define the src attribute for the script element, you are telling the browser that you want to load the JavaScript contained in another file. In this case, this is the main jQuery library, which the browser will find in the file jquery-2.0.2.js.

*Once we've selected the jQuery library that we'll will be using (the 1.x or 2.x line), we'll have the choice of two files from jquery.com; one with a .js file extension and the other with .min.js.

*For version of the 2.x line that is current, the files are called jquery-2.0.2.js and jquery-2.0.2.min.js. The jquery-2.0.2.js file is the one generally used during the development of a web site or application. This file is around 240KB and standard JavaScript code. You can open and read the file to learn about how jQuery implements its features and use the browser debugger to figure out what is going wrong if you encounter problems in your code.

*The other file, jquery.2.0.2.min.js, is intended for use when we deploy our site or web application to users. It contains the same JavaScript code but has been minified, meaning that all of the whitespace characters have been removed, and the meaningful variable names have been replaced with single-character names to save space. The minified script is almost impossible to read for the purposes of debugging, but it is much smaller. The minification process reduces the size of the file to about 82KB and if we're serving up a lot of pages that rely on jQuery, then this difference can save us significant amounts of bandwidth.

*Example: The following example shows how to add jQuery to an HTML document. We've also moved the CSS styles to a separate style sheet called styles.css.

```
//Adding jQuery to an HTML document:

<!DOCTYPE html>

<html>

<head>

  <title>Example</title>

  <link rel="stylesheet" type="text/css" href="styles.css"/>
```

```
  <script src="jquery-2.0.2.js" type="text/javascript"></script>
</head>
<body>
  //Body omitted.
</body>
</html>
```

*Importing Different Versions of the jQuery JavaScript Library for Different Versions of Internet Explorer in a HTML Document Using Conditional Comments and the <script> Element: The above example included only version 2.0.2 of the jQuery library. This version offers the best performance but it doesn't support older versions of Internet Explorer. The good news is that we don't have to choose between performance and compatibility. There is a technique that dynamically selects between the 1.x and 2.x jQuery libraries automatically, using a feature called conditional comments that Microsoft created as a nonstandard enhancement to HTML for Internet Explorer 5.

*The following example shows how to apply the conditional comments feature to the example HTML document.

```
//Using conditional comments to dynamically select between jQuery 1.x and 2.x:
<head>
  <title>Example</title>
  <link rel="stylesheet" type="text/css" href="styles.css"/>
  <!--[if lt IE 9]>
    <script src="jquery-1.10.1.js" type="text/javascript"></script>
  <![endif]-->
  <!--[if gte IE 9]><!-->
    <script src="jquery-2.0.2.js" type="text/javascript"></script>
  <!--<![endif]-->
</head>
<body>
  //Body omitted.
</body>
</html>
```

*The conditional comments will load jQuery 1.10.1 for versions prior to Internet Explorer 9 and jQuery 2.0.2 for all other Internet Explorer 9 and later browsers. We should take care to copy these comments exactly, as it is easy to make a mistake.

*Note: For the remaining examples, we'll make explicit use of jQuery 2.0.2 so as to keep the examples clear and simple, but it is recommend that we use the conditional comments technique in our projects.

*For more information about conditional comments: http://en.wikipedia.org/wiki/Conditional_comment

*Obtaining jQuery from a Public Content Delivery Network: An alternative to storing the jQuery library on our own web servers is to use a public content delivery network (CDN) that hosts jQuery. A CDN is a distributed network of servers that deliver files to the user using the server that is closest to them. There are a couple of benefits to using a CDN. The first benefit is a faster experience to the user, because the jQuery library file is downloaded from the server closest to them, rather than from our servers. Often a download won't be required at all: jQuery is so popular that the user's browser may have already cached the library from another application or site that also uses jQuery. The second benefit is that none of our bandwidth is used to deliver jQuery to the user. For high-traffic sites, this can be a significant cost savings.

*When using a CDN, we must have confidence in the CDN operator. We want to be sure that the user receives the file they are supposed to and that service will always be available. Google and Microsoft both provide CDN services for jQuery, and other popular JavaScript libraries, free of charge. Both companies have good experience running highly available services and are unlikely to deliberately tamper with the jQuery library. You can learn about the Microsoft service at www.asp.net/ajaxlibrary/cdn.ashx and about the Google service athttp://code.google.com/apis/libraries/devguide.html.

*The CDN approach isn't suitable for applications that are delivered to users within an intranet because it causes all the browsers to go to the Internet to get the jQuery library, rather than access the local server, which is generally closer and faster and has lower bandwidth costs.

Example: A First jQuery Script: Now that we've added the jQuery library to the document, we can write some JavaScript that uses jQuery functionality.
*The following contains a simple script element that shows off some of the basic jQuery features.

```
//A first jQuery script:
<!DOCTYPE html>
<html>
<head>
  <title>Example</title>
  <link rel="stylesheet" type="text/css" href="styles.css"/>
  <script src="jquery-2.0.2.js" type="text/javascript"></script>
  <script type="text/javascript">
    $(document).ready(function ()
    {
      $("img:odd").mouseenter(function (e)
      {
        $(this).css("opacity", 0.5);
      }).mouseout(function (e)
      {
        $(this).css("opacity", 1.0);
      });
    });
  </script>
</head>
<body>
  //Body omitted.
</body>
```

</html>
*This short script demonstrates some of the most important features and characteristics of jQuery.
*This script uses the the :odd pseudo-selector, which selects the odd-numbered elements matched by the main part of the selector, img in this case, which selects all of the img elements. The :odd selector is zero-based, meaning that the first element is considered to be even. We'll talk about these selectors in detail later. Thus, this script has the effect of changing the opacity of odd-numbered image elements when the mouse is moved over them. This has the effect of making these images look a little brighter and washed out. When the mouse is moved away from the image, the opacity returns to its previous value. The images for even-numbered image elements are unaffected.

The jQuery jQuery() Function: https://api.jquery.com/jQuery/: Returns a collection of matched elements either found in the DOM based on passed argument(s) or created by passing an HTML string. jQuery function can be called in the following overloaded jQuery methods:
*(1) jQuery(selector [, context]): Accepts a string containing a CSS selector which is then used to match a set of elements. Parameters: selector, type: Selector, a string containing a CSS selector expression; context, type: element or jQuery, a DOM Element, document, or jQuery to use as context. Returns: jQuery. In this formulation, jQuery(), which can also be written as $(), searches through the DOM for any elements that match the provided selector and creates a new jQuery object that references these elements:
$("div.foo");
If no elements match the provided selector, the new jQuery object is "empty"; that is, it contains no elements and has .length property of 0.
*Selector Context: By default, selectors perform their searches within the DOM starting at the document root. However, an alternate context can be given for the search by using the optional second parameter to the $() function. For example, to do a search within an event handler, the search can be restricted like so:
```
$( "div.foo" ).click(function()
{
  $( "span", this ).addClass( "bar" );

});
```
When the search for the span selector is restricted to the context of this, only spans within the clicked element will get the additional class. Note: Internally, selector context is implemented with the .find() method, so $("span", this) is equivalent to $(this).find("span").
*(2) jQuery(element), jQuery(elementArray): Parameters: element, type: Element, a DOM element to wrap in a jQuery object; elementArray, type: Array, an array containing a set of DOM elements to wrap in a jQuery object. Returns: jQuery. Using DOM elements: These formulations of this function create a jQuery object using one or more DOM elements that were already selected in some other way. A jQuery object is created from the array elements in the order they appeared in the array; unlike most other multi-element jQuery operations, the elements are not sorted in DOM order. Elements will be copied from the array as-is and won't be unwrapped if they're already jQuery collections. Please note that although you can pass text nodes and comment nodes into a jQuery collection this way, most operations don't support them. The few that do will have an explicit note on their API documentation page.
*A common use of single-DOM-element construction is to call jQuery methods on an element that has been passed to a callback function through the keyword this:
```
$( "div.foo" ).click(function()
{
  $( this ).slideUp();
});
```
This example causes elements to be hidden with a sliding animation when clicked. Because the handler receives the clicked item in the this keyword as a bare DOM element, the element must be passed to the $() function before applying jQuery methods to it.
*XML data returned from an Ajax call can be passed to the $() function so individual elements of the XML structure can be retrieved using .find() and other DOM traversal methods.
```
$.post( "url.xml", function( data )
{
  var $child = $( data ).find( "child" );
});
```

*(3) jQuery(selection): Parameters: selection, type: jQuery, an existing jQuery object to clone. Returns: jQuery. Cloning jQuery Objects: When a jQuery object is passed to the $() function, a clone of the object is created. This new jQuery object references the same DOM elements as the initial one.
*(4) jQuery(): Parameters: This signature does not accept any arguments. Returns: jQuery. Returning an Empty Set: As of jQuery 1.4, calling the jQuery() method with no arguments returns an empty jQuery set, with a .length property of 0. In previous versions of jQuery, this would return a set containing the document node.
*(5) jQuery(object): Parameters: object, type: PlainObject, a plain object to wrap in a jQuery object. Returns: jQuery. Working With Plain Objects: At present, the only operations supported on plain JavaScript objects wrapped in jQuery are: .data(),.prop(),.on(), .off(), .trigger() and .triggerHandler(). The use of .data(), or any method requiring .data(), on a plain object will result in a new property on the object called jQuery{randomNumber}, such as jQuery123456789.

```
//Define a plain object
var foo = { foo: "bar", hello: "world" };
//Pass it to the jQuery function
var $foo = $( foo );
//Test accessing property values
var test1 = $foo.prop( "foo" ); //bar
//Test setting property values
$foo.prop( "foo", "foobar" );
var test2 = $foo.prop( "foo" ); //foobar
//Test using .data() as summarized above
$foo.data( "keyName", "someValue" );
console.log( $foo ); //will now contain a jQuery{randomNumber} property
//Test binding an event name and triggering
$foo.on( "eventName", function ()
{
  console.log( "eventName was called" );
});
$foo.trigger( "eventName" ); //Logs "eventName was called"
```

Note: Should .trigger("eventName") be used, it will search for an "eventName" property on the object and attempt to execute it after any attached jQuery handlers are executed. It does not check whether the property is a function or not. To avoid this behavior, .triggerHandler("eventName") should be used instead.

```
$foo.triggerHandler( "eventName" ); //Also logs "eventName was called"
```

*(6) jQuery(html [, ownerDocument]): Parameters: html, type: htmlString, a string of HTML to create on the fly, and is parsed as HTML, not XML; ownerDocument, type: document, a document in which the new elements will be created. Returns: jQuery. Creates DOM elements on the fly from the provided string of raw HTML. Creating New DOM Elements: If a string is passed as the parameter to $(), jQuery examines the string to see if it looks like HTML, i.e., it starts with <tag ... >. If not, the string is interpret-ted as a selector expression. But if the string appears to be an HTML snippet, jQuery attempts to create new DOM elements as described by the HTML. Then a jQuery object is created and returned that refers to these elements. For explicit parsing of a string to HTML, use the $.parseHTML() method. You can perform any of the usual jQuery methods on this object:

```
$( "<p id='test'>My <em>new</em> text</p>" ).appendTo( "body" );
```

By default, elements are created with an .ownerDocument matching the document into which the jQuery library was loaded. Elements being injected into a different document should be created using that document, e.g., $(" hello iframe ", $("#myiframe").prop("contentWindow").document).
*If the HTML is more complex than a single tag without attributes, as it is in the above example, the actual creation of the elements is handled by the browser's .innerHTML mechanism. In most cases, jQuery creates a new element and sets the innerHTML property of the element to the HTML snippet that was passed in. When the parameter has a single tag. with optional closing tag or quick-closing, such as $("<img />") or $("<img>"), $("<a>") or $("<a>"), jQuery creates the element using the native JavaScript .createElement() function.
*When passing in complex HTML, some browsers may not generate a DOM that exactly replicates the HTML source provided. As mentioned, jQuery uses the browser's .innerHTML property to parse the passed HTML and insert it into the current document. During this process, some browsers filter out certain elements such as <html>, <title>, or <head> elements. As a result, the elements inserted may not be representative of the original string passed.

*Filtering isn't, however, limited to these tags. For example, Internet Explorer prior to version 8 will also convert all href properties on links to absolute URLs, and Internet Explorer prior to version 9 will not correctly handle HTML5 elements without the addition of a separate compatibility layer.
*To ensure cross-platform compatibility, the snippet must be well-formed. Tags that can contain other elements should be paired with a closing tag:
$("");
*Tags that cannot contain elements may be quick-closed or not:
$("<img>");
$("<input>");
*When passing HTML to jQuery(), note that text nodes are not treated as DOM elements. With the exception of a few methods, such as .content(), they are generally ignored or removed. Examples:
var el = $("
2
3"); //returns [
, "2",
]
el = $("
2
3 >"); //returns [
, "2", <br>, "3 >"]
*(7) jQuery(callback): Parameters: callback, type: Function(), the function to execute when the DOM is ready. Returns: jQuery. Binds a function to be executed when the DOM has finished loading. This functi-on behaves just like $(document).ready(), in that it should be used to wrap other $() operations on your page that depend on the DOM being ready. While this function is, technically, chainable, there really isn't much use for chaining against it.
*Example: Find all p elements that are children of a div element and apply a border to them.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>jQuery demo</title>
  <script src="https://code.jquery.com/jquery-1.10.2.js"></script>
</head>
<body>
  <p>one</p>
  <div><p>two</p></div>
  <p>three</p>
  <script>
    $( "div > p" ).css( "border", "1px solid gray" );
  </script>
</body>
</html>
```

Output:
one
two //This has a border applied to it.
three
*Example: Find all inputs of type radio within the first form in the document.
$("input:radio", document.forms[0]);
*Example: Find all div elements within an XML document from an Ajax response.
$("div", xml.responseXML);
*Example: Set the background color of the page to black.
$(document.body).css("background", "black");
*Example: Hide all the input elements within a form.
$(myForm.elements).hide();
*Example: Create a div element, and all of its contents, dynamically and append it to the body element. Internally, an element is created and its innerHTML property set to the given markup.
$(" Hello ").appendTo("body")
*Example: Execute the function when the DOM is ready to be used.

```
$(function()
{
  //Document is ready
});
```

Use both the shortcut for $(document).ready() and the argument to write failsafe jQuery code using the $ alias, without relying on the global alias.

```
jQuery(function( $ )
{
  //Your code using failsafe $ alias here...
});
```

Accessing jQuery with the Main jQuery() Function, or its Shorthand $(…) Function: We access jQuery with the main jQuery function jQuery(), which is the entry point to the world of jQuery. The main jQuery() function can also be accessed through its shorthand function $(…) function, which we will refer to as the $ function for simplicity.

*We can rewrite the example script above to use the main jQuery() function name if we prefer, as below:

```
//Using the jQuery function in place of the shorthand $ function:
<script type="text/javascript">
  jQuery(document).ready(function ()
  {
    jQuery("img:odd").mouseenter(function(e)
    {
      jQuery(this).css("opacity", 0.5);
    }).mouseout(function(e)
    {
      jQuery(this).css("opacity", 1.0);
    });
  });
</script>
```

*This script provides the same functionality as the previous example. It requires slightly more typing but has the advantage of making the use of jQuery explicit. This can be useful because jQuery is not the only JavaScript library that uses the $ notation, which can cause problems when we use multiple libraries in the same document.

*For our purpose, we'll be using the $ notation throughout the examples, since it is the normal convention for jQuery, and because we won't be using any other library that wants control of the $ notation.

*We can make jQuery relinquish control of the $ by calling the jQuery.noConflict method, as below:

```
//Releasing jQuery's control of the $ function:
<script type="text/javascript">
  jQuery.noConflict();
  jQuery(document).ready(function ()
  {
    jQuery("img:odd").mouseenter(function(e)
    {
      jQuery(this).css("opacity", 0.5);
    }).mouseout(function(e)
    {
      jQuery(this).css("opacity", 1.0);
    });
  });
</script>
```

*We can also define our own shorthand function name for accessing jQuery. We do this by assigning the result of the noConflict method call to a variable, as shown below:

```
//Creating a shorthand function name for accessing jQuery:
<script type="text/javascript">
  var jq = jQuery.noConflict();
  jq(document).ready(function ()
  {
    jq("img:odd").mouseenter(function(e)
    {
      jq(this).css("opacity", 0.5);
    }).mouseout(function(e)
    {
      jq(this).css("opacity", 1.0);
```

```
    });
  });
</script>
```
*In this example, we created our own shorthand function name jq for accessing jQuery, and then used this shorthand throughout the rest of the script.
*Irrespective of how we refer to the main jQuery function, we can pass the same set of arguments, the most important of which are described in the table below. All of these arguments are described later.

Argument	Description
$(function)	Specifies a function to be executed when the DOM is ready.
$(selector)	Selects elements from the document.
$(selector, $(contextSelector))	Selects elements from the document by narrowing the selection of the elements from the document by using a context selector.
$(HTMLElement)	Creates a jQuery object from an HTMLElement object.
$(HTMLElement[])	Creates a jQuery object from an array of HTMLElement objects.
$()	Creates an empty selection.
$(HTML)	Creates new elements from a fragment of HTML.
$(HTML, map)	Creates new elements from a fragment of HTML with a map object to define attributes.

Table: Arguments to the main jQuery function.

The jQuery .ready(handler) Function: https://api.jquery.com/ready/: Specify a function to execute when the DOM is fully loaded. Parameter: handler, type: Function(), a function to execute after the DOM is ready. Returns: jQuery. The .ready() method offers a way to run JavaScript code as soon as the page's Document Object Model (DOM) becomes safe to manipulate. This will often be a good time to perform tasks that are needed before the user views or interacts with the page, for example to add event handlers and initialize plugins.
*jQuery offers several ways to attach a function that will run when the DOM is ready. All of the following syntaxes are equivalent:

```
$( handler )
$( document ).ready( handler )
$( "document" ).ready( handler )
$( "img" ).ready( handler )
$().ready( handler )
```

As of jQuery 3.0, only the first syntax is recommended; the other syntaxes still work but are deprecated.
*The .ready() method is typically used with an anonymous function:

```
$( document ).ready(function()
{
  //Handler for .ready() called.
});
```

Which is equivalent to the recommended way of calling:

```
$(function()
{
  //Handler for .ready() called.
});
```

*Aliasing the jQuery Object: When $.noConflict() is used to avoid namespace conflicts, the $ shortcut is no longer available. However, the .ready() handler is passed a reference to the jQuery object that called the method. This allows the handler to use a jQuery object, for example as $, without knowing its aliased name:

```
jq2 = jQuery.noConflict();
jq2(function( $ )
{
  //Code using $ as usual goes here; the actual jQuery object is jq2
});
```

*Example: Display a message when the DOM is loaded.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
```

```
  <title>ready demo</title>
  <style>
    p { color: red; }
  </style>
  <script src="https://code.jquery.com/jquery-1.10.2.js"></script>
  <script>
    $(function()
    {
      $( "p" ).text( "The DOM is now loaded and can be manipulated." );
    });
  </script>
</head>
<body>
  <p>Not loaded yet.</p>
</body>
</html>
```

Output:
The DOM is now loaded and can be manipulated.

Using jQuery ready() Function to Wait for the Document Object Model (DOM) to be Created: In using the DOM API, we had previously placed the script element at the end of the document so that the browser would create all of the objects in the DOM before executing my JavaScript code.
*We can neatly avoid this issue by using jQuery, by using the ready() function, which is used to define the function we want to be called when the ready event is fired, as follows:

```
<script type="text/javascript">
  $(document).ready(function ()
  {
    //Code to execute.
  });
</script>
```

*Instead of putting the JavaScript statements directly in the script element, we've passed the document object to the $ function and called the ready method, passing in a function that we want to be executed only when the browser has loaded all of the content in the HTML document.
*The following shows how we applied this technique in the example HTML document:

```
//Waiting for the DOM:
<script type="text/javascript">
  $(document).ready(function ()
  {
    $("img:odd").mouseenter(function (e)
    {
      $(this).css("opacity", 0.5);
    }).mouseout(function (e)
    {
      $(this).css("opacity", 1.0);
    });
  });
</script>
```

*Using the ready function means we can place the script element anywhere in the HTML document, as we are safe in the knowledge that jQuery will prevent the function from being executed prematurely. We prefer to put our script elements in the head element of the HTML document, but that's just a preference.
*Note: Passing a function to the ready method creates a handler for the jQuery ready event. We cover jQuery events later. For the moment, please accept that the function we pass to the ready method will be invoked when the document is loaded and the DOM is ready for use.
*Caution: The ready method takes a function as a parameter. A common error is to omit the function part of this incantation and just pass a series of JavaScript statements to the ready method. This doesn't work and the statements are executed by the browser immediately, rather than when the DOM is ready.

*Consider the following script:

```
<script type="text/javascript">
  function countImgElements()
  {
    return $("img").length;
  }
  $(document).ready(function()
  {
    console.log("Ready function invoked. IMG count: " + countImgElements());
  });
  $(document).ready( console.log("Ready statement invoked. IMG count: " + countImgElements()) );
</script>
```

*We call the ready method twice, once with a function and once just passing in a regular JavaScript statement. In both cases, we call the countImgElements function, which returns the number of img elements that are present in the DOM. When we load the document, the script is executed, and the following output is written to the console:

Ready statement invoked. IMG count: 0
Ready function invoked. IMG count: 6

*As we can see, the statement without the function is executed as the document is loaded and before the browser has discovered the img elements in the document and created the corresponding DOM objects.

*Passing User-Defined Function to jQuery $ Function to Wait for the Document Object Model (DOM) to be Created: As an alternative notation, we can pass our function as the parameter to the jQuery $ function if we prefer. This has the same effect as using the $(document).ready approach.

*The following provides a demonstration.

//Deferring the execution of a function until the DOM Is ready:

```
<script type="text/javascript">
  $(function()
  {
    $("img:odd").mouseenter(function(e)
    {
      $(this).css("opacity", 0.5);
    }).mouseout(function(e)
    {
      $(this).css("opacity", 1.0);
    })
  });
</script>
```

The jQuery.holdReady(hold) Function: https://api.jquery.com/jQuery.holdReady: Holds or releases the execution of jQuery's ready event. Parameter: hold, type: Boolean, indicates whether the ready hold is being requested or released. Returns: undefined. The $.holdReady() method allows the caller to delay jQuery's ready event. This advanced feature would typically be used by dynamic script loaders that want to load additi-onal JavaScript such as jQuery plugins before allowing the ready event to occur, even though the DOM may be ready. This method must be called early in the document, such as in the <head> immediately after the jQuery script tag. Calling this method after the ready event has already fired will have no effect. To delay the ready event, first call $.holdReady(true). When the ready event should be released to execute, call $.holdReady(false). Note that multiple holds can be put on the ready event, one for each $.holdR-eady(true) call. The ready event will not actually fire until all holds have been released with a correspo-nding number of $.holdReady(false) calls.

*Example: Delay the ready event until a custom plugin has loaded.

```
$.holdReady( true );
$.getScript( "myplugin.js", function()
{
  $.holdReady( false );
});
```

Deferring the DOM ready Event with jQuery holdReady Method: We can control when the ready event is triggered by using the holdReady method. This can be useful if we need to load external resources dynamically, an unusual technique.
*The holdReady method must be called before the ready event is triggered and can then be called again when we are ready.
*The following gives an example of using this method.

```
//Using the holdReady method:
<script type="text/javascript">
  $.holdReady(true);
  $(document).ready(function()
  {
    console.log("Ready event triggered");
    $("img:odd").mouseenter(function(e)
    {
      $(this).css("opacity", 0.5);
    }).mouseout(function(e)
    {
      $(this).css("opacity", 1.0);
    })
  });
  setTimeout(function()
  {
    console.log("Releasing hold");
    $.holdReady(false);
  }, 5000);
</script>
```

*We call the holdReady method at the start of the script element, passing in true as the argument that indicates that we want the ready event to be held. We then define the function we want to be called when the ready event is fired with the ready function.
*Finally, we use the setTimeout method to invoke a function after five seconds. This function calls the holdReady method with an argument of false, which tells jQuery to trigger the ready event. The net effect is that we delay the ready event for five seconds. About the setTimeout Function: setTimeout is a native JavaScript function, although it can be used with a library such as jQuery, which can be used to calls a function or executes a code snippet after a specified delay in milliseconds. This might be useful if, for example, we wished to display a popup after a visitor has been browsing our page for a certain amount of time, or we want a short delay before removing a hover effect from an element, in case the user moused out accidentally.
*We've added some debug messages that write the following output to the console when the document is loaded into the browser:

```
Releasing hold
Ready event triggered
```

*Note: We can call the holdReady method multiple times, but the number of calls to the holdReady method with the true argument must be balanced by the same number of calls with the false argument before the ready event will be triggered.

Creating jQuery Objects from a HTML DOM (HTMLElement) Object or an Array of HTML DOM (HTMLElement) Objects: We can create jQuery objects by passing an HTMLElement object or an array of HTMLElement objects as the argument to the $ function, such as $(document). This can be useful when dealing with JavaScript code that isn't written in jQuery or in situations where jQuery exposes the underlying DOM objects, such as event processing.
*The following is an example.

```
//Creating jQuery objects from DOM objects:
<script type="text/javascript">
  $(document).ready(function()
  {
    var elems = document.getElementsByTagName("img");
    $(elems).mouseenter(function(e)
```

```
    {
      $(this).css("opacity", 0.5);
    }).mouseout(function(e)
    {
      $(this).css("opacity", 1.0);
    })
  });
</script>
```

*In the example, we select the img elements in document using the document.getElementsByTagName method, rather than using jQuery directly with a selector, as shown in the next section.
*We pass the results of this method, which is a collection of HTMLElement objects, to the $ function, which returns a regular jQuery object that we can use in the usual way.
*This script also demonstrates how we can create a jQuery object from a single HTMLElement object:
$(this).css("opacity", 1.0);
*When we're handling events, jQuery sets the value of the this variable to the HTMLElement that is processing the event.

Using jQuery Selectors to Select Elements and Element Types from the DOM: One of the most important areas of jQuery functionality is how you select elements from the DOM using jQuery selectors.
*When we use jQuery to select elements from the DOM, the result from the $ function is a jQuery Java-Script object, which represents zero or more DOM elements. The properties and methods defined by the jQuery object can then be applied to this jQuery object that is returned by the selection, as shown later.
*In the following example script, we locate all of the odd img elements.

```
//Selecting Elements from the DOM:
<script type="text/javascript">
  $(document).ready(function()
  {
    $("img:odd").mouseenter(function(e)
    {
      $(this).css("opacity", 0.5);
    }).mouseout(function(e)
    {
      $(this).css("opacity", 1.0);
    })
  });
</script>
```

*To select elements, you simply pass a selector to the $ function. jQuery supports all of the CSS selectors, plus some additional ones that give us some handy fine-grained control.
*In the example we used the :odd pseudo-selector, which selects the odd-numbered elements matched by the main part of the selector, img in this case, which selects all of the img elements. The :odd selector is zero-based, meaning that the first element is considered to be even.
*The following table lists the most useful jQuery selectors:

Selector	Description
:animated	Selects all elements that are being animated.
:contains(text)	Selects elements that contain the specified text.
:eq(n)	Selects the element at the nth index (zero-based).
:even	Selects all the event-numbered elements (zero-based).
:first	Selects the first matched element.
:gt(n)	Selects all of the elements with an index greater than n (zero-based).
:has(selector)	Selects elements that contain at least one element that matches the specified selector.
:last	Selects the last matched element.
:lt(n)	Selects all of the elements with an index smaller than n (zero-based).
:odd	Selects all the odd-numbered elements (zero-based).
:text	Selects all text elements.

Table: jQuery extension selectors.

*We've called these the most useful because they define functionality that would be difficult to re-create using CSS selectors. These selectors are used just like the CSS pseudo-selectors. They can be used on their own, in which case they are applied to all of the elements in the DOM, like this:
$(":even")
or combined with other selectors to narrow the selection, like this:
$("img:even")
*jQuery also defines selectors that select elements based on element type, as described in table below.

Selector	Description
:button	Selects all buttons.
:checkbox	Selects all check boxes.
:file	Selects all file elements.
:header	Selects all header elements (h1, h2, and so on).
:hidden	Selects all hidden elements.
:image	Selects all image elements.
:input	Selects all input elements.
:last	Selects the last matched element.
:parent	Selects all of the elements that are parents to other elements.
:password	Selects all password elements.
:radio	Selects all radio elements.
:reset	Selects all elements that reset a form.
:selected	Selects all elements that are selected.
:submit	Selects all form submission elements.
:visible	Selects all visible elements.

Table: jQuery type extension selectors.
*Using jQuery Selectors to Select Elements and Element Types from the DOM After Narrowing the Selection with a Context Selector: By default, jQuery searches the entire DOM for elements, but you can narrow the scope of a selection by providing an additional context selector argument to the $ function, as in the selector $(selector, $(contextSelector)), which selects elements from the document by narrowing the selection of the elements from the document by using a context selector. This gives the search a context selector, which is used as the starting point for matching elements, as shown below:

```
//Narrowing the selection of the elements from the document by using a context selector.
<script type="text/javascript">
  $(document).ready(function()
  {
    $("img:odd", $(".drow")).mouseenter(function(e)
    {
      $(this).css("opacity", 0.5);
    }).mouseout(function(e)
    {
      $(this).css("opacity", 1.0);
    })
  });
</script>
```

*In this example, we use one jQuery selection as a context selector for another. The context selector is evaluated first, and it matches all of the elements that are members of the drow class. This set of elements is then used as the context for the img:odd selector.
*When we supply a context selector that contains multiple elements, then each element is used as a starting point in the search. There is an interesting subtlety in this approach. The elements that match the context selector are gathered together, and then the main selection is performed. In the example, this means the img:odd selector is applied to the results of the drow selector, which means that the odd-numbered elements are not the same as when you search the entire document. The net result is that the opacity effect is applied to the odd-numbered img elements in the drow class.

*If we just want to match elements starting at a given element in the document, then we can use an HTMLElement object as the context selector. The following example also shows how to easily switch between the jQuery world and HTMLElement objects.

```
//Using an HTMLElement as the context selector:
<script type="text/javascript">
  $(document).ready(function()
  {
    var elem = document.getElementById("oblock");
    $("img:odd", elem).mouseenter(function(e)
    {
      $(this).css("opacity", 0.5);
    }).mouseout(function(e)
    {
      $(this).css("opacity", 1.0);
    })
  });
</script>
```

*The script in this example searches for odd-numbered img elements, limiting the search to those elements that are descended from the element whose id is oblock. Of course, we could achieve the same effect using the descendant CSS selector. The benefit of this approach arises when we want to narrow a search programmatically, without having to construct a selector string. A good example of such a situation is when handling an event, which is covered later.

Applying jQuery Object Properties and Methods to Elements and Element Types Selected from the DOM by jQuery Selectors or jQuery Objects Created from HTML DOM HTMLElement Objects:

*When we use jQuery to select elements from the DOM, the result from the $ function is a jQuery JavaScript object, which represents zero or more DOM elements. Also, when we perform a jQuery operation that modifies one or more elements, the result is likely to be a jQuery object, which is an important characteristic.

*The properties and methods defined by the jQuery object can then be applied to this jQuery object that is returned by the selection, as shown below.

*The following are some of the basic jQuery object properties and methods:

Selector	Description	Returns

context	*Returns the set of elements used as the search context.
	*The jQuery context property provides us with details of the context used when the jQuery object was created.
	*If a single HTMLElement object was used as the context, then the context property will return that HTMLElement.
	*If no context was used or if multiple elements were used, then the context property returns undefined instead.
	*Example: The following shows the context property in use.
	//Determining the context for a jQuery object:
	<script type="text/javascript">
	$(document).ready(function()
	{
	var jq1 = $("img:odd");
	console.log("No context: " + jq1.context.tagName);
	var jq2 = $("img:odd", $(".drow"));
	console.log("Multiple context elements: " +
	jq2.context.tagName);
	var jq3 = $("img:odd",
	document.getElementById("oblock"));
	console.log("Single context element: " +
	jq3.context.tagName);
	});
	</script>
	*This script selects elements using no context, multiple context
		HTMLElement

	objects, and a single context object. The output is as follows: No context: undefined Multiple context elements: undefined Single context element: DIV	

each(function)	*Performs the function on each of the selected elements.
	*Example: Iterate a Function over DOM Objects: The each method lets you define a function that is performed for each DOM object in the jQuery object.
	*The following gives a demonstration:
	//Using the jQuery object each method:
	<script type="text/javascript">
	$(document).ready(function()
	{
	$("img:odd").each(function(index, elem)
	{
	console.log("Element: " + elem.tagName + " " +
	elem.src);
	});
	});
	</script>
	*jQuery passes two arguments to the specified function. The first is the index of the element in the collection, and the second is the element object itself.
	*In this example, we write the tag name and the value of the src property to the console, producing the same results as the previous script.
		jQuery

index(HTMLElement)	*Returns the index of the specified HTMLElement.	Number
index(jQuery)	*Returns the index of the first element in the jQuery object.	Number

index(selector)	*Returns the index of the first element in the jQuery object in the set of elements matched by the selector.
	*Example: Finding Indices of Specific Elements: The index method lets you find the index of an HTMLElement in a jQuery object. You can pass the index that you want using either an HTMLElement or jQuery object as the argument.
	*When you use a jQuery object, the first matched element is the one whose index is returned.
	*The following gives a demonstration.
	//Locating the index of an HTMLElement.
	<script type="text/javascript">
	$(document).ready(function()
	{
	var elems = $("body *");
	//Find an index using the basic DOM API
	var index =
	elems.index(document.getElementById("oblock"));
	console.log("Index using DOM element is: " + index);
	//Find an index using another jQuery object
	index = elems.index($("#oblock"));
	console.log("Index using jQuery object is: " + index);
	});
	</script>
	*In this example, we locate a method using the DOM API's getElementById method to find an element by using the id attribute value. This returns an HTMLElement object.
		Number

	*We then use the index method on a jQuery object to find the index of the object that represents the same element. We repeat the process using a jQuery object, which we obtain through the $ function. *We write the results from both approaches to the console, which produces the following results: Index using DOM element is: 2 Index using jQuery object is: 2	

get(index)	*Gets the HTMLElement object at the specified index.
	*Example: Finding Elements of Specific Indices: The get method is the complement to the index method, such that we specify an index and receive the HTMLElement object at that position in the jQuery object.
	*This has the same effect as using the array-style index.
	*the following provides a demonstration.
	//Getting the HTMLElement Object at a given index:
	<script type="text/javascript">
	$(document).ready(function()
	{
	var elem = $("img:odd").get(1);
	console.log("Element: " + elem.tagName + " " +
	elem.src);
	});
	</script>
	*In this script, we select the odd-numbered img elements, use the get method to retrieve the HTMLElement object at index 1, and write the value of the tagName and src properties to the console. The output from this script is as follows:
	Element: IMGhttp://www.jacquisflowershop.com/jquery/peony.png
		HTMLElement

length	*Returns the number of elements contained by the jQuery object.
	*Example: Creating jQuery Objects from an Array of HTML DOM (HTMLElement) Objects and Enumerate the Contents of the jQuery Object: We can create a jQuery object from an array of HTMLElement objects.
	*We can use the length property or the size method of the jQuery object to determine how many elements are collected in the jQuery object and access individual DOM objects by using an array-style index, using the [and] brackets.
	*Note: We can use the toArray method to extract the HTMLElement objects from the jQuery object as an array. It is easier to use the jQuery object itself, but sometimes it is useful to work with the DOM objects, such as when dealing with legacy code that wasn't written using jQuery.
	*The following shows how to enumerate the contents of a jQuery object to access the HTMLElement array objects contained within it.
	//Treating a jQuery object as an array:
	<script type="text/javascript">
	$(document).ready(function()
	{
	var elems = $("img:odd");
	for (var i = 0; i < elems.length; i++)
	{
	console.log("Element: " + elems[i].tagName + " " +
	elems[i].src);
	}
	});
	</script>
		Number

<table>
<tr><td></td><td>*In the listing, we use the $ function to select the odd-numbered img elements and enumerate the selected elements to print out the value of the tagName and src properties to the console.</td><td></td></tr>
<tr><td>size()</td><td>*Returns the number of elements in the jQuery object.</td><td>Number</td></tr>
<tr><td>toArray()</td><td>*Returns the HTMLElement objects contained by the jQuery object as an array.</td><td>HTMLElement[]</td></tr>
<tr><td>css(name),
css(names),
css(name, value),
css(map),
css(name, function)</td><td>*Using jQuery to work with CSS Properties:
*jQuery provides a set of convenience elements that make dealing with CSS easy, rather than work with the DOM methods and properties to set the value of the style attribute to define values for a CSS property for a set of elements.
*These methods operate on the style attribute of individual elements.
*The following table describes css, the most broadly useful of these methods
<table>
<tr><th>Method</th><th>Description</th></tr>
<tr><td>css(name)</td><td>*Gets the value of the specified property from the first element in the jQuery object.</td></tr>
<tr><td>css(names)</td><td>*Gets the value of multiple CSS properties, expressed as an array.</td></tr>
<tr><td>css(name, value)</td><td>*Sets the value of the specific property for all elements in the jQuery object.</td></tr>
<tr><td>css(map)</td><td>*Sets multiple properties for all of the elements in a jQuery object using a map object.</td></tr>
<tr><td>css(name, function)</td><td>*Sets values for the specified property for all of the elements in a jQuery object using a function.</td></tr>
</table>
Table: The css method.
*Getting and Setting a Single CSS Property with css Method: To read the value of a CSS property, you pass the property name to the css method. What you receive is the value from the first element in the jQuery object only. However, when you set a property, the change is applied to all of the elements.
*The following shows the basic use of the css property.
//Using the css method to get and set CSS property values
<script type="text/javascript">
$(document).ready(function()
{
var sizeVal = $("label").css("font-size");
console.log("Size: " + sizeVal);
$("label").css("font-size", "1.5em");
});
</script>
*Note that although we used the actual property name (font-size) and not the camel-case property name defined by the HTMLElement object (fontSize), jQuery happily supports both.
*In this script, we select all of the label elements and use the css method to get the value of the font-size property and write it to the console. We then select all of the label elements again and apply a new value for the same property to all of them.
*The output of the script is as follows:
Size: 16px
*Note: Setting a property to the empty string ("") has the effect of removing the property from the element's style attribute.
*Getting Multiple CSS Properties with the css Method: We can get the value of multiple CSS properties by passing an array of property names to the css method. This method returns an object that has</td><td></td></tr>
</table>

properties for each of the names in the array and the value of each property in the object is set to the value of the corresponding CSS property for the first element in the selection.
*In the following, we can see how we've used the css method to get values for three CSS properties.

```
//Using the css method to get multiple CSS property values
<script type="text/javascript">
  $(document).ready(function ()
  {
    var propertyNames = ["font-size", "color", "border"];
    var cssValues = $("label").css(propertyNames);
    for (var i = 0; i < propertyNames.length; i++)
    {
      console.log("Property: " + propertyNames[i] +
        " Value: " + cssValues[propertyNames[i]]);
    }
  });
</script>
```

*We create an array that contains the names of the three CSS properties we're interested in: font-size, color, and border. We pass this array to the css method and we receive an object that contains the values we want. That object can be expressed as follows: {font-size: "16px", color: "rgb(0, 0, 0)", border: "0px none rgb(0, 0, 0)"}
*To process the object, we iterate through the array of property names and read the corresponding property value, producing the following console output:

```
Property: font-size Value: 16px
Property: color Value: rgb(0, 0, 0)
Property: border Value: 0px none rgb(0, 0, 0)
```

*Setting Multiple CSS Properties with the css Method: We can set multiple properties in two different ways. The first is simply by chaining calls to the css method, as shown below:

```
//Chaining calls to the css method
<script type="text/javascript">
  $(document).ready(function()
  {
    $("label").css("font-size", "1.5em").css("color", "blue");
  });
</script>
```

*In this script, we set values for the font-size and color properties.
*We can achieve the same effect using a map object, as shown below. The map object follows the same pattern as the object we received when we used the css method to get multiple property values in the previous section.

```
//Setting multiple values using a map object
<script type="text/javascript">
  $(document).ready(function()
  {
    var cssVals = { "font-size": "1.5em", "color": "blue" };
    $("label").css(cssVals);
  });
</script>
```

*Both of these scripts create the effect.
*Setting Relative CSS Properties with the css Method: The css method can accept relative values, which are numeric values that are preceded by += or -= and that are added to or subtracted from the

current value. This technique can be used only with CSS properties that are expressed in numeric units, as shown below:

```
//Using relative values with the css method
<script type="text/javascript">
  $(document).ready(function()
  {
    $("label:odd").css("font-size", "+=5")
    $("label:even").css("font-size", "-=5")
  });
</script>
```

*These values are assumed to be in the same units that would be returned when the property value is read. In this case, we've increased the font size of the odd-numbered label elements by 5 pixels and decreased it for the even-numbered label elements by the same amount.

*Setting CSS Properties Using a Function with the css Method: We can set property values dynamically by passing a function to the css method, as shown below. The arguments passed to the function are the index of the element and the current value of the property. The this variable is set to the HTMLElement object for the element, and we return the value we want to set.

```
//Setting CSS Values with a Function
<script type="text/javascript">
  $(document).ready(function()
  {
    $("label").css("border", function(index, currentValue)
    {
      if ($(this).closest("#row1").length > 0)
      {
        return "thick solid red";
      }
      else if (index % 2 == 1)
      {
        return "thick double blue";
      }
    });
  });
</script>
```

*Using the CSS Property-Specific CSS Convenience Methods:
In addition to the css method, jQuery defines a number of convenience methods that can be used to get or set commonly used CSS properties and information derived from them.

*The following table describes these convenience methods.

Method	Description
height()	Gets the height in pixels for the first element in the jQuery object.
height(value)	Sets the height for all of the elements in the jQuery object.
height(function)	Sets the height for all of the elements in the jQuery object using a function.
innerHeight()	Gets the inner height of the first element in the jQuery object. This height includes the padding but excludes the border and margin.
innerWidth()	Gets the inner width of the first element in

	the jQuery object. This width includes the padding but excludes the border and margin.
offset()	Returns the coordinates of the first element in the jQuery object relative to document.
outerHeight(boolean)	Gets the height of the first element in the jQuery object, including padding and border. The argument determines if the margin is included.
outerWidth(boolean)	Gets the width of the first element in the jQuery object, including padding and border. The argument determines whether the margin is included.
position()	Returns the coordinates of the first element in the jQuery object relative to the offset.
scrollLeft(), scrollTop()	Gets the horizontal or vertical position of the first element in the jQuery object.
scrollLeft(value), scrollTop(value)	Sets the horizontal or vertical position of all the elements in a jQuery object.
width()	Gets the width of the first element in a jQuery object.
width(value)	Sets the width of all of the elements in a jQuery object.
width(function)	Sets the width for all of the elements in the jQuery object using a function.

Table: Methods for working with specific CSS properties

*Most of these methods are self-evident, but a couple warrant explanation. The result from the offset and position methods is an object that has top and left properties, indicating the location of the element. The following provides a demonstration using the position method.

```
//Using the position method
<script type="text/javascript">
  $(document).ready(function()
  {
    var pos = $("img").position();
    console.log("Position top: " + pos.top + " left: " + pos.left);
  });
</script>
```

*This script writes out the value of the top and left properties of the object returned by the method. The result is as follows:

Position top: 108.078125 left: 18

*Setting the Width and Height of Elements Using a Function: We can set the width and height for a set of elements dynami-cally by passing a function to the width or height method. The argume-nts to this method are the index of the element and the current property value. The this variable is set to the HTML-Element of the current element, and we return the value we want assigned.

```
//Setting the height of elements using a function
<script type="text/javascript">
  $(document).ready(function()
  {
```

	$("#row1 img").css("border", "thick solid red") .height(function(index, currentValue) { return (index + 1) * 25; }); }); </script> *In this script, we use the index value as a multiplier for the height.	

Table: Basic jQuery object properties and methods.

*Modifying Multiple Elements and Chaining Method Calls to jQuery Object Holding Elements and Element Types Selected from the DOM by jQuery Selectors or jQuery Objects Created from HTML DOM HTMLElement Objects: One of the features that make jQuery so concise and expressive is that calling a method on a jQuery object usually modifies all of the elements that the object contains. We said usually, because some methods perform operations that don't apply to multiple elements.

*Example: The following shows how to perform an operation on multiple elements using the DOM API.

```
//Operating on multiple elements using the DOM API
<script type="text/javascript">
  $(document).ready(function()
  {
    var labelElems = document.getElementsByTagName("label");
    for (var i = 0; i < labelElems.length; i++)
    {
      labelElems[i].style.color = "blue";
    }
  });
</script>
```

*The statements in the script element select all of the label elements and set the value of the CSS color property to blue.

*The following shows how to perform the same task using jQuery. We can perform the task using a single jQuery statement, which is less effort than using the DOM API.

```
//Operating on multiple elements Using the jQuery
<script type="text/javascript">
  $(document).ready(function ()
  {
    $("label").css("color", "blue");
  });
</script>
```

*Another nice feature of the jQuery object is that it implements a fluent API. This means that whenever we call a method that modifies the contents of the object, the result of the method is another jQuery obj-ect, which allows us to perform method chaining, as shown below. In this example, we create a jQuery object using the $ function, call the css method to set a value for the color property, and then call the css method again, this time to set the font-size property.

```
//Method chaining method calls on a jQuery object
<script type="text/javascript">
  $(document).ready(function()
  {
    $("label").css("color", "blue").css("font-size", ".75em");
  });
</script>
```

*The following shows the equivalent using the basic DOM API. We can see that it doesn't require much work to achieve the same effect using jQuery, because we already have a for loop that is enumerating the selected elements.

```
//Setting multiple css properties on multiple elements using the basic DOM API.
<script type="text/javascript">
  $(document).ready(function()
  {
    var labelElems = document.getElementsByTagName("label");
```

```
    for (var i = 0; i < labelElems.length; i++)
    {
      labelElems[i].style.color = "blue";
      labelElems[i].style.fontSize = ".75em";
    }
  });
</script>
```
*We get the real benefit from the fluent API when chaining methods that make more substantial changes to the set of elements contained in the jQuery object. The following provides a demonstration.
```
//A More Sophisticated Chaining Example
<script type="text/javascript">
  $(document).ready(function()
  {
    $("label").css("color", "blue").add("input[name!='rose']").filter("[for!='snowdrop']")
      .css("font-size", ".75em");
  });
</script>
```
*This is an over-the-top example, but it demonstrates the flexibility that jQuery offers. Let's break down the chained methods to make sense of what is happening. We start with this:
```
$("label").css("color", "blue")
```
We've selected all of the label elements in the document and set the value of the CSS color property to be blue for all of them.
*The next step is as follows:
```
$("label").css("color", "blue").add("input[name!='rose']")
```
The add method adds the elements that match the specified selector to the jQuery object. In this case, we've selected the input elements that don't have a name attribute whose value is rose. These are combined with the previously matched elements to give me a jQuery objects that contains a mix of label and input elements.
*Here is the next addition:
```
$("label").css("color", "blue").add("input[name!='rose']").filter("[for!='snowdrop']")
```
The filter method removes all of the elements in a jQuery object that don't meet a specified condition, thus allows us to remove any element from the jQuery object that has a for attribute whose value is snowdrop.
*The final step is to call the css method again, this time setting the font-size property to .75em.
```
$("label").css("color", "blue").add("input[name!='rose']").filter("[for!='snowdrop']")
  .css("font-size", ".75em");
```
*The net result of this is as follows:
1. All label elements are assigned the value blue for the color CSS property.
2. All label elements except the one that has the for attribute value of snowdrop are assigned the value .75em for the CSS font-size property.
3. All input elements that don't have a name attribute value of rose are assigned the value of .75em for the CSS font-size property.
*Achieving the same effect using the basic DOM API is a lot more complex, and is shown below. It demonstrates that jQuery provides a level of fluidity and expressiveness that is impossible to achieve using the basic DOM API.
```
//Setting multiple css properties on multiple elements using the basic DOM API.
<script type="text/javascript">
  $(document).ready(function()
  {
    var elems = document.getElementsByTagName("label");
    for (var i = 0; i < elems.length; i++)
    {
      elems[i].style.color = "blue";
      if (elems[i].getAttribute("for") != "snowdrop")
      {
        elems[i].style.fontSize= ".75em";
      }
    }
    elems = document.getElementsByTagName("input");
```

```
    for (var i = 0; i < elems.length; i++)
    {
      if (elems[i].getAttribute("name") != "rose")
      {
        elems[i].style.fontSize= ".75em";
      }
    }
  });
</script>
```

XXXXXXXStart

XXXXXXXEnd

AngularJS JavaScript Framework for Web Development:

*Ref: https://www.tutorialspoint.com/angularjs/index.htm, completed all tutorials.

*AngularJS is an open source JavaScript web application development framework. It was developed by Google. It extends HTML DOM with additional attributes and makes it more responsive to user actions.

*AngularJS is a powerful JavaScript based development framework to create RICH Internet Application(RIA). It is a framework to build large scale and high performance web application while keeping them as easy-to-maintain.

*AngularJS lets you write client side application, using JavaScript, in a clean MVC(Model View Controller) way.

*ngularJS is primarily used to make single page, data driven web applications. What this means is additional content is brought into the page dynamically without the need to refresh the page.

*Definition of AngularJS as put by its official documentation is as follows: AngularJS is a structural framework for dynamic web apps. It lets you use HTML as your template language and lets you extend HTML's syntax to express your application's components clearly and succinctly. Angular's two-way data binding and dependency injection eliminate much of the code you currently have to write. And it all happens within the browser, making it an ideal partner with any server technology.

*Application written in AngularJS is cross-browser compliant. AngularJS automatically handles JavaScript code suitable for each browser.

*AngularJS is open source, completely free, and used by thousands of developers around the world. It is licensed under the Apache License version 2.0.

*This tutorial describes the components of AngularJS with suitable examples.

*Core Features of AngularJS: Following are the most important features of AngularJS:

*(1) Data-binding: is the automatic synchronization of data between model and view components. AngularJS provides two-way data binding capability to HTML thus giving user a rich and responsive experience

*(2) Scope: are the objects that refer to the model. They act as a glue between controller and view.

*(3) Controller: are the JavaScript functions that are bound to a particular scope. In AngularJS, views are pure html pages, and controllers written in JavaScript do the business processing.

*(4) Services: AngularJS come with several built-in services for example $https: to make a XMLHttpRequests. These are singleton objects which are instantiated only once in app.

*(5) Filters: select a subset of items from an array in the model, and returns a new array.

*(6) Directives: Directives are markers on DOM elements (such as elements, attributes, css, and more). These can be used to create custom HTML tags that serve as new, custom widgets. AngularJS has built-in directives (ngBind, ngModel, etc.)

*(7) Templates: are the rendered view with information from the controller and model. These can be a single file (like index.html) or multiple views in one page using "partials".

*(8) Routing: a concept of switching views.

*(9) Model View Whatever: MVC is a design pattern for dividing an application into different parts (called Model, View and Controller), each with distinct responsibilities. AngularJS does not implement MVC in the traditional sense, but rather something closer to MVVM (Model-View-ViewModel). The Angular JS team refers it humorously as Model View Whatever (MVW).

*(10) Deep Linking: allows you to encode the state of application in the URL so that it can be bookmarked. The application can then be restored from the URL to the same state.

*(11) Dependency Injection: AngularJS has a built-in dependency injection system that helps the developer by making the application easier to develop, understand, and test. AngularJS uses dependency injection and make use of separation of concerns.

*Advantages of AngularJS:

*AngularJS code is unit testable.

*AngularJS provides reusable components.

*AngularJS applications can run on all major browsers and smart phones including Android and iOS based phones/tablets.

*Disadvantages of AngularJS:

*Not Secure: Being JavaScript only framework, application written in AngularJS are not safe. Server side authentication and authorization is must to keep an application secure.

*Not degradable: If your application user disables JavaScript then user will just see the basic page and nothing more.

***Downloading AngularJS**: AngularJS can be downloaded from https://angularjs.org/. Here, there are two options to download AngularJS library:

*View on GitHub: Click on this button to go to GitHub at https://github.com/angular/angular.js, and get all of the latest scripts.

*Download AngularJS: Click on this button. This screen gives various options of downloading Angular JS, such as:

*There are two different options legacy and latest. The names itself are self descriptive. legacy has version less than 1.2.x and latest has 1.5.x version.

*Downloading and hosting files locally.

*We can also go with the minified, uncompressed or zipped version.

*CDN access: We also have access to a CDN. The CDN will give you access around the world to regional data centers that in this case, Google host. This means using CDN moves the responsibility of hosting files from your own servers to a series of external ones. This also offers an advantage that if the visitor to your webpage has already downloaded a copy of AngularJS from the same CDN, it won't have to be re-downloaded.

***The AngularJS Components**: An AngularJS application consists of following three important parts:

*ng-app: This directive defines and links an AngularJS application to HTML.

*ng-model: This directive binds the values of AngularJS application model data to HTML input controls.

*ng-bind: This directive binds the AngularJS application model data to HTML tags.

***AngularJS: the First Application**: Before we start with creating actual HelloWorld application using AngularJS, let us see what are the actual parts of a AngularJS application.

*Example 1: Now let us write a simple HTML file that uses the AngularJS library.

//File: testAngularJS.htm

```
<!doctype html>

<html ng-app>


  <head>

    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.5.2/angular.min.js"></script>

  </head>

  <body>

    <div>
```

```
      <label>Name:</label>
      <input type = "text" ng-model = "yourName" placeholder = "Enter a name here">
      <hr />
      <h1>Hello {{yourName}}!</h1>
    </div>
  </body>
</html>
```

*Example 2: Now let us write another simple HTML file that uses the AngularJS library.

```
//File: myfirstexample.htm
<!doctype html>
<html>
  <head>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.5.2/angular.min.js"></script>
  </head>
  <body ng-app = "myapp">
    <div ng-controller = "HelloController" >
      <h2>Welcome {{helloTo.title}} to the world of Tutorialspoint!</h2>
    </div>
    <script>
      angular.module("myapp", []) .controller("HelloController", function($scope) {
        $scope.helloTo = {};
        $scope.helloTo.title = "AngularJS";
      });
    </script>
  </body>
```

</html>

*The following sections describe the steps to create AngularJS Application:

*Step 1: Load the AngularJS framework by includeing AngularJS JavaScript file with the HTML <script> tag: We need to include the AngularJS JavaScript file in the HTML so as to use AngularJS. AngularJS being a pure Java-Script framework, It can be added using <Script> tag.

```
<head>

   <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.5.2/angular.min.js"></script>

</head>
```

or choose a latest version of Angular JS. Check the latest version of AngularJS on their official website.

*Step 2: Define AngularJS Application using ng-app directive: Declare what elements of the HTML file are processed by AngularJS by adding the ng-app directive as an attribute to the HTML element: The ng-app directive indicates the start of AngularJS application. We tell what part of the HTML contains the AngularJS app, by adding the ng-app directive as an attribute to an HTML element, which makes that element and all of its nested children elements as AngularJS apps. It defines the root element of the AngularJS application. Thus, applying the ng-app attribute to the root HTML element makes the entire document an AngularJS app. It automatically initializes or bootstraps the application when web page containing AngularJS Application is loaded. It is also used to load various AngularJS modules in AngularJS Application.

We can add the ng-app attribute to any HTML element. such as the body element as shown below, which creates an angular module named myapp:

```
<body ng-app = "myapp">

   ...

</body>
```

Or the ag-app attribute can be applied any HTML element to make it an angularJS app. In following example, we've defined a default AngularJS application using ng-app attribute of a div element.

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

   ...

</div>
```

Closing tag indicates the end of AngularJS application.

*Step 3: Define a model name using ng-model directive on an HTML input control: The ng-model directive binds the values of AngularJS application model data to HTML input controls.

```
<p>Enter your Name: <input type = "text" ng-model = "name"></p>
```

The ng-model directive then creates a model variable named "name" which can be used with the html page and within the div having ng-app directive.

*Step 4: Bind the value of above model defined using ng-bind directive to a HTML tag: The ng-bind directive binds the AngularJS application model data to HTML tags.

```
<p>Hello <span ng-bind = "name"></span>!</p>
```

The ng-bind then uses the name model to be displayed in the html span tag whenever user input something in the text box. As explained later, an alternative is to use an AngularJS expression written inside double braces like {{ expression}}. Expressions are used to bind application model data to html. Expressions behaves in same way as ng-bind directives in that they bind to the model data. AngularJS application expressions are pure javascript expressions and outputs the data where they are used.

```
<p>Hello {{name}}!</p>
```

*Step 5: Specifing the controller that populates the model to use in a HTML element view by adding the ng-controller directive as an attribute to a HTML element: The following is a element that is used as a view:

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

   <h2>Welcome {{helloTo.title}} to the world of Tutorialspoint!</h2>

</div>
```

The ng-controller tells AngularJS what controller to use with this view. helloTo.title tells AngularJS to write the "model" value named helloTo.title to the HTML at this location.

*Step 6: Adding and registering a controller JavaScript function in a HTML <script> tag: The controller part is:

```
<script>

   angular.module("myapp", []).controller("HelloController", function($scope) {

      $scope.helloTo = {};

      $scope.helloTo.title = "AngularJS";

   });

</script>
```

This code registers a controller function named HelloController in the angular module named myapp. The controller function is registered in angular via the angular.module(...).controller(...) function call. A controller is a JavaScript object containing attributes/properties and functions. Each controller accepts $scope as a parameter which refers to the application/module that controller is to control. The controller function adds a helloTo JavaScript object to the model object, and in that object it adds a title field.

*Step 7: Executing an angularJS application in a browser: Save the above code as myfirstexample.html and open it in any browser. We'll see an output as below:

Welcome AngularJS to the world of Tutorialspoint!

When the page is loaded in the browser, following happens:

*(1) The HTML document is loaded into the browser, and evaluated by the browser.

*(2) AngularJS JavaScript file is loaded, the angular global object is created.

*(3) Next, JavaScript which registers controller functions is executed.

*(4) Next AngularJS scans through the HTML to look for AngularJS apps and views. Once view is located, it connects that view to the corresponding controller function.

*(5) Next, AngularJS executes the controller functions. It then renders the views with data from the model populated by the controller. The page is now ready.

AngularJS MVC Architecture: Model View Controller or MVC as it is popularly called, is a software design pattern for developing web applications. A Model View Controller pattern is made up of the following three parts:

*(1) Model: It is the lowest level of the pattern responsible for maintaining data.

*(2) View: It is responsible for displaying all or a portion of the data to the user.

*(3) Controller: It is a software code that controls the interactions between the Model and View.

*MVC is popular because it isolates the application logic from the user interface layer. The controller receives all requests for the application and then works with the model to prepare any data needed by the view. The view then uses the data prepared by the controller to generate a final presentable response.

*The MVC abstraction can be graphically represented as follows:

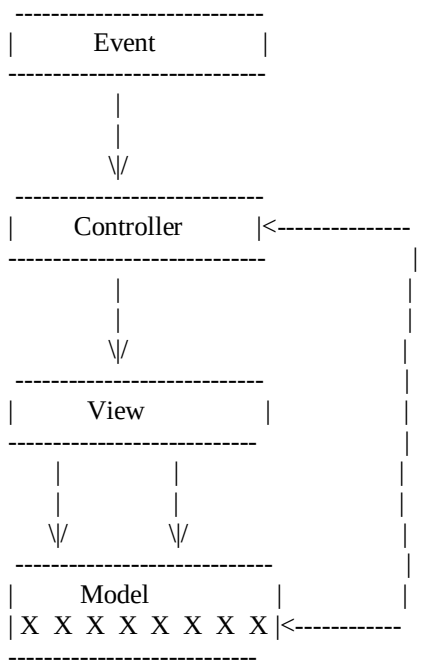

Fig: The MVC architecture

* AngularJS is a MVC based framework for web development.

*The Model: The model is responsible for managing application data. It responds to the request from view and to the instructions from controller to update itself.

*The View: A presentation of data in a particular format, triggered by the controller's decision to present the data. They are script-based template systems such as JSP, ASP, PHP and very easy to integrate with AJAX technology.

*The Controller: The controller responds to user input and performs interactions on the data model objects. The controller receives input, validates it, and then performs business operations that modify the state of the data model.

AngularJS Directives: AngularJS directives are used to extend HTML. These are special attributes starting with ng- prefix. We're going to discuss following directives:

*(1) ng-app: This directive starts an AngularJS Application. This was discussed above and is skipped.

*(2) ng-model: This directive binds the values of AngularJS application data to HTML input controls. This was discussed above and is skipped.

*(3) ng-init: This directive initializes application data.

*(4) ng-repeat: This directive repeats html elements for each item in a collection.

*ng-init directive: ng-init directive initializes an AngularJS Application model data. It is used to put values to the variables to be used in the application. In following example, we'll initialize an array of countries. We're using JSON syntax to define array of countries:

```
<div ng-app = "" ng-init = "countries = [{locale:'en-US',name:'United States'}, {locale:'en-GB',name:'United Kingdom'}, {locale:'en-DE',name:'Germany'}]">

   ...

</div>
```

*ng-repeat directive: ng-repeat directive repeats html elements for each item in a collection. In following example, we've iterated over array of countries.

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

   ...

   <p>List of Countries with locale:</p>



   <ol>

      <li ng-repeat = "country in countries">
```

```
        {{ 'Country: ' + country.name + ', Locale: ' + country.locale }}
      </li>
    </ol>
</div>
```

*Example: The following example will showcase all the above mentioned directives:

```
//File: textAngularJS.htm
<html>
   <head>
      <title>AngularJS Directives</title>
      <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
   </head>
   <body>
      <h1>Sample Application</h1>
      <div ng-app = "" ng-init = "countries = [{locale:'en-US',name:'United States'}, {locale:'en-GB',name:'United
Kingdom'}, {locale:'en-FR',name:'France'}]">
         <p>Enter your Name: <input type = "text" ng-model = "name"></p>
         <p>Hello <span ng-bind = "name"></span>!</p>
         <p>List of Countries with locale:</p>
         <ol>
            <li ng-repeat = "country in countries">
               {{ 'Country: ' + country.name + ', Locale: ' + country.locale }}
            </li>
         </ol>
      </div>
   </body>
</html>
```

*Output: Open textAngularJS.htm in a web browser. Enter your name and see the result.

Sample Application

Enter your Name:

Hello !

List of Countries with locale:

Country: United States, Locale: en-US

Country: United Kingdom, Locale: en-GB

Country: Germany, Locale: en-DE

AngularJS Expressions: written in double braces {{...}}: Expressions are used to bind application model data to html. Expressions are written inside double braces like {{ expression}}. Expressions behaves in same way as ng-bind directives in that they bind to the model data. AngularJS application expressions are pure javascript expressions and outputs the data where they are used.

*Expressions using numbers and JavaScript operators:

```
<p>Expense on Books : {{cost * quantity}} Rs</p>
```

*Expressions using JavaScript strings:

```
<p>Hello {{student.firstname + " " + student.lastname}}!</p>
```

*Expressions using JavaScript objects:

```
<p>Roll No: {{student.rollno}}</p>
```

*Expressions using JavaScript arrays:

```
<p>Marks(Math): {{marks[3]}}</p>
```

*Example: Following example uses all the above mentioned expressions:

```
//File: testAngularJSExpressions.htm
<html>
  <head>
    <title>AngularJS Expressions</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
```

```
  </head>

  <body>

    <h1>Sample Application</h1>

    <div ng-app = "" ng-init = "quantity = 1;cost = 30; student = {firstname:'Thomas', lastname:'Albert',
rollno:101};marks = [80,90,75,73,60]">

      <p>Hello {{student.firstname + " " + student.lastname}}!</p>

      <p>Expense on Books : {{cost * quantity}} US$</p>

      <p>Roll No: {{student.rollno}}</p>

      <p>Marks(Math): {{marks[3]}}</p>

    </div>

  </body>

</html>
```

*Output: Open textAngularJSExpressions.htm in a web browser. See the result.

Sample Application

Hello Thomas Albert!

Expense on Books : 30 US$

Roll No: 101

Marks(Math): 73

AngularJS Controllers: AngularJS application mainly relies on controllers to control the flow of data from the model data to the view in the application. A controller is defined using ng-controller directive. A controller is a JavaScript object containing attributes/properties and functions. Each controller accepts $scope as a parameter which refers to the application/module that controller is to control.

*Step 1: Specifing the controller that populates the model to use in a HTML element view by adding the ng-controller directive as an attribute to a HTML element. Declare a controller studentController using ng-controller directive into a element that is used as a view:

```
<div ng-app = "" ng-controller = "studentController">

  ...

</div>
```

The ng-controller tells AngularJS what controller to use with this view.

*Step 2: Adding and registering a controller JavaScript function in a HTML <script> tag. As a next step we'll define the studentController as follows:

```
<script>

  angular.module("myapp", []).controller("studentController ", function($scope) {

    $scope.student = {

      firstName: "Thomas",

      lastName: "Albert",

      fullName: function() {

        var studentObject;

        studentObject = $scope.student;

        return studentObject.firstName + " " + studentObject.lastName;

      }

    };

  });

</script>
```

This code registers a controller function named studentController in the angular module named myapp. The controller function is registered in angular via the angular.module(...).controller(...) function call. A controller is a JavaScript object containing attributes/properties and functions. Each controller accepts $scope as a parameter which refers to the application/module that controller is to control. We can also define the controller object in separate JS file and refer that file in the html page.

*studentController defined as a JavaScript object with $scope as argument.

*$scope refers to application which is to use the studentController object.

*$scope.student is property of studentController object.

*firstName and lastName are two properties of $scope.student object. We've passed the default values to them.

*fullName is the function of $scope.student object which acceses the student object and returns combined name.

*Now we can use studentController's student property using ng-model or using expressions as follows.

We've bounded student.firstName and student.lastname to two input boxes.

Enter first name: <input type = "text" ng-model = "student.firstName">

Enter last name: <input type = "text" ng-model = "student.lastName">

We've bounded student.fullName() to HTML.

You are entering: {{student.fullName()}}

*Now whenever we type anything in first name and last name input boxes, we can see the full name getting updated automatically.

*Example: the following example will showcase use of controller.

//File: testAngularJSControllers.htm

```
<html>
  <head>
    <title>Angular JS Controller</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
    <script>
      var mainApp = angular.module("mainApp", []);
      mainApp.controller('studentController', function($scope) {
        $scope.student = {
          firstName: "Thomas",
          lastName: "Albert",
          fullName: function() {
            var studentObject;
            studentObject = $scope.student;
            return studentObject.firstName + " " + studentObject.lastName;
          }
        };
      });
```

```
    </script>
  </head>
  <body>
    <h2>AngularJS Sample Application</h2>
    <div ng-app = "mainApp" ng-controller = "studentController">
      Enter first name: <input type = "text" ng-model = "student.firstName"><br><br>
      Enter last name: <input type = "text" ng-model = "student.lastName"><br>
      <br>
      You are entering: {{student.fullName()}}
    </div>
  </body>
</html>
```

*Output: Open textAngularJSController.htm in a web browser. See the result.

AngularJS Filters: Filters are used to change/modify the data and can be clubbed in expression or directives using pipe character. Following is the list of commonly used filters.

*uppercase: converts a text to upper case text.

*lowercase: converts a text to lower case text.

*currency: formats text in a currency format.

*filter: filter the array to a subset of it based on provided criteria.

*orderby: orders the array based on provided criteria.

*uppercase Filter: Add uppercase filter to an expression using pipe character. Here we've added uppercase filter to print student name in all capital letters.

```
Enter first name:<input type = "text" ng-model = "student.firstName">

Enter last name: <input type = "text" ng-model = "student.lastName">

Name in Upper Case: {{student.fullName() | uppercase}}
```

*lowercase Filter: Add lowercase filter to an expression using pipe character. Here we've added lowercase filter to print student name in all lowercase letters.

```
Enter first name:<input type = "text" ng-model = "student.firstName">

Enter last name: <input type = "text" ng-model = "student.lastName">

Name in Lower Case: {{student.fullName() | lowercase}}
```

*currency filter: Add currency filter to an expression returning number using pipe character. Here we've added currency filter to print fees using currency format.

```
Enter fees: <input type = "text" ng-model = "student.fees">

fees: {{student.fees | currency}}
```

*filter Filter: To display only required subjects, we've used subjectName as filter.

```
Enter subject: <input type = "text" ng-model = "subjectName">

Subject:

<ul>

  <li ng-repeat = "subject in student.subjects | filter: subjectName">

    {{ subject.name + ', marks:' + subject.marks }}

  </li>

</ul>
```

*orderby Filter: To order subjects by marks, we've used orderBy marks.

```
Subject:

<ul>

  <li ng-repeat = "subject in student.subjects | orderBy:'marks'">

    {{ subject.name + ', marks:' + subject.marks }}

  </li>

</ul>
```

*Example: The following example will showcase all the above mentioned filters:

```
//File: testAngularJSFilters.htm

<html>
```

```
<head>
  <title>Angular JS Filters</title>
  <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
  <script>
    var mainApp = angular.module("mainApp", []);
    mainApp.controller('studentController', function($scope) {
      $scope.student = {
        firstName: "Thomas",
        lastName: "Albert",
        fees:500,
        subjects:[
          {name:'Physics',marks:70},
          {name:'Chemistry',marks:80},
          {name:'Math',marks:65}
        ],
        fullName: function() {
          var studentObject;
          studentObject = $scope.student;
          return studentObject.firstName + " " + studentObject.lastName;
        }
      };
    });
  </script>
</head>
<body>
```

```
<h2>AngularJS Sample Application</h2>

<div ng-app = "mainApp" ng-controller = "studentController">
  <table border = "0">
    <tr>
      <td>Enter first name:</td>
      <td><input type = "text" ng-model = "student.firstName"></td>
    </tr>
    <tr>
      <td>Enter last name: </td>
      <td><input type = "text" ng-model = "student.lastName"></td>
    </tr>
    <tr>
      <td>Enter fees: </td>
      <td><input type = "text" ng-model = "student.fees"></td>
    </tr>
    <tr>
      <td>Enter subject: </td>
      <td><input type = "text" ng-model = "subjectName"></td>
    </tr>
  </table>
  <br/>
  <table border = "0">
    <tr>
      <td>Name in Upper Case: </td><td>{{student.fullName() | uppercase}}</td>
    </tr>
```

```
        <tr>
          <td>Name in Lower Case: </td><td>{{student.fullName() | lowercase}}</td>
        </tr>
        <tr>
          <td>fees: </td><td>{{student.fees | currency}}
          </td>
        </tr>
        <tr>
          <td>Subject:</td>
          <td>
            <ul>
              <li ng-repeat = "subject in student.subjects | filter: subjectName |orderBy:'marks'">
                {{ subject.name + ', marks:' + subject.marks }}
              </li>
            </ul>
          </td>
        </tr>
      </table>
    </div>
  </body>
</html>
```

*Output: Open textAngularJS.htm in a web browser. See the result.

AngularJS Tables using the ng-repeat Directive: Table data is normally repeatable by nature. ng-repeat directive can be used to draw table easily. Following example states the use of ng-repeat directive to draw a table.

```
<table>
```

```
   <tr>
      <th>Name</th>
      <th>Marks</th>
   </tr>
   <tr ng-repeat = "subject in student.subjects">
      <td>{{ subject.name }}</td>
      <td>{{ subject.marks }}</td>
   </tr>
</table>
```

Table can be styled using CSS Styling.

```
<style>
   table, th , td {
      border: 1px solid grey;
      border-collapse: collapse;
      padding: 5px;
   }
   table tr:nth-child(odd) {
      background-color: #f2f2f2;
   }
   table tr:nth-child(even) {
      background-color: #ffffff;
   }
</style>
```

*Example: The following example will showcase all the above mentioned ng-repeat directive.

```
//File: testAngularJSTables.htm
```

```
<html>
  <head>
    <title>Angular JS Table</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
    <script>
      var mainApp = angular.module("mainApp", []);
      mainApp.controller('studentController', function($scope) {
        $scope.student = {
          firstName: "Thomas",
          lastName: "Albert",
          fees:500,
          subjects:[
            {name:'Physics',marks:70},
            {name:'Chemistry',marks:80},
            {name:'Math',marks:65},
            {name:'English',marks:75},
            {name:'German',marks:67}
          ],
          fullName: function() {
            var studentObject;
            studentObject = $scope.student;
            return studentObject.firstName + " " + studentObject.lastName;
          }
        };
      });
```

```
  </script>
  <style>
    table, th , td {
      border: 1px solid grey;
      border-collapse: collapse;
      padding: 5px;
    }
    table tr:nth-child(odd) {
      background-color: #f2f2f2;
    }
    table tr:nth-child(even) {
      background-color: #ffffff;
    }
  </style>
</head>
<body>
  <h2>AngularJS Sample Application</h2>
  <div ng-app = "mainApp" ng-controller = "studentController">
    <table border = "0">
      <tr>
        <td>Enter first name:</td>
        <td><input type = "text" ng-model = "student.firstName"></td>
      </tr>
      <tr>
        <td>Enter last name: </td>
```

```
        <td>
          <input type = "text" ng-model = "student.lastName">
        </td>
      </tr>
      <tr>
        <td>Name: </td>
        <td>{{student.fullName()}}</td>
      </tr>
      <tr>
        <td>Subject:</td>
        <td>
          <table>
            <tr>
              <th>Name</th>.
              <th>Marks</th>
            </tr>
            <tr ng-repeat = "subject in student.subjects">
              <td>{{ subject.name }}</td>
              <td>{{ subject.marks }}</td>
            </tr>
          </table>
        </td>
      </tr>
    </table>
  </div>
```

</body>

</html>

*Output: Open textAngularJSTables.htm in a web browser. See the result.

AngularJS Directives for Binding to HTML DOM Elements and their Events: The following directives can be used to bind application data to attributes of HTML DOM Elements and their events:

*ng-disabled: disables a given control.

*ng-show: shows a given control.

*ng-hide: hides a given control.

*ng-click: represents a AngularJS click event.

*ng-disabled directive: Add ng-disabled attribute to a HTML button and pass it a model. Bind the model to an checkbox and see the variation.

<input type = "checkbox" ng-model = "enableDisableButton">Disable Button

<button ng-disabled = "enableDisableButton">Click Me!</button>

*ng-show directive: Add ng-show attribute to a HTML button and pass it a model. Bind the model to an checkbox and see the variation.

<input type = "checkbox" ng-model = "showHide1">Show Button

<button ng-show = "showHide1">Click Me!</button>

*ng-hide directive: Add ng-hide attribute to a HTML button and pass it a model. Bind the model to an checkbox and see the variation.

<input type = "checkbox" ng-model = "showHide2">Hide Button

<button ng-hide = "showHide2">Click Me!</button>

*ng-click directive: Add ng-click attribute to a HTML button and update a model. Bind the model to html and see the variation.

Total click: {{ clickCounter }}

<button ng-click = "clickCounter = clickCounter + 1">Click Me!</button>

*Example: The following example will showcase all the above mentioned directives.

//File: testAngularJSDirectivesForHtmlDOM.htm

```
<html>
  <head>
    <title>AngularJS Directives for HTML DOM</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
  </head>
  <body>
    <h2>AngularJS Sample Application</h2>
    <div ng-app = "">
      <table border = "0">
        <tr>
          <td><input type = "checkbox" ng-model = "enableDisableButton">Disable Button</td>
          <td><button ng-disabled = "enableDisableButton">Click Me!</button></td>
        </tr>
        <tr>
          <td><input type = "checkbox" ng-model = "showHide1">Show Button</td>
          <td><button ng-show = "showHide1">Click Me!</button></td>
        </tr>
        <tr>
          <td><input type = "checkbox" ng-model = "showHide2">Hide Button</td>
          <td><button ng-hide = "showHide2">Click Me!</button></td>
        </tr>
        <tr>
          <td><p>Total click: {{ clickCounter }}</p></td>
          <td><button ng-click = "clickCounter = clickCounter + 1">Click Me!</button></td>
        </tr>
```

```
        </table>
      </div>
   </body>
</html>
```

*Output: Open testAngularJSDirectivesForHtmlDOM.htm in a web browser. See the result.

AngularJS Application Modules/Files and Controller Modules/Files: AngularJS supports modular approach. Modules are used to separate logics say services, controllers, application etc. and keep the code clean. We define modules in separate js files and name them as per the module.js file. In this example we're going to create two modules.

*Application Module/File: used to initialize an application with controller(s).

*Controller Module/File: used to define the controller.

*Application Module/File:

```
//File: mainApp.js

var mainApp = angular.module("mainApp", []);
```

Here we've declared an application mainApp module using angular.module function. We've passed an empty array to it. This array generally contains dependent modules.

*Controller Module/File:

```
//File: studentController.js

mainApp.controller("studentController", function($scope) {
   $scope.student = {
      firstName: "Thomas",
      lastName: "Albert",
      fees:500,
      subjects:[
         {name:'Physics',marks:70},
         {name:'Chemistry',marks:80},
         {name:'Math',marks:65},
```

```
      {name:'English',marks:75},
      {name:'German',marks:67}
    ],
    fullName: function() {
      var studentObject;
      studentObject = $scope.student;
      return studentObject.firstName + " " + studentObject.lastName;
    }
  };
});
```

*Here we've declared a controller studentController module using mainApp.controller function.

*Use Application Modules/Files and Controller Modules/Files using the ng-app and ng-controller Directives Respectively in AngularJS Web Applications: In the following we use application module using ng-app directive and controller module using ng-controller directive. We've imported mainApp.js and studentController.js in the main html page using <script> tags:

```
<head>
  <script src = "mainApp.js"></script>
  <script src = "studentController.js"></script>
</head>
<body>
  <div ng-app = "mainApp" ng-controller = "studentController">
    ...
  </div>
</body>
```

*Example: Following example will showcase all the above mentioned modules.

```
//File: mainApp.js
var mainApp = angular.module("mainApp", []);
```

```
//File: studentController.js
mainApp.controller("studentController", function($scope) {
   $scope.student = {
      firstName: "Thomas",
      lastName: "Albert",
      fees:500,
      subjects:[
         {name:'Physics',marks:70},
         {name:'Chemistry',marks:80},
         {name:'Math',marks:65},
         {name:'English',marks:75},
         {name:'German',marks:67}
      ],
      fullName: function() {
         var studentObject;
         studentObject = $scope.student;
         return studentObject.firstName + " " + studentObject.lastName;
      }
   };
});
//File: testAngularJSImportModules.htm
<html>
   <head>
      <title>Angular JS Modules</title>
      <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
```

```
  <script src = "/angularjs/src/module/mainApp.js"></script>
  <script src = "/angularjs/src/module/studentController.js"></script>
  <style>
    table, th , td {
      border: 1px solid grey;
      border-collapse: collapse;
      padding: 5px;
    }
    table tr:nth-child(odd) {
      background-color: #f2f2f2;
    }
    table tr:nth-child(even) {
      background-color: #ffffff;
    }
  </style>
</head>
<body>
  <h2>AngularJS Sample Application</h2>
  <div ng-app = "mainApp" ng-controller = "studentController">
    <table border = "0">
      <tr>
        <td>Enter first name:</td>
        <td><input type = "text" ng-model = "student.firstName"></td>
      </tr>
      <tr>
```

```
        <td>Enter last name: </td>
        <td><input type = "text" ng-model = "student.lastName"></td>
      </tr>
      <tr>
        <td>Name: </td>
        <td>{{student.fullName()}}</td>
      </tr>
      <tr>
        <td>Subject:</td>
        <td>
          <table>
            <tr>
              <th>Name</th>
              <th>Marks</th>
            </tr>
            <tr ng-repeat = "subject in student.subjects">
              <td>{{ subject.name }}</td>
              <td>{{ subject.marks }}</td>
            </tr>
          </table>
        </td>
      </tr>
    </table>
  </div>
</body>
```

</html>

Output

Open textAngularJS.htm in a web browser. See the result.

AngularJS Forms Validation, Forms Handling Events, and Forms Handling: AngularJS enriches form filling and validation. We can use ng-click to handle AngularJS click on button and use $dirty and $invalid flags to do the validations in seemless way. Use novalidate with a form declaration to disable any browser specific validation. Forms controls makes heavy use of Angular events. Let's have a quick look on events first.

*AngularJS Handling Control Events: AngularJS provides multiple events which can be associated with the HTML controls. For example ng-click is normally associated with button. Following are supported events in Angular JS:

*ng-click

*ng-dbl-click

*ng-mousedown

*ng-mouseup

*ng-mouseenter

*ng-mouseleave

*ng-mousemove

*ng-mouseover

*ng-keydown

*ng-keyup

*ng-keypress

*ng-change

*ng-click

*Reset data of a form using ng-click directive of a button:

<input name = "firstname" type = "text" ng-model = "firstName" required>

<input name = "lastname" type = "text" ng-model = "lastName" required>

<input name = "email" type = "email" ng-model = "email" required>

<button ng-click = "reset()">Reset</button>

```
<script>
  function studentController($scope) {
    $scope.reset = function(){
      $scope.firstName = "Thomas";
      $scope.lastName = "Albert";
      $scope.email = "talbert@bankofamerica.com";
    }
    $scope.reset();
  }
</script>
```

*Validate data of a form using ng-click directive of a button: Following can be used to track errors on a form:

*$dirty: states that value has been changed.

*$invalid: states that value entered is invalid.

*$error: states the exact error.

*Example: Following example will showcase all the above mentioned directives.

//File: testAngularJSFormValidation.htm

```
<html>
  <head>
    <title>Angular JS Forms</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
    <script>
      var mainApp = angular.module("mainApp", []);
      mainApp.controller('studentController', function($scope) {
        $scope.reset = function(){
          $scope.firstName = "Thomas";
```

```
            $scope.lastName = "Albert";
            $scope.email = "talbert@bankofamerica.com";
         }
         $scope.reset();
      });
   </script>
   <style>
      table, th , td {
         border: 1px solid grey;
         border-collapse: collapse;
         padding: 5px;
      }
      table tr:nth-child(odd) {
         background-color: #f2f2f2;
      }
      table tr:nth-child(even) {
         background-color: #ffffff;
      }
   </style>
</head>
<body>
   <h2>AngularJS Sample Application</h2>
   <div ng-app = "mainApp" ng-controller = "studentController">
      <form name = "studentForm" novalidate>
         <table border = "0">
```

```
<tr>
  <td>Enter first name:</td>
  <td><input name = "firstname" type = "text" ng-model = "firstName" required>
    <span style = "color:red" ng-show = "studentForm.firstname.$dirty &&
studentForm.firstname.$invalid">
      <span ng-show = "studentForm.firstname.$error.required">First Name is required.</span>
    </span>
  </td>
</tr>
<tr>
  <td>Enter last name: </td>
  <td><input name = "lastname"  type = "text" ng-model = "lastName" required>
    <span style = "color:red" ng-show = "studentForm.lastname.$dirty &&
studentForm.lastname.$invalid">
      <span ng-show = "studentForm.lastname.$error.required">Last Name is required.</span>
    </span>
  </td>
</tr>
<tr>
  <td>Email: </td><td><input name = "email" type = "email" ng-model = "email" length = "100"
required>
    <span style = "color:red" ng-show = "studentForm.email.$dirty && studentForm.email.$invalid">
      <span ng-show = "studentForm.email.$error.required">Email is required.</span>
      <span ng-show = "studentForm.email.$error.email">Invalid email address.</span>
    </span>
  </td>
</tr>
```

```
            <tr>
              <td>
                <button ng-click = "reset()">Reset</button>
              </td>
              <td>
                <button ng-disabled = "studentForm.firstname.$dirty &&
                  studentForm.firstname.$invalid || studentForm.lastname.$dirty &&
                  studentForm.lastname.$invalid || studentForm.email.$dirty &&
                  studentForm.email.$invalid" ng-click="submit()">Submit</button>
              </td>
            </tr>
          </table>
        </form>
      </div>
    </body>
</html>
```

*Output: Open textAngularJSFormValidation.htm in a web browser. See the result.

AngularJS Including Other HTML Pages with the ng-include Directive: HTML does not support embedding html pages within html page. To achieve this functionality the following ways are used:

*Using Ajax: Make a server call to get the corresponding html page and set it in innerHTML of html control.

*Using Server Side Includes: JSP, ASP.Net, PHP and other web side server technologies can include html pages within a dynamic page.

*Using AngularJS, we can embed HTML pages within a HTML page using ng-include directive.

```
<div ng-app = "" ng-controller = "studentController">
   <div ng-include = "'main.htm'"></div>
   <div ng-include = "'subjects.htm'"></div>
```

```
</div>
```

*Example

//File: main.htm

```
<table border = "0">
   <tr>
      <td>Enter first name:</td>
      <td><input type = "text" ng-model = "student.firstName"></td>
   </tr>
   <tr>
      <td>Enter last name: </td>
      <td><input type = "text" ng-model = "student.lastName"></td>
   </tr>
   <tr>
      <td>Name: </td>
      <td>{{student.fullName()}}</td>
   </tr>
</table>
```

//File: subjects.htm

```
<p>Subjects:</p>
<table>
   <tr>
      <th>Name</th>
      <th>Marks</th>
   </tr>
   <tr ng-repeat = "subject in student.subjects">
```

```
      <td>{{ subject.name }}</td>
      <td>{{ subject.marks }}</td>
    </tr>
</table>
//File: testAngularJSIncludes.htm
<html>
  <head>
    <title>Angular JS Includes</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
    <script>
      var mainApp = angular.module("mainApp", []);
      mainApp.controller('studentController', function($scope) {
        $scope.student = {
          firstName: "Thomas",
          lastName: "Albert",
          fees:500,
          subjects:[
            {name:'Physics',marks:70},
            {name:'Chemistry',marks:80},
            {name:'Math',marks:65},
            {name:'English',marks:75},
            {name:'German',marks:67}
          ],
          fullName: function() {
            var studentObject;
```

```
            studentObject = $scope.student;
            return studentObject.firstName + " " + studentObject.lastName;
          }
        };
      });
    </script>
    <style>
      table, th , td {
        border: 1px solid grey;
        border-collapse: collapse;
        padding: 5px;
      }
      table tr:nth-child(odd) {
        background-color: #f2f2f2;
      }
      table tr:nth-child(even) {
        background-color: #ffffff;
      }
    </style>
  </head>
  <body>
    <h2>AngularJS Sample Application</h2>
    <div ng-app = "mainApp" ng-controller="studentController">
      <div ng-include = "'/angularjs/src/include/main.htm'"></div>
      <div ng-include = "'/angularjs/src/include/subjects.htm'"></div>
```

```
    </div>

  </body>

</html>
```

*Output: To run this example, you need to deploy textAngularJS.htm, main.htm and subjects.htm to a webserver. Open textAngularJS.htm using url of your server in a web browser. See the result.

AngularJS Ajax using the $https Service: AngularJS provides $https: control which works as a service to read data from the server. The server makes a database call to get the desired records. AngularJS needs data in JSON format. Once the data is ready, $https: can be used to get the data from server in the following manner:

```
function studentController($scope,$https:) {

var url = "data.txt";

  $https:.get(url).success( function(response) {

    $scope.students = response;

  });

}
```

*Here, the file data.txt contains student records. $https: service makes an ajax call and sets response to its property students. students model can be used to draw tables in HTML.

*Examples:

data.txt

```
[

  {

    "Name" : "Thomas Albert",

    "RollNo" : 101,

    "Percentage" : "80%"

  },

  {

    "Name" : "James Doe",

    "RollNo" : 201,
```

```
    "Percentage" : "70%"
  },
  {
    "Name" : "Robert Doe",
    "RollNo" : 191,
    "Percentage" : "75%"
  },
  {
    "Name" : "Julian Joe",
    "RollNo" : 111,
    "Percentage" : "77%"
  }
]
//File: testAngularJSAjax.htm
<html>
  <head>
    <title>Angular JS Includes</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.2.15/angular.min.js"></script>
    <script>
      function studentController($scope,$http) {
        var url = "data.txt";
        $http.get(url).then( function(response) {
          $scope.students = response.data;
        });
      }
```

```
  </script>
  <style>
    table, th , td {
      border: 1px solid grey;
      border-collapse: collapse;
      padding: 5px;
    }
    table tr:nth-child(odd) {
      background-color: #f2f2f2;
    }
    table tr:nth-child(even) {
      background-color: #ffffff;
    }
  </style>
</head>
<body>
  <h2>AngularJS Sample Application</h2>
  <div ng-app = "" ng-controller = "studentController">
    <table>
      <tr>
        <th>Name</th>
        <th>Roll No</th>
        <th>Percentage</th>
      </tr>
      <tr ng-repeat = "student in students">
```

```
            <td>{{ student.Name }}</td>

            <td>{{ student.RollNo }}</td>

            <td>{{ student.Percentage }}</td>

          </tr>

        </table>

      </div>

    </body>

</html>
```

*Output: To execute this example, you need to deploy testAngularJSAjax.htm and data.txt file to a web server. Open the file testAngularJSAjax.htm using the URL of your server in a web browser and see the result.

AngularJS Views and Views Templates with the ng-view and ng-template Directives Respectively and the $routeProvider Service: AngularJS supports Single Page Application via multiple views on a single page. To do this AngularJS has provided ng-view and ng-template directives and $routeProvider services.

*ng-view: ng-view tag simply creates a place holder where a corresponding view (html or ng-template view) can be placed based on the configuration.

Usage: Define a div with ng-view within the main module.

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

   ...

   <div ng-view></div>

</div>
```

*ng-template: ng-template directive is used to create an html view using script tag. It contains "id" attribute which is used by $routeProvider to map a view with a controller.

Usage: Define a script block with type as ng-template within the main module.

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

   ...

   <script type = "text/ng-template" id = "addStudent.htm">

      <h2> Add Student </h2>
```

```
    {{message}}
  </script>
</div>
```

*$routeProvider: $routeProvider is the key service which set the configuration of urls, map them with the corresponding html page or ng-template, and attach a controller with the same.

Usage: Define a script block with main module and set the routing configuration.

```
var mainApp = angular.module("mainApp", ['ngRoute']);

mainApp.config(['$routeProvider', function($routeProvider) {
  $routeProvider.
  when('/addStudent', {
    templateUrl: 'addStudent.htm', controller: 'AddStudentController'
  }).
  when('/viewStudents', {
    templateUrl: 'viewStudents.htm', controller: 'ViewStudentsController'
  }).
  otherwise({
    redirectTo: '/addStudent'
  });
}]);
```

*Following are the important points to be considered in above example.

*$routeProvider is defined as a function under config of mainApp module using key as '$routeProvider'.

*$routeProvider.when defines a url "/addStudent" which then is mapped to "addStudent.htm". addStudent.htm should be present in the same path as main html page.If htm page is not defined then ng-template to be used with id="addStudent.htm". We've used ng-template.

*"otherwise" is used to set the default view.

*"controller" is used to set the corresponding controller for the view.

*Example: The following example will showcase all the above mentioned directives:

```
//File: testAngularJSViewsTemplates.htm

<html>

  <head>

    <title>Angular JS Views</title>

    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>

    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular-route.min.js"></script>

      <script type = "text/ng-template" id = "addStudent.htm">

        <h2> Add Student </h2>

        {{message}}

      </script>

      <script type = "text/ng-template" id = "viewStudents.htm">

        <h2> View Students </h2>

        {{message}}

      </script>

    <script>

      var mainApp = angular.module("mainApp", ['ngRoute']);

      mainApp.config(['$routeProvider', function($routeProvider) {

        $routeProvider.when('/addStudent', {

          templateUrl: 'addStudent.htm',

          controller: 'AddStudentController'

        }).when('/viewStudents', {

          templateUrl: 'viewStudents.htm',

          controller: 'ViewStudentsController'

        }).otherwise({

          redirectTo: '/addStudent'
```

```
        });
      }]);
      mainApp.controller('AddStudentController', function($scope) {
        $scope.message = "This page will be used to display add student form";
      });
      mainApp.controller('ViewStudentsController', function($scope) {
        $scope.message = "This page will be used to display all the students";
      });
    </script>
  </head>
  <body>
    <h2>AngularJS Sample Application</h2>
    <div ng-app = "mainApp">
      <p><a href = "#addStudent">Add Student</a></p>
      <p><a href = "#viewStudents">View Students</a></p>
      <div ng-view></div>
    </div>
  </body>
</html>
```

*Output: Open testAngularJSViewsTemplates.htm in a web browser. See the result.

AngularJS Scopes JavaScript Object $scope to Join the View and the Model by the Controller: Scope is a special javascript object which plays the role of joining controller with the views. Scope contains the model data. In controllers, model data is accessed via $scope object.

```
<script>
  var mainApp = angular.module("mainApp", []);
  mainApp.controller("shapeController", function($scope) {
```

```
    $scope.message = "In shape controller";

    $scope.type = "Shape";

  });

</script>
```

*Following are the important points to be considered in above example.

*$scope is passed as first argument to controller during its constructor definition.

*$scope.message and $scope.type are the models which are to be used in the HTML page.

*We've set values to models which will be reflected in the application module whose controller is shapeController.

*We can define functions as well in $scope.

*Scope Inheritance: Scope are controllers specific. If we defines nested controllers then child controller will inherit the scope of its parent controller.

```
<script>

  var mainApp = angular.module("mainApp", []);

  mainApp.controller("shapeController", function($scope) {

    $scope.message = "In shape controller";

    $scope.type = "Shape";

  });

  mainApp.controller("circleController", function($scope) {

    $scope.message = "In circle controller";

  });

</script>
```

*Following are the important points to be considered in above example.

*We've set values to models in shapeController.

*We've overridden message in child controller circleController. When "message" is used within module of controller circleController, the overridden message will be used.

*Example: The following example will showcase all the above mentioned directives.

//File: testAngularJSScopes.htm

```
<html>
  <head>
    <title>Angular JS Forms</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
    <script>
      var mainApp = angular.module("mainApp", []);
      mainApp.controller("shapeController", function($scope) {
        $scope.message = "In shape controller";
        $scope.type = "Shape";
      });
      mainApp.controller("circleController", function($scope) {
        $scope.message = "In circle controller";
      });
      mainApp.controller("squareController", function($scope) {
        $scope.message = "In square controller";
        $scope.type = "Square";
      });
    </script>
  </head>
  <body>
    <h2>AngularJS Sample Application</h2>
    <div ng-app = "mainApp" ng-controller = "shapeController">
      <p>{{message}} <br/> {{type}} </p>
      <div ng-controller = "circleController">
        <p>{{message}} <br/> {{type}} </p>
```

```
        </div>

        <div ng-controller = "squareController">

          <p>{{message}} <br/> {{type}} </p>

        </div>

      </div>

    </body>

</html>
```

*Output: Open testAngularJSScopes.htm in a web browser. See the result.

AngularJS Sample Application

In shape controller

Shape

In circle controller

Shape

In square controller

Square

AngularJS Services: AngularJS supports the concepts of "Separation of Concerns" using services architecture. Services are javascript functions and are responsible to do a specific tasks only. This makes them an individual entity which is maintainable and testable. Controllers, filters can call them as on requirement basis. Services are normally injected using dependency injection mechanism of AngularJS.

*AngularJS provides many inbuilt services for example, $https:, $route, $window, $location etc. Each service is responsible for a specific task for example, $https: is used to make ajax call to get the server data. $route is used to define the routing information and so on. Inbuilt services are always prefixed with $ symbol.

*There are two ways to create a service.

*(1) factory

*(2) service

*Using factory method to create a service: Using factory method, we first define a factory and then assign method to it.

```
var mainApp = angular.module("mainApp", []);
```

```
mainApp.factory('MathService', function() {
  var factory = {};
  factory.multiply = function(a, b) {
    return a * b
  }
  return factory;
});
```

*Using service method to create a service: Using service method, we define a service and then assign method to it. We've also injected an already available service to it.

```
mainApp.service('CalcService', function(MathService){
  this.square = function(a) {
    return MathService.multiply(a,a);
  }
});
```

*Example: The following example will showcase all the above mentioned directives:

```
//File: testAngularJSServices.htm
<html>
  <head>
    <title>Angular JS Services</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
    <script>
      var mainApp = angular.module("mainApp", []);
      mainApp.factory('MathService', function() {
        var factory = {};
        factory.multiply = function(a, b) {
          return a * b
```

```
        }
        return factory;
      });
      mainApp.service('CalcService', function(MathService){
        this.square = function(a) {
          return MathService.multiply(a,a);
        }
      });
      mainApp.controller('CalcController', function($scope, CalcService) {
        $scope.square = function() {
          $scope.result = CalcService.square($scope.number);
        }
      });
    </script>
  </head>
  <body>
    <h2>AngularJS Sample Application</h2>
    <div ng-app = "mainApp" ng-controller = "CalcController">
      <p>Enter a number: <input type = "number" ng-model = "number" /></p>
      <button ng-click = "square()">X<sup>2</sup></button>
      <p>Result: {{result}}</p>
    </div>
  </body>
</html>
```

*Output: testAngularJSServices.htm in a web browser. See the result.

AngularJS Dependency Injection: Dependency Injection is a software design pattern in which components are given their dependencies instead of hard coding them within the component. This relieves a component from locating the dependency and makes dependencies configurable. This helps in making components reusable, maintainable and testable.

*AngularJS provides a supreme Dependency Injection mechanism. It provides following core components which can be injected into each other as dependencies.

*(1) value

*(2) factory

*(3) service

*(4) provider

*(5) constant

*value Dependency Injection: value is simple javascript object and it is used to pass values to controller during config phase.

```
//define a module

var mainApp = angular.module("mainApp", []);

//create a value object as "defaultInput" and pass it a data.

mainApp.value("defaultInput", 5);

...

//inject the value in the controller using its name "defaultInput"

mainApp.controller('CalcController', function($scope, CalcService, defaultInput) {

  $scope.number = defaultInput;

  $scope.result = CalcService.square($scope.number);

  $scope.square = function() {

    $scope.result = CalcService.square($scope.number);

  }

});
```

*factory Dependency Injection: factory is a function which is used to return value. It creates value on demand whenever a service or controller requires. It normally uses a factory function to calculate and return the value.

```
//define a module

var mainApp = angular.module("mainApp", []);

//create a factory "MathService" which provides a method multiply to return multiplication of two numbers

mainApp.factory('MathService', function() {
   var factory = {};
   factory.multiply = function(a, b) {
      return a * b
   }
   return factory;
});

//inject the factory "MathService" in a service to utilize the multiply method of factory.

mainApp.service('CalcService', function(MathService){
   this.square = function(a) {
      return MathService.multiply(a,a);
   }
});

...
```

*service Pependency Injection: service is a singleton javascript object containing a set of functions to perform certain tasks. Services are defined using service() functions and then injected into controllers.

```
//define a module

var mainApp = angular.module("mainApp", []);

...

//create a service which defines a method square to return square of a number.

mainApp.service('CalcService', function(MathService){
   this.square = function(a) {
      return MathService.multiply(a,a);
```

```
  }

});

//inject the service "CalcService" into the controller

mainApp.controller('CalcController', function($scope, CalcService, defaultInput) {

  $scope.number = defaultInput;

  $scope.result = CalcService.square($scope.number);

  $scope.square = function() {

    $scope.result = CalcService.square($scope.number);

  }

});
```

*provider Dependency Injection: provider is used by AngularJS internally to create services, factory etc. during config phase(phase during which AngularJS bootstraps itself). Below mention script can be used to create MathService that we've created earlier. Provider is a special factory method with a method get() which is used to return the value/service/factory.

```
//define a module

var mainApp = angular.module("mainApp", []);

...

//create a service using provider which defines a method square to return square of a number.

mainApp.config(function($provide) {

  $provide.provider('MathService', function() {

    this.$get = function() {

      var factory = {};

      factory.multiply = function(a, b) {

        return a * b;

      }

      return factory;

    };
```

```
  });

});
```

*constant Dependency Injection: constants are used to pass values at config phase considering the fact that value can not be used to be passed during config phase.

```
mainApp.constant("configParam", "constant value");
```

*Example: The following example will showcase all the above mentioned directives:

```
//File: testAngularJSDependencyInjection.htm

<html>
  <head>
    <title>AngularJS Dependency Injection</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
    <script>
      var mainApp = angular.module("mainApp", []);
      mainApp.config(function($provide) {
        $provide.provider('MathService', function() {
          this.$get = function() {
            var factory = {};
            factory.multiply = function(a, b) {
              return a * b;
            }
            return factory;
          };
        });
      });
      mainApp.value("defaultInput", 5);
      mainApp.factory('MathService', function() {
```

```
      var factory = {};
      factory.multiply = function(a, b) {
        return a * b;
      }
      return factory;
    });
    mainApp.service('CalcService', function(MathService){
      this.square = function(a) {
        return MathService.multiply(a,a);
      }
    });
    mainApp.controller('CalcController', function($scope, CalcService, defaultInput) {
      $scope.number = defaultInput;
      $scope.result = CalcService.square($scope.number);
      $scope.square = function() {
        $scope.result = CalcService.square($scope.number);
      }
    });
  </script>
</head>
<body>
  <h2>AngularJS Sample Application</h2>
  <div ng-app = "mainApp" ng-controller = "CalcController">
    <p>Enter a number: <input type = "number" ng-model = "number" /></p>
    <button ng-click = "square()">X<sup>2</sup></button>
```

```
      <p>Result: {{result}}</p>

    </div>

  </body>

</html>
```

*Output: Open testAngularJSDependencyInjection.htm in a web browser. See the result.

AngularJS Custom Directives: Custom directives are used in AngularJS to extend the functionality of HTML. Custom directives are defined using "directive" function. A custom directive simply replaces the element for which it is activated. AngularJS application during bootstrap finds the matching elements and do one time activity using its compile() method of the custom directive then process the element using link() method of the custom directive based on the scope of the directive. AngularJS provides support to create custom directives for following type of elements.

*(1) Element directives: Directive activates when a matching element is encountered.

*(2) Attribute directives: Directive activates when a matching attribute is encountered.

*(3) CSS directives: Directive activates when a matching css style is encountered.

*(4) Comment directive: Directive activates when a matching comment is encountered.

*Understanding Custom Directive:

*(1) Define custom html tags.

```
<student name = "Thomas"></student><br/>

<student name = "John"></student>
```

*(2) Define custom directive to handle above custom html tags.

```
var mainApp = angular.module("mainApp", []);

//Create a directive, first parameter is the html element to be attached.

//We are attaching student html tag.

//This directive will be activated as soon as any student element is encountered in html

mainApp.directive('student', function() {

  //define the directive object

  var directive = {};

  //restrict = E, signifies that directive is Element directive
```

```
   directive.restrict = 'E';

   //template replaces the complete element with its text.

   directive.template = "Student: <b>{{student.name}}</b> , Roll No: <b>{{student.rollno}}</b>";

   //scope is used to distinguish each student element based on criteria.

   directive.scope = {

      student : "=name"

   }

   //compile is called during application initialization. AngularJS calls it once when html page is loaded.

   directive.compile = function(element, attributes) {

      element.css("border", "1px solid #cccccc");

      //linkFunction is linked with each element with scope to get the element specific data.

      var linkFunction = function($scope, element, attributes) {

         element.html("Student: <b>"+$scope.student.name +"</b> , Roll No:
<b>"+$scope.student.rollno+"</b><br/>");

         element.css("background-color", "#ff00ff");

      }

      return linkFunction;

   }

   return directive;

});
```

*(3) Define controller to update the scope for directive. Here we are using name attribute's value as scope's child.

```
mainApp.controller('StudentController', function($scope) {

   $scope.Thomas = {};

   $scope.Thomas.name = "Thomas Albert";

   $scope.Thomas.rollno  = 1;

   $scope.John = {};
```

```
   $scope.John.name = "John Doe";
   $scope.John.rollno  = 2;
});
*Example:
//File: textAngularJSCustomDirectives.htm
<html>
   <head>
      <title>Angular JS Custom Directives</title>
      <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
      <script>
         var mainApp = angular.module("mainApp", []);
         mainApp.directive('student', function() {
            var directive = {};
            directive.restrict = 'E';
            directive.template = "Student: <b>{{student.name}}</b> , Roll No: <b>{{student.rollno}}</b>";
            directive.scope = {
               student : "=name"
            }
            directive.compile = function(element, attributes) {
               element.css("border", "1px solid #cccccc");
               var linkFunction = function($scope, element, attributes) {
                  element.html("Student: <b>"+$scope.student.name +"</b> , Roll No:
<b>"+$scope.student.rollno+"</b><br/>");
                  element.css("background-color", "#ff00ff");
               }
               return linkFunction;
```

```
        }
        return directive;
      });
      mainApp.controller('StudentController', function($scope) {
        $scope.Thomas = {};
        $scope.Thomas.name = "Thomas Albert";
        $scope.Thomas.rollno  = 1;
        $scope.John = {};
        $scope.John.name = "John Doe";
        $scope.John.rollno  = 2;
      });
    </script>
  </head>
  <body>
    <h2>AngularJS Sample Application</h2>
    <div ng-app = "mainApp" ng-controller = "StudentController">
      <student name = "Thomas"></student><br/>
      <student name = "John"></student>
    </div>
  </body>
</html>
```

*output: Open textAngularJSCustomDirectives.htm in a web browser. See the result.

AngularJS Internationalization: AngularJS supports inbuilt internationalization for three types of filters currency, date and numbers. We only need to incorporate corresponding js according to locale of the country. By default it handles the locale of the browser. For example, to use Danish locale, use following script.

```
<script src = "https://code.angularjs.org/1.2.5/i18n/angular-locale_da-dk.js"></script>
```

*Example using Danish locale

```
//File: testAngularJSInternalizationDanishLocale.htm
<html>
  <head>
    <title>Angular JS Forms</title>
  </head>
  <body>
    <h2>AngularJS Sample Application</h2>
    <div ng-app = "mainApp" ng-controller = "StudentController">
      {{fees | currency }}  <br/><br/>
      {{admissiondate | date }}  <br/><br/>
      {{rollno | number }}
    </div>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
    <script src = "https://code.angularjs.org/1.3.14/i18n/angular-locale_da-dk.js"></script>
    <script>
      var mainApp = angular.module("mainApp", []);
      mainApp.controller('StudentController', function($scope) {
        $scope.fees = 100;
        $scope.admissiondate  = new Date();
        $scope.rollno = 123.45;
      });
    </script>
  </body>
</html>
```

*Output: Open testAngularJSInternalizationDanishLocale.htm in a web browser. See the result.

*Example using Browser's locale:

//File: testAngularJSInternalizationBrowserLocale.htm

```
<html>
  <head>
    <title>Angular JS Forms</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
    <!-- <script src = "https://code.angularjs.org/1.3.14/i18n/angular-locale_da-dk.js"></script> -->
    <script>
      var mainApp = angular.module("mainApp", []);
      mainApp.controller('StudentController', function($scope) {
        $scope.fees = 100;
        $scope.admissiondate  = new Date();
        $scope.rollno = 123.45;
      });
    </script>
  </head>
  <body>
    <h2>AngularJS Sample Application</h2>
    <div ng-app = "mainApp" ng-controller = "StudentController">
      {{fees | currency }}  <br/><br/>
      {{admissiondate | date }}  <br/><br/>
      {{rollno | number }}
    </div>
  </body>
```

</html>

*Output: Open testAngularJSInternalizationBrowserLocale.htm in a web browser. See the result.

Angular 2.0 JavaScript Framework for Web Development:

XXXStart Using Arrow Functions, p.77.

XXXXXStart with Creating a Two-Way Data Binding, p.27.

*Ref: Freeman, “Pro Angular”, Second Edition, 2017, Apress.

http://file.allitebooks.com/20170203/Pro%20Angular,%202nd%20Edition.pdf

*Ref: Freeman, “Pro AngularJS”, First Edition, 2016, Apress.

http://file.allitebooks.com/20150626/Pro%20AngularJS.pdf

*Ref: Rodrigo Branas, AngularJS Essentials, First Edition, 2014, Packt Publishing.

http://www.ebooksbucket.com/uploads/itprogramming/javascript/AngularJS-Essentials-Branas-Rodrigo.pdf

*HTML Web Development Guide: https://developer.mozilla.org/en-US/docs/Learn/HTML

*Angular is an open source JavaScript library that is sponsored and maintained by Google. Angular provides a high-productivity web development experience. Angular has been used in some of the largest and most complex web apps around.

*Angular is an open-source JavaScript library that lets you create great web-sites by using just HTML, CSS, and JavaScript to write the frontend code. Angular.JS allows us to create amazing web applications with so little code.

*Angular includes a lot of built-in functionality, and provides endless customization options.

*Angular also provides Reactive Extensions package, which underpins a lot of the functionality provided by Angular and is used directly by advanced features, and provides asynchronous HTTP requests in an Angular application, URL routing to navigate around an application, and animate HTML elements when the state of the application changes.

*Angular provides us with many data handling techniques, providing a complete set of technologies that are capable to accomplish any challenge related to present, transform, and validate data on the user's interface.
*Angular was built on the belief that declarative programming is the best choice to construct the user interface, while imperative programming is much better and preferred to implement an application's business logic.
*To achieve this, Angular empowers traditional HTML by extending its current vocabulary. The result is the development of expressive, reusable, and maintainable application components, leaving behind a lot of unnecessary code and keeping the team focused on the valuable and important things.
*Angular is a part of a new generation of libraries and frameworks to support the development of more productive, flexible, maintainable, and testable web applications.

*Angular Vs. AngularJS: Angular, also known as Angular 2, is the second major release of the framework originally known as AngularJS.
*The original AngularJS has been widely popular but was awkward to use and required developers to deal with some arcane and oddly implemented features that made web application development more complex than it needed to be.
*Angular, which we cover here, is a complete rewrite that is easier to learn, is easier to work with, and is much more consistent. It is still a complex framework, but creating web applications with Angular is a more pleasant experience than with AngularJS.
*Although the differences between AngularJS and Angular are so profound, if you have an AngularJS application that you want to upgrade to Angular, then you can use the upgrade adapter, which allows code from both versions of the framework to coexist in the same application. See https://angular.io/docs/ts/ latest/guide/upgrade.html for details.
*This can ease the transition, although AngularJS and Angular are so different that it is best to make a clean start and switch to Angular for a ground-up rewrite. This isn't always possible, of course, especially for complex applications, but the process of migrating while also managing coexistence is a difficult one to master and can lead to problems that are hard to track down and correct.

Two Types of Angular Applications: the Round-Trip and Single-Page Applications: In broad terms, there are two kinds of web application: round-trip and single-page applications.
*For a long time, web apps were developed to follow a round-trip model. The browser requests an initial HTML document from the server. User interactions, such as clicking a link or submitting a form, led the browser to request and receive a completely new HTML document.
*In this kind of application, the browser is essentially a rending engine for HTML content, and all of the application logic and data resides on the server. The browser makes a series of stateless HTTP requests that the server handles by generating HTML documents dynamically. A lot of current web development is still for round-trip applications, not least because they require little from the browser, which ensures the widest possible client support.
*But there are some serious drawbacks to round-trip applications: They make the user wait while the next HTML document is requested and loaded, they require a large server-side infrastructure to process all the requests and manage all the application state, and they require a lot of bandwidth because each HTML document has to be self-contained, leading to a lot of the same content being included in each response from the server.
*Single-page applications take a different approach. An initial HTML document is sent to the browser, but user interactions lead to Ajax requests for small fragments of HTML or data inserted into the existing set of elements being displayed to the user. The initial HTML document is never reloaded or replaced, and the user can continue to interact with the existing HTML while the Ajax requests are being performed asynchronously, even if that just means seeing a "data loading" message.
*Most current apps fall somewhere between the extremes, tending to use the basic round-trip model enhanced with JavaScript to reduce the number of complete page changes, although the emphasis is often on reducing the number of form errors by performing client-side validation.
*Angular gives the greatest return from its initial workload as an application gets closer to the single-page model. That's not to say you can't use Angular with round-trip applications; you can, of course, but there are other technologies that are simpler and better suit discrete HTML pages, either working directly with the Document Object Model (DOM) API or using a library to simplify its use, such as jQuery.
*Angular excels in single-page applications and especially in complex round-trip applications. For simple projects, using the DOM API directly or a library like jQuery is generally a better choice, although nothing prevents you from using Angular in all of your projects.
*There is a tendency for current web app projects to move toward the single-page application model, not just because of the initialization process but because the benefits of using the MVC pattern, really start to manifest themselves in larger and more complex projects, which benefit the most from single-page model.
*Comparing Angular and jQuery: Angular and jQuery take different approaches to web app development.
*jQuery explicitly manipulates the browser's Document Object Model (DOM) to create an application.
*jQuery is, without any doubt, a powerful tool. jQuery is robust and reliable, and you can get results pretty much immediately. jQuery has a fluid, and you can extend the jQuery library easily.
*But jQuery isn't the right tool for every job any more than Angular is. It can be hard to write and manage large applications using jQuery, and thorough unit testing can be a challenge.
*Angular takes the approach to co-opt the browser into being the foundation for application development.
*Angular also uses the DOM to present HTML content to users but takes an entirely different path to

building applications, focusing more on the data in the application and associating it to HTML elements through dynamic data bindings inside the browser.
*The main drawback of Angular is that there is an up-front investment in development time before you start to see results; something that is common in any MVC-based development. This initial investment is worthwhile, however, for complex apps.
*So, in short, use jQuery, or use the JavaScript DOM API directly, for low-complexity web apps where unit testing isn't critical and you require immediate results. Use Angular for single-page web apps, when you have time for careful design and when you can easily control the HTML generated by the server.

Angular Architectural Concepts: The Model-View-Controller (MVC) for the Browser:
*The goal of Angular is to bring the tools and capabilities that have been available only for server-side development to the web client and, in doing so, make it easier to develop, test, and maintain rich and complex web applications. Angular delivers the kind of functionality that used to be available only to server-side developers, but entirely in the browser. This means Angular has a lot of work to do each time an HTML document to which Angular has been applied is loaded: the HTML elements have to be compiled, the data bindings have to be evaluated, components and other building blocks need to be executed, and so on.
*This kind of work takes time to perform, and the amount of time depends on the complexity of the HTML document, on the associated JavaScript code, and critically, on quality of the browser and the processing capability of the device. You won't notice any delay when using the latest browsers on a capable desktop machine, but old browsers on underpowered smartphones can really slow down the initial setup of an Angular app. The goal, therefore, is to perform this setup as infrequently as possible and deliver as much of the app as possible to the user when it is performed. This means giving careful thought to the kind of web application you build.
*Angular works by allowing you to extend HTML. Angular applications express functionality through custom elements, and a complex application can produce an HTML document that contains a mix of standard and custom markup.
*The style of development that Angular supports is derived through the use of the Model-View-Controller (MVC) pattern, although this is sometimes referred to as Model-View-Whatever, since there are countless variations on this pattern that can be adhered to when using Angular.
*We'll focus on the standard MVC pattern, since it is the most established and widely used.
*The Model-View-Controller (MVC) pattern has become very popular and has become one of the legends of the enterprise architecture design.
*Basically, the model represents the knowledge that the view is responsible for presentation, while the controller mediates the relationship between model and view.
*Angular adopts a Model-View-Whatever (MVW) architecture. Regardless of the name, the most important benefit is that the framework provides a clear separation of the concerns between the application layers, providing modularity, flexibility, and testability.
*In terms of concepts, a typical Angular application consists primarily of a view, model, and controller, but there are other important components, such as services, directives, and filters.
*The view, also called template, is entirely written in HTML, which provides a great opportunity to see web designers and JavaScript developers working side by side.
*It also takes advantage of the directives mechanism, which is a type of extension of the HTML vocabulary that brings the ability to perform programming language tasks such as iterating over an array or even evaluating an expression conditionally.
*Behind the view, there is the controller. At first, the controller contains all the business logic implementation used by the view. However, as the application grows, it becomes really important to perform some refactoring activities, such as moving the code from the controller to other components (for example, services) in order to keep the cohesion high.
*The connection between the view and the controller is done by a shared object called scope. It is located between them and is used to exchange information related to the model.
*The model is a simple Plain-Old-JavaScript-Object (POJO). It looks very clear and easy to understand, bringing simplicity to the development by not requiring any special syntax to be created.
*There is wide support for Angular in popular development tools, and you can pick your favorites.
*Understanding the MVC Pattern for Web Applications: The term Model-View-Controller was conceived as a way to organize some early GUI applications, and they are especially well-suited to web applications.
*The MVC pattern first took hold in the server-side end of web development, through toolkits like Ruby on Rails, Java Struts MVC, and the ASP.NET MVC Framework.

*In recent years, the MVC pattern has been seen as a way to manage the growing richness and complexity of client-side web development as well, and it is in this environment that Angular has emerged.
*The key to applying the MVC pattern is to implement the key premise of a separation of concerns, in which the data model in the application is decoupled from the business and presentation logic.
*In clientside web development, this means separating the data, the logic that operates on that data, and the HTML elements used to display the data. The result is a client-side application that is easier to develop, maintain, and test.
*The three main building blocks are the model, the controller, and the view.
*Of course, Angular exists in the browser, which leads to a twist on the MVC theme, as illustrated below:

Fig: XXXCopy Figure 3-3. A client-side implementation of the MVC pattern.
*The client-side implementation of the MVC pattern gets its data from server-side components, usually via a RESTful web service. The goal of the controller and the view is to operate on the data in the model in order to perform DOM manipulation so as to create and manage HTML elements that the user can interact with. Those interactions are fed back to the controller, closing the loop to form an interactive application.
*Angular uses slightly different terminology for its building blocks, which means that the MVC model implemented using Angular looks more like the following:

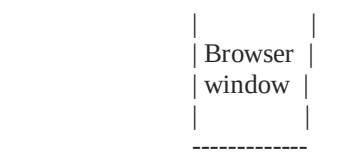

Fig: XXXCopy Figure 3-4. The Angular implementation of the MVC pattern.
*The figure shows the basic mapping of Angular building blocks to the MVC pattern. To support the MVC pattern, Angular provides a broad set of additional features, which we describe:
*(1) Component's job: Contains logic that transforms the model based on user interaction.
*(2) Angular HTML Template's job: Contains logic for displaying data to the user by manipulating the DOM based on user interaction.
*Note: Using a client-side framework like Angular doesn't preclude using a server-side MVC framework, but you'll find that an Angular client takes on some of the complexity that would have otherwise existed at the server. This is generally a good thing because it offloads work from the server to the client, and that allows for more clients to be supported with less server capacity.

Installing Node.js and NPM: Many of the tools used for Angular development rely on Node.js, also known as Node, which was created in 2009 as a simple and efficient runtime for server-side applications written in JavaScript.
*Node.js is based on the JavaScript engine used in the Chrome browser and provides an API for executing JavaScript code outside of the browser environment.
*Node.js has enjoyed success as an application server, but it has also provided the foundation for a new generation of cross-platform development and build tools. Node.js has become an essential tool for web application development.
*The version we use here is the 6.9.1 release. There are more recent releases for your own projects
*A complete set of 6.9.1 releases, with installers for Windows and Mac OS and binary packages for other platforms, is available at: https://nodejs.org/dist/v6.9.1.
*When you install Node.js, make sure you select the installer option to add the Node.js executables to the path. When installation is complete, run the following command:
node -v
If the installation has gone as it should, then you will see the following version number displayed:
V6.9.1
*Updating NPM: The success of Node.js has been helped by the Node Package Manager (NPM), which provides easy access to an enormous catalog of development packages, including Angular. NPM takes care of downloading packages and managing the dependencies between them.

*Part of the initial process is updating the version of NPM that was installed with Node.js, which is done by running the following command, with the root or administrator privileges:
npm install -g npm@3.10.9
*Installing an Editor: Angular development can be done with any programmer's editor. Some editors have enhanced support for working with Angular, including highlighting key terms and good tool integration. The following are some of the free open-source editors for web application development:
*(1) Brackets IDE: Brackets is a free open source editor developed by Adobe. See brackets.io for details.
*(2) Atom IDE: Atom is a free, open source, cross-platform editor that has a particular emphasis on customization and extensibility. See atom.io for details.
*Installing a Browser: The final choice to make is the browser that you will use to check your work during development. All the current-generation browsers have good developer support and work well with Angular. Google Chrome is a good choice.

Installing all the NPM Packages that are Required for a Project:
*Creating a Development Package File package.json: NPM uses a file called package.json to get a list of the software packages that are required for a project. The package.json file lists the packages required to get started with Angular development and some commands to use them.
*Each configuration entry section in package.json is described below:
*(1) dependencies: This is a list of NPM packages that the web application relies on to run. Each package is specified with a version number. The dependencies section in the listing contains the core Angular packages, libraries that Angular depends on, polyfill libraries that add modern features for old browsers, and the Bootstrap CSS library that we use to style HTML.
*(2) devDependencies: This is a list of NPM packages that are relied on for development but that are not required by the application once it has been deployed. This section contains packages that will compile TypeScript files, provide a development HTTP server, and allow multiple commands to be run at the same time using NPM.
*(3) scripts: This is a list of scripts that can be run from the command line. The scripts section in the listing starts the TypeScript compiler and the development HTTP server.
*The following is a sample package.json to be placed in the development folder:

```
//File: package.json
{
  "dependencies":
  {
    "@angular/common": "2.2.0",
    "@angular/compiler": "2.2.0",
    "@angular/core": "2.2.0",
    "@angular/forms": "2.2.0",
    "@angular/platform-browser": "2.2.0",
    "@angular/platform-browser-dynamic": "2.2.0",
    "reflect-metadata": "0.1.8",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "0.6.26",
    "core-js": "2.4.1",
    "classlist.js": "1.1.20150312",
    "systemjs": "0.19.40",
    "bootstrap": "4.0.0-alpha.4"
  },
  "devDependencies":
  {
    "lite-server": "2.2.2",
    "typescript": "2.0.3",
    "typings": "1.4.0",
    "concurrently": "3.1.0"
  },
  "scripts":
  {
    "start": "concurrently \"npm run tscwatch\" \"npm run lite\" ",
```

```
    "tsc": "tsc",
    "tscwatch": "tsc -w",
    "lite": "lite-server",
    "typings": "typings"
  }
}
```
*Installing the NPM Packages for Development: To process the package.json file to download and install the packages that it specifies, run the following command inside the root application folder:
npm install
*Once the installation process has completed, you will be left with a node_modules directory in your root application folder that contains all the packages specified in the package.json file, along with their dependencies.
*There will be a lot of packages in the node_modules folder because the tendency in NPM development is to build on existing functionality that other packages contain, which is good development practice but does mean the initial download and installation can take a while.
*Configuring the TypeScript Compiler: Angular applications are written in TypeScript, which is a superset of JavaScript. Working with TypeScript provides some useful advantages, but requires that TypeScript files are processed to generate backward-compatible JavaScript that can be used by browsers.
*TypeScript simplifies working with Angular and, with just a few exceptions, you can write Angular applications using the JavaScript skills you already have.
*The TypeScript compiler requires a configuration file to control the kind of JavaScript files that it generates.
*The following is a sample file called tsconfig.json in the root application folder and added the configuration statements shown below:
```
//File: tsconfig.json:
{
  "compilerOptions":
  {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true
  },
  "exclude": [ "node_modules" ]
}
```
*What each of these configurations is explained later.
*Installing the TypeScript Type Information: The TypeScript compiler relies on descriptions of the standard JavaScript APIs, known as type definitions, for the enhancements that TypeScript provides to the JavaScript language specification. These descriptions are required to prevent the compiler reporting errors. Run the following commands in the root application folder:
npm run typings -- install dt ~ core-js --save –global
npm run typings -- install dt ~ node --save –global
*Once the commands have completed, the rppt development folder will contain typings/globals/core-js and typings/global/node folders that contain the type definitions and a typings.json file that contains details of the type information that was downloaded, as shown below. You may see different version numbers in your typings.json file.
```
//File: typings.json:
{
  "globalDependencies":
  {
    "core-js": "registry:dt/core-js#0.0.0+20160914114559",
    "node": "registry:dt/node#6.0.0+20161110151007"
  }
}
```

*Creating the HTML File: We'll start with an HTML placeholder that contains static content, which we'll later enhance using Angular. We create an HTML file called index.html in the root application folder and added the markup shown below:

```
//File: index.html:
<!DOCTYPE html>
<html>
<head>
  <title>ToDo</title>
  <meta charset="utf-8" />
  <link href="node_modules/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
  <h1>Content will go here</h1>
</body>
</html>
```

*This is a basic HTML document with a link element that tells the browser to load the Bootstrap CSS stylesheet from the package installed in the node_modules folder, installed by NPM earlier.

*Starting the Server: The project tools and basic structure are in place, so it is time to test that everything is working. Run the following command from the root application folder:

```
npm start
```

*This command tells npm to run the start script, which starts the TypeScript compiler and the light-weight development HTTP server that was installed using the package.json file.

*After a few seconds, a new browser window will start, and you will see the placeholder content from index.html.

*Preparing an HTML File to Add Support for Angular: The first step toward adding Angular to the application is to prepare the index.html file, as shown below. The new script elements add the JavaScript files for the Angular framework, and the third-party JavaScript libraries and polyfill that provide compatibility for older browsers. These files are all from the NPM packages added to the package.json file earlier. The scripts must be added in the order shown.

```
//File: index.html: Preparing for Angular in the index.html File:
<!DOCTYPE html>
<html>
<head>
  <title>ToDo</title>
  <meta charset="utf-8" />
  <link href="node_modules/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
  <script src="node_modules/classlist.js/classList.min.js"></script>
  <script src="node_modules/core-js/client/shim.min.js"></script>
  <script src="node_modules/zone.js/dist/zone.min.js"></script>
  <script src="node_modules/reflect-metadata/Reflect.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>
</head>
<body class="m-a-1">
  <todo-app>Angular placeholder</todo-app>
</body>
</html>
```

*JavaScript code is added to an HTML document using the script element. The src attribute is used to specify which JavaScript file should be loaded. The script elements add the JavaScript files that Angular applications rely on. There are no entries for Angular functionality or other the application features yet; we'll add them later.

*The second change above is to replace the content of the body element and replace it with a todo-app element. There is no todo-app element in the HTML specification and the browser will ignore it when parsing the HTML file, but this element will be the entry point into the world of Angular and will be replaced with my application content. When you save the index.html file, the browser will reload the file and show the placeholder message.

*Replacing the HTML Content: The final test of the development environment is to change the content in the index.html file, shown below:

```
//File: index.html:
<!DOCTYPE html>
<html>
<head>
  <title>ToDo</title>
  <meta charset="utf-8" />
  <link href="node_modules/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body class="m-a-1">
  <h3 class="bg-primary p-a-1">Adam's To Do List</h3>
  <div class="m-t-1 m-b-1">
    <input class="form-control" />
    <button class="btn btn-primary m-t-1">Add</button>
  </div>
  <table class="table table-striped table-bordered">
    <thead>
      <tr>
        <th>Description</th>
        <th>Done</th>
      </tr>
    </thead>
    <tbody>
      <tr><td>Buy Flowers</td><td>No</td></tr>
      <tr><td>Get Shoes</td><td>No</td></tr>
      <tr><td>Collect Tickets</td><td>Yes</td></tr>
      <tr><td>Call Joe</td><td>No</td></tr>
    </tbody>
  </table>
</body>
</html>
```

*The lite-server package, which is the development HTTP server, adds a fragment of JavaScript to the HTML content it delivers to the browser.
*The JavaScript opens a connection back to the server and waits for a signal to reload the page, which is sent when the server detects a change in any of the files in the root development directory.
*As soon as you save the index.html file, HTTP lite-server will detect the change and send the signal, and trigger a reload in the browser, reflecting the new content.
*The HTML elements in the index.html file show how the simple Angular application will look.
*The key elements are a banner with the user's name, an input element and an Add button that add a new to-do item to the list, and a table that contains all the to-do items and indicates whether they have been completed.
*We also use the excellent Bootstrap CSS framework to style HTML content. Bootstrap is applied by assigning elements to classes, like this:

```
<h3 class="bg-primary p-a-1">Adam's To Do List</h3>
```

*This h3 element has been assigned to two classes. The bg-primary class sets the background color of the element to the primary color of the current Bootstrap theme. I am using the default theme, for which the primary color is dark blue and there are other themed colors available, including bg-secondary, bg-info, and bg-danger. The p-a-1 class adds a fixed amount of padding to all edges of the element, ensuring that the text has some space around it.

The Angular Compilers: Angular Just-in-Time (JIT) and Ahead-of-Time (AOT) Compiler:
*Ref: https://angular.io/guide/aot-compiler
*An Angular application consists largely of Angular components and their HTML templates. Before the browser can render the application, the components and templates must be converted to executable JavaScript by an Angular compiler.
*Angular offers two ways to compile your application:
1.Just-in-Time (JIT), which compiles your app in the browser at runtime
2.Ahead-of-Time (AOT), which compiles your app at build time.

*JIT compilation is the default when you run the build-only or the build-and-serve-locally CLI commands:
ng build
ng serve
*The Angular Ahead-of-Time (AOT) compiler converts your Angular HTML and TypeScript code into efficient JavaScript code during the build phase before the browser downloads and runs that code. We look at how to build with the AOT compiler and how to write Angular metadata that AOT can compile.
For AOT compilation, append the --aot flags to the build-only or the build-and-serve-locally CLI commands:
ng build --aot
ng serve --aot
The --prod meta-flag compiles with AOT by default.

Using the JavaScript export Keyword to Export Types that can be Imported in Other JavaScript Modules:
*The JavaScript export Keyword:
*Ref: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export
*The export statement is used when creating JavaScript modules to export functions, objects, or primitive values from the module so they can be used by other programs with the import statement.
*This feature is only implemented natively in Safari and Chrome at this time. It is implemented in many transpilers, such as the Traceur Compiler, Babel or Rollup.
*Syntax:
export { name1, name2, …, nameN };
export { variable1 as name1, variable2 as name2, …, nameN };
export let name1, name2, …, nameN; // also var, function
export let name1 = …, name2 = …, …, nameN; // also var, const
export default expression;
export default function (…) { … } // also class, function*
export default function name1(…) { … } // also class, function*
export { name1 as default, … };
export * from …;
export { name1, name2, …, nameN } from …;
export { import1 as name1, import2 as name2, …, nameN } from …;
*Where:
*nameN: Identifier to be exported, so that it can be imported via import in another script.
*Description: There are two different types of export. Each type corresponds to one of the above syntax:
*(1) Named exports:
//exports a function declared earlier:
export { myFunction };
//exports a constant:
export const foo = Math.sqrt(2);
*(2) Default exports (function):
export default function() {}
*(3) Default exports (class):
export default class {}
*Named exports are useful to export several values. During the import, it is mandatory to use the same name of the corresponding object.
*But a default export can be imported with any name for example:
export default k = 12; //in file test.js
import m from './test' //note that we got the freedom to use import m instead of import k, because k was default export
console.log(m); //will log 12
*There can be only one default export.
*The following syntax does not export a default export from the imported module:
export * from …;
*If you need to export the default, write the following instead:
import mod from "mod";
export default mod;
*Examples

*(1) Using named exports: In the module, we could use the following code:
```
//module "my-module.js"
function cube(x)
{
  return x * x * x;
}
const foo = Math.PI + Math.SQRT2;
export { cube, foo };
```
*This way, in another script, we could have:
```
import { cube, foo } from 'my-module';
console.log(cube(3)); // 27
console.log(foo);     // 4.555806215962888
```
*(2) Using the default export: If we want to export a single value or to have a fallback value for our module, we could use a default export:
```
//module "my-module.js"
export default function cube(x)
{
   return x * x * x;
}
```
*Then, in another script, it will be straightforward to import the default export:
```
import cube from 'my-module';
console.log(cube(3)); // 27
```
*Note that it is not possible to use var, let or const with export default.

Using the JavaScript import Keyword to Import Exported Types in Other JavaScript Modules:
*The JavaScript import keyword is the counterpart to the export keyword used above, and is used to declare a dependency on the contents of a JavaScript module.
*The import keyword is used twice above, as shown here:
```
import { Component } from "@angular/core";
import { Model } from "./model";
```
*The import statement is used to specify the types that are imported, between curly braces, from modules.
*The first import statement in the listing above is used to load the @angular/core module, which contains the key Angular functionality, including support for components. The @angular/core module contains many classes that have been packaged together so that the browser can load them all in a single JavaScript file. In this case, the import statement is used to load the Component type from the module.
*The second import statement is used to load the Model class from a file in the project. The target for this kind of import starts with ./, which indicates that the module is defined relative to the current file.
*Notice that neither import statement includes a file extension. This is because the relationship between the target of an import statement and the file that is loaded by the browser is managed by a module loader, which we configure earlier in the in the "Putting the Application Together" section.
*The JavaScript import Keyword:
*Ref: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import
*The import statement is used to import bindings which are exported by another module. Bindings are used to refer to property and method names.
*This feature is only just beginning to be implemented in browsers natively at this time. It is implemented in many transpilers, such as TypeScript and Babel, and bundlers such as Rollup and Webpack.
*Syntax:
```
import defaultExport from "module-name";
import * as name from "module-name";
import { export } from "module-name";
import { export as alias } from "module-name";
import { export1 , export2 } from "module-name";
import { export1 , export2 as alias2 , [...] } from "module-name";
import defaultExport, { export [ , [...] ] } from "module-name";
import defaultExport, * as name from "module-name";
import "module-name";
```

*Where:
*defaultExport: Name that will refer to the default export from the module.
*module-name: The module to import from. This is often a relative or absolute path name to the .js file containing the module, excluding the .js extension. Certain bundlers may permit or require the use of the extension; check your environment. Only single quotes and double quotes Strings are allowed.
*name: Name of the module object that will be used as a kind of namespace when referring to the imports. The name parameter is the name of the "module object" which will be used as a kind of namespace to refer to the exports. The export parameters specify individual named exports, while the import * as name syntax imports all of them. Some examples are given below to clarify the syntax.
*export, exportN: Name of the exports to be imported.
*alias, aliasN: Names that will refer to the named imports.
*Examples:
*(1) Import an entire module's contents: This inserts myModule into the current scope, containing all the exports from the module in the file located in /modules/my-module.js.
import * as myModule from '/modules/my-module.js';
Here, accessing the exports means using the module name, "myModule" in this case, as a namespace. For example, if the module imported above includes an exportdoAllTheAmazingThings(), you would call it like this:
myModule.doAllTheAmazingThings();
*(2) Import a single export from a module: Given an object or value named myExport which has been exported from the module my-module either implicitly, because the entire module is exported, or explicitly, using the export statement, this inserts myExport into the current scope.
import {myExport} from '/modules/my-module.js';
*(3) Import multiple exports from module: This inserts both foo and bar into the current scope.
import {foo, bar} from '/modules/my-module.js';
*(4) Import an export with a more convenient alias: You can rename an export when importing it. For example, this inserts shortName into the current scope.
import {reallyReallyLongModuleExportName as shortName} from '/modules/my-module.js';
*(5) Rename multiple exports during import: Import multiple exports from a module with convenient aliases.
```
import {
  reallyReallyLongModuleExportName as shortName,
  anotherLongModuleName as short
} from '/modules/my-module.js';
```
*(6) Import a module for its side effects only to run the module’s global code: Import an entire module for side effects only, without importing anything. This runs the module's global code, but doesn't actually import any values.
import '/modules/my-module.js';
*(7) Importing defaults: It is possible to have a default export, whether it is an object, a function, a class, etc.. The import statement may then be used to import such defaults.
The simplest version directly imports the default:
import myDefault from '/modules/my-module.js';
It is also possible to use the default syntax with the ones seen above, namespace imports or named imports. In such cases, the default import will have to be declared first. For instance:
import myDefault, * as myModule from '/modules/my-module.js';
//myModule used as a namespace
Or,
import myDefault, {foo, bar} from '/modules/my-module.js';
//specific, named imports

Using JavaScript Classes:
*The final part of the code defines a class that Angular can instantiate to create the component:
```
export class AppComponent
{
  model = new Model();
  getName()
  {
    return this.model.user;
  }
```

}
*These statements define a class called AppComponent that has a model property and a getName function, which provide the functionality required to support the data binding in the template in the code.
*When a new instance of the AppComponent class is created, the model property will be set to a new instance of the Model class defined earlier.
*The getName function returns the value of the user property defined by the Model object, which we had defined earlier with an: export class Model { //Omitted } with properties user and items.

The JavaScript Document Object Model (DOM) of a HTML Document: When the browser loads and processes an HTML document, it creates the Document Object Model (DOM).
*The DOM is a model in which JavaScript objects are used to represent each element in the document, and DOM is also the mechanism by which you can programmatically engage with the content of an HTML document.
*The browser maintains a live model of the HTML document represented by JavaScript objects.
*You rarely work directly with the DOM in Angular, but it is important to understand that when Angular modifies these objects, the browser updates the content it displays to reflect the modifications. This is one of the key foundations of web applications. If we were not able to modify the DOM, we would not be able to create client-side web apps.

Creating an Angular Application: We need three key pieces of functionality required to build a simple Angular application: a model, a template, and a component.
*Then we need to bring them together to create the application.
*Note: In any Angular project, there is a period where you have to define the main parts of the application and plumb them together. In more complex and realistic Angular application; there is a lot of initial setup and configuration required, but then the features start to quickly snap into place.

Creating a Data Model for the Angular Model-View-Controller (MVC) Architecture:
*Understanding the Models in MVC Pattern Web Applications: Models, the M in MVC, contain the data that users work with. The model contains logic that manages the persistence of data.
*There are two broad types of model:
*(1) view models, which represent just data passed from the component to the template, and
*(2) domain models, which contain the data in a business domain, along with operations, transformations, and rules for creating, storing, and manipulating that data, collectively referred to as the model logic.
*Many Angular models will effectively push the logic to the server side and invoke it via a RESTful web service because there is little support for data persistence within the browser and it is simply easier to get the data you require over Ajax. We explain how Angular can be used with RESTful web services later.
*Note: There are client-side persistence APIs defined as part of the HTML5 standard effort. The quality of these standards is currently mixed, and the implementations vary in quality. The main problem, however, is that most users still rely on browsers that don't implement the new APIs, and this is especially true in corporate environments, where older versions of Internet Explorer are still widely used because of problems migrating line-of-business applications to standard versions of HTML. If you are interested in client-side persistence, then the best place to start is the IndexedDB specification, which can be found at https://www.w3.org/TR/IndexedDB.
*The model in a web application built using the MVC pattern should:
*(1) Contain the domain data,
*(2) Contain the logic for creating, managing, and modifying the domain data, even if that means executing remote logic via web services,
*(3) Provide a clean API that exposes the model data and operations on it.
*Understanding the View Data in MVC Pattern Web Applications: The domain model isn't the only data in an Angular application. Components can create view data, also known as view model data or view models, to simplify templates and their interactions with the component.
*The benefits of ensuring that the model is isolated from the controller and views are that you can test your logic more easily, and that enhancing and maintaining the overall application is simpler and easier.
*The best domain models contain the logic for getting and storing data persistently and the logic for create, read, update, and delete operations (known collectively as CRUD) or separate models for querying and modifying data, known as the Command and Query Responsibility Segregation (CQRS) pattern.
*This can mean the model contains the logic directly, but more often the model will contain the logic for calling RESTful web services to invoke server-side database operations.

*When we created the static mock-up of the application, the data was distributed across all the HTML elements. The user's name is contained in the header, like this:
<h3 class="bg-primary p-a-1">Adam's To Do List</h3>
and the details of the to-do items are contained within td elements in the table, like this:
<tr><td>Buy Flowers</td><td>No</td></tr>
*The next task is to pull all the data together to create a data model. Separating the data from the way it is presented to the user is one of the key ideas in the MVC pattern.
*Angular applications are typically written in TypeScript. TypeScript is a superscript of JavaScript, but one of its main advantages is that it lets you write code using the latest JavaScript language specification with features that are not yet supported in all of the browsers that can run Angular applications.
*One of the packages added to the project in the previous section was the TypeScript compiler, which we set up to generate browser-friendly JavaScript files automatically when a change to a TypeScript file is detected.
*To create a data model for the application, we add a file called model.ts to the root application folder folder (note: TypeScript files have the .ts extension) and added the code shown below:

```
//File: model.ts:
var model =
{
  user: "Adam",
  items: [{ action: "Buy Flowers", done: false }, { action: "Get Shoes", done: false },
          { action: "Collect Tickets", done: true }, { action: "Call Joe", done: false }]
};
```

*One of the most important features of TypeScript is that you can just write "normal" JavaScript code as though you were targeting the browser directly.
*In the code, we used the JavaScript object literal syntax to assign a value to a global variable called model. The data model object has a user property that provides the name of the application's user and an items property, which is set to an array of objects with action and done properties, each of which represents a task in the to-do list.
*Note: Pay attention to the extension of the file. Although browsers only JavaScript features, it relies on the TypeScript compiler to convert them into code that will run in any browser. That means the .ts file must be used, which then allows the TypeScript compiler to create the corresponding .js file that will be used by the browser. It can be awkward if you are used to writing JavaScript files directly, but it allows the TypeScript compiler to translate the most recent features from the JavaScript specification into code that will run on older browsers.
*It can also be hard to correctly order the script elements. Browsers execute the content of JavaScript files in the order defined by the script elements in the HTML document, and it is easy to create a situation where code in the file loaded by one script element depends on functionality loaded by another script element that the browser has yet to load. Managing the JavaScript content in an application can be simplified by using a module loader, which takes responsibility for detecting and resolving dependencies between JavaScript files, loading them, and ensuring their contents are executed in the right order. The package.json file we created earlier included the SystemJS module loader, which can be applied to the HTML document to manage the JavaScript files.
*When you save the changes to the file, the TypeScript compiler will detect the change and generate a file called model.js, with the following contents:

```
//File: model.js:
var model =
{
  user: "Adam",
  items: [{ action: "Buy Flowers", done: false }, { action: "Get Shoes", done: false },
          { action: "Collect Tickets", done: true }, { action: "Call Joe", done: false }]
};
```

*This is the most important aspect of using TypeScript: you don't have to use the features it provides, and you can write entire Angular applications using just the JavaScript features that are supported by all browsers. But part of the value of TypeScript is that it converts code that uses the latest JavaScript language features into code that will run anywhere, even in browsers that don't support those features.
*The following listing shows the data model rewritten to use JavaScript features that were added in the ECMAScript 6 standard (known as ES6).

```
//File: Using ES6 features in the model.ts file:
export class Model
{
```

```
  user;
  items;
  constructor()
  {
    this.user = "Adam";
    this.items = [new TodoItem("Buy Flowers", false), new TodoItem("Get Shoes", false),
                  new TodoItem("Collect Tickets", false), new TodoItem("Call Joe", false)]
  }
}
export class TodoItem
{
  action;
  done;
  constructor(action, done)
  {
    this.action = action;
    this.done = done;
  }
}
```

*This is still standard JavaScript code, but the class keyword was introduced in a later version of the language than most web application developers are familiar with because it is not supported by older browsers. The class keyword is used to define types that can be instantiated with the new keyword to create objects that have well-defined data and behavior.

*Many of the features added in recent versions of the JavaScript language are syntactic sugar to help programmers avoid some of the most common JavaScript pitfalls, such as the unusual type system, and make it more familiar for programmers experienced in other languages, such as C# or Java.

*The class keyword doesn't change the way that JavaScript handles types.

*The export keyword relates to JavaScript modules. When using modules, each TypeScript or JavaScript file is considered to be a self-contained unit of functionality, and the export keyword is used to identity data or types that you want to use elsewhere in the application. JavaScript modules are used to manage the dependencies that arise between files in a project and avoid having to manually manage a complex set of script elements in the HTML file.

*The TypeScript compiler processes the code to generate JavaScript code that uses only the subset of features that are widely supported by browsers. Even though I used the class and export keywords, the model.js file generated by the TypeScript compiler produced JavaScript code that will work in browsers that don't implement that feature, like this:

```
"use strict";
var Model = (function ()
{
  function Model()
  {
    this.user = "Adam";
    this.items = [new TodoItem("Buy Flowers", false), new TodoItem("Get Shoes", false),
                  new TodoItem("Collect Tickets", false), new TodoItem("Call Joe", false)];
}
  return Model;
}());
exports.Model = Model;
var TodoItem = (function ()
{
  function TodoItem(action, done)
  {
    this.action = action;
    this.done = done;
  }
  return TodoItem;
```

```
}());
exports.TodoItem = TodoItem;
```
*We'll not be showing the code that the TypeScript compiler produces. The important point is that the compilation process translates new JavaScript features that are not widely supported by browsers into standard features that are supported.
*Using RESTful Web Services to do Server-Side CRUD Operations and Build the Models in MVC Pattern Web Applications: The logic for domain models in Angular apps is often split between the client and the server. The server contains the persistent store, typically a database, and contains the logic for managing it. In the case of a SQL database, for example, the required logic would include opening connections to the database server, executing SQL queries, and processing the results so they can be sent to the client.
*You don't want the client-side code accessing the data store directly; doing so would create a tight coupling between the client and the data store that would complicate unit testing and make it difficult to change the data store without also making changes to the client code as well.
*By using the server to mediate access to the data store, you prevent tight coupling. The logic on the client is responsible for getting the data to and from the server and is unaware of the details of how that data is stored or accessed behind the scenes.
*There are lots of ways of passing data between the client and the server. One of the most common is to use Asynchronous JavaScript and XML (Ajax) requests to call server-side code, getting the server to send JSON and making changes to data using HTML forms.
*This approach can work well and is the foundation of RESTful web services, which use the nature of HTTP requests to perform CRUD operations on data stored on the server side.

Creating a Angular HTML Template for Data Binding to a Data Model for the MVC Architecture:
*Angular HTML Template's job: Contains logic for displaying data to the user by manipulating the DOM based on user interaction.
*Angular relies on HTML element attributes to apply a lot of its functionality. Most of the time, the values of attributes are evaluated as JavaScript expressions.
*Understanding the Views/HTML Templates in MVC Pattern Web Applications: Views, which are known as templates in Angular, are defined using HTML elements that are enhanced by data bindings.
*It is the data bindings that make Angular so flexible, and they transform HTML elements into the foundation for dynamic web applications.
*We look at the different types of data bindings that Angular provides later.
*Templates should:
*(1) Contain the logic and markup required to present data to the user.
*Templates should not:
*(1) Contain complex logic. This is better placed in a component or one of the other Angular building blocks, such as directives, services, or pipes.
*(2) Contain logic that creates, stores, or manipulates the domain model. Templates can contain logic, but it should be simple and used sparingly. Putting anything but the simplest method calls or expressions in a template makes the overall application harder to test and maintain.
*We need a way to display the data values in the model to the user. In Angular, this is done using a template, which is a fragment of HTML that contains instructions that are performed by Angular.
*In an HTML file, the markup shown below:
```
<h3 class="bg-primary p-a-1">{{getName()}}'s To Do List</h3>
```
*We'll add more elements to this shortly, but a single h3 element is enough to get started.
*Including a data value in a template is done using double braces: {{ and }}, and Angular evaluates whatever you put between the double braces to get the value to display.
*The {{ and }} characters are an example of a data binding, which means that they create a relationship between the template and a data value. Data bindings are an important Angular feature.
*In this case, the data binding tells Angular to invoke a function called getName and use the result as the contents of the h3 element. The getName function doesn't exist in the application yet, but will be created in the next section by an Angular component that provides the data model to template for data binding.

Creating an Angular Component for Managing an Angular HTML Template by Providing it with a Data Model the Angular HTML Template can Data Bind to in the MVC Architecture:
*Understanding Controllers/Components in MVC Pattern Web Applications:

*Controllers, which are known as components in Angular, are the connective tissue in an Angular web app, acting as conduits between the data model and views.
*Components add business domain logic required to present some aspect of the model and perform operations on it.
*A component that follows the MVC pattern should:
*(1) Contain the logic required to set up the initial state of the template,
*(2) Contain the logic/behaviors required by the template to present data from the model, and
*(3) Contain the logic/behaviors required to update the model based on user interaction
*A component should not:
*(1) Contain logic that manipulates the DOM based on user interaction; that is the job of the template,
*(2) Contain logic that manages the persistence of data; that is the job of the model.
*An Angular component is responsible for managing a template providing it with data and logic it needs.
*Components are the parts of an Angular application that do most of the heavy lifting. As a consequence, they can be used for all sorts of tasks.
*At the moment, we have a data model that contains a user property with the name to display, and we have a template that displays the name by invoking a getName property.
*What we need is a component to act as the bridge between them.
*We now add a JavaScript file called app.component.ts, shown below, in the root application folder:

```
//File: app.component.ts:
import { Component } from "@angular/core";
import { Model } from "./model";
@Component(
{
  selector: "todo-app",
  templateUrl: "app/app.component.html"
})
export class AppComponent
{
  model = new Model();
  getName()
  {
    return this.model.user;
  }
}
```

*This is still JavaScript, but it relies on features that we may not have encountered before but that underpin Angular development. The code in the listing can be broken into three main sections, as described later.
***Using the @Component Decorator to Create an Angular Component for Managing an Angular HTML Template by Providing it with a Data Model the Angular HTML Template can Data Bind to in the MVC Architecture**:
*Ref: https://angular.io/api/core/Component
*The following part of above code is an example of a decorator, which provides metadata about a class:

```
@Component(
{
  selector: "todo-app",
  templateUrl: "app/app.component.html"
})
```

*This is the @Component decorator, and, as its name suggests, it tells Angular that this is an Angular component. The decorator provides configuration information through its properties, which in the case of @Component includes properties called selector and templateUrl.
*The selector property specifies a CSS selector that matches the HTML element to which the component will be applied: in this case, we've specified the todo-app element, which we added to index.html file as:

```
<body class="m-a-1">
  <todo-app>Angular placeholder</todo-app>
</body>
```

*When an Angular application starts, Angular scans the HTML in the current document and looks for elements that correspond to components. It will find the todo-app element and know that it should be placed under the control of this component.

*The templateUrl property is used to tell Angular how to find the component's template for data binding to a data model, which is the app.component.html file in the app folder for this component.
*There are also other properties that can be used with the @Component decorator and the other decorators that Angular supports.
*__The @Component Decorator__:
*Ref: https://angular.io/api/core/Component
*npm Package: @angular/core
*Module: import { Component } from '@angular/core';
*Source: core/src/metadata/directives.ts
*Marks a class as an Angular component and collects component configuration metadata.
*@Component metadata overview:

```
@Component(
{
  changeDetection?: ChangeDetectionStrategy
  viewProviders?: Provider[]
  moduleId?: string
  templateUrl?: string
  template?: string
  styleUrls?: string[]
  styles?: string[]
  animations?: any[]
  encapsulation?: ViewEncapsulation
  interpolation?: [string, string]
  entryComponents?: Array<Type<any>|any[]>
  preserveWhitespaces?: boolean
  //Inherited from core/Directive
  selector?: string
  inputs?: string[]
  outputs?: string[]
  host?: {[key: string]: string}
  providers?: Provider[]
  exportAs?: string
  queries?: {[key: string]: any}
})
```

*How to use:

```
@Component({selector: 'greet', template: 'Hello {{name}}!'})
class Greet
{
  name: string = 'World';
}
```

*Description: Component decorator allows you to mark a class as an Angular component and provide additional metadata that determines how the component should be processed, instantiated and used at runtime.
*Components are the most basic building block of an UI in an Angular application.
*An Angular application is a tree of Angular components. Angular components are a subset of directives.
*Unlike directives, components always have a template and only one component can be instantiated per an element in a template.
*A component must belong to an NgModule in order for it to be usable by another component or application. To specify that a component is a member of an NgModule, you should list it in the declarations field of that NgModule.
*In addition to the metadata configuration specified via the Component decorator, components can control their runtime behavior by implementing various Life-Cycle hooks.
*Component Metadata Properties:
*(1) animations?: any[]: List of animations of this component. Animations are defined on components via an animation-like DSL. This DSL approach to describing animations allows for a flexibility that both benefits developers and the framework. Animations work by listening on state changes that occur on an element within the template. When a state change occurs, Angular can then take advantage and animate the arc in between. This works similar to how CSS transitions work, however, by having a programmatic DSL, animations are not limited to

environments that are DOM-specific. For animations to be available for use, animation state changes are placed within animation triggers which are housed inside of the animations annotation metadata. Within a trigger both state and transition entries can be placed.

```
@Component(
{
  selector: 'animation-cmp',
  templateUrl: 'animation-cmp.html',
  animations: [
    //This here is our animation trigger that will contain our state change animations:
    trigger('myTriggerName', [
      //The styles defined for the `on` and `off`states declared below are persisted on the
      //element once the animation completes.
      state('on', style({ opacity: 1 }),
      state('off', style({ opacity: 0 }),
      //This here is our animation that kicks off when this state change jump is true.
      transition('on => off', [animate("1s")])
    ])
  ]
})
```

*As depicted in the code above, a group of related animation states are all contained within an animation trigger, the code example above called the trigger myTriggerName. When a trigger is created then it can be bound onto an element within the component's template via a property prefixed by an @ symbol followed by trigger name and an expression that is used to determine the state value for that trigger.

```
<!-- animation-cmp.html -->
<div @myTriggerName="expression">...</div>
```

*For state changes to be executed, the expression value must change value from its existing value to something that we have set an animation to animate on, in the example above we are listening to a change of state between on and off. The expression value attached to the trigger must be something that can be evaluated with the template/component context.

*DSL Animation Functions in Angular: Each of the animation DSL functions listed below are used for crafting animations: trigger(), state(), transition(), group(), sequence(), style(), animate(), and keyframes(). Each of these animation DSL functions are described below:

*(a) trigger(): npm Package: @angular/animations; Module: import { trigger } from '@angular/animations'; Source: core/src/animation/animation_metadata_wrapped.ts

```
function trigger(name: string, definitions: AnimationMetadata[]): AnimationTriggerMetadata;
```

*(b) state(): npm Package: @angular/animations; Module: import { state } from '@angular/animations'; Source: core/src/animation/animation_metadata_wrapped.ts

```
function state(name: string, styles: AnimationStyleMetadata): AnimationStateMetadata;
```

*(c) transition(): npm Package: @angular/animations; Module: import { transition } from '@angular/animations'; Source: core/src/animation/animation_metadata_wrapped.ts

```
function transition(stateChangeExpr: string, steps: AnimationMetadata | AnimationMetadata[]):
      AnimationTransitionMetadata;
```

*(d) group(): npm Package: @angular/animations; Module: import { group } from '@angular/animations'; Source: core/src/animation/animation_metadata_wrapped.ts

```
function group(steps: AnimationMetadata[]): AnimationGroupMetadata;
```

*(e) sequence(): npm Package: @angular/animations; Module: import { sequence } from '@angular/animations'; Source: core/src/animation/animation_metadata_wrapped.ts

```
function sequence(steps: AnimationMetadata[]): AnimationSequenceMetadata;
```

*(f) style(): npm Package: @angular/animations; Module: import { style } from '@angular/animations'; Source: core/src/animation/animation_metadata_wrapped.ts

```
function style(tokens: {[key: string]: string | number} |
      Array<{[key: string]: string | number}>): AnimationStyleMetadata;
```

*(g) animate(): npm Package: @angular/animations; Module: import { animate } from '@angular/animations'; Source: core/src/animation/animation_metadata_wrapped.ts

```
function animate(timings: string | number, styles?: AnimationStyleMetadata |
      AnimationKeyframesSequenceMetadata): AnimationAnimateMetadata;
```

*(h) keyframes(): npm Package: @angular/animations; Module: import { keyframes } from '@angular/animations'; Source: core/src/animation/animation_metadata_wrapped.ts
function keyframes(steps: AnimationStyleMetadata[]): AnimationKeyframesSequenceMetadata;
*(2) changeDetection?: ChangeDetectionStrategy: Defines the change detection strategy used by this component. When a component is instantiated, Angular creates a change detector, which is responsible for propagating the component's bindings. The changeDetection property defines, whether the change detection will be checked every time or only when the component tells it to do so.
*(3) encapsulation?: ViewEncapsulation: Specifies how the template and the styles should be used by this component. Values are:
*(a) ViewEncapsulation.Native: to use shadow roots - only works if natively available on the platform,
*(b) ViewEncapsulation.Emulated: to use shimmed CSS that emulates the native behavior,
*(c) ViewEncapsulation.None: to use global CSS without any encapsulation.
*When no encapsulation is defined for the component, the default value from the CompilerOptions is used. The default is ViewEncapsulation.Emulated. Provide a new CompilerOptions to override this value.
*If the encapsulation is set to ViewEncapsulation.Emulated and the component has no styles nor styleUrls the encapsulation will automatically be switched to ViewEncapsulation.None.
*(4) entryComponents?: Array<Type<any>|any[]>: List of components that are dynamically inserted into the view of this component. Defines components that should be compiled as well when this component is defined. For each components listed here, Angular will create a ComponentFactory and store it in the ComponentFactoryResolver.
*(5) preserveWhitespaces?: boolean: If Component.preserveWhitespaces is set to false potentially superfluous whitespace characters, ones matching the \s character class in JavaScript regular expressions, will be removed from a compiled template. This can greatly reduce AOT-generated code size as well as speed up view creation.
*Current implementation works according to the following rules:
*(a) all whitespaces at the beginning and the end of a template are removed (trimmed);
*(b) text nodes consisting of whitespaces only are removed (ex.: <button>Action 1</button> <button>Action 2</button> will be converted to <button>Action 1</button><button>Action 2</button> (no whitespaces between buttons),
*(c) series of whitespaces in text nodes are replaced with one space (ex.: \n some text\n will be converted to some text);
*(d) text nodes are left as-is inside HTML tags where whitespaces are significant (ex. <pre>, <textarea>).
*Described transformations can potentially influence DOM nodes layout so the preserveWhitespaces option is true be default, no whitespace removal. If you want to change the default setting for all components in your application you can use the preserveWhitespaces option of the AOT compiler.
*Even if you decide to opt-in for whitespace removal there are ways of preserving whitespaces in certain fragments of a template. You can force a space to be preserved in a text node by using the &ngsp; pseudo-entity. &ngsp; will be replaced with a space character by Angular's template compiler, ex.:
<a>Spaces&ngsp;<a>between&ngsp;<a>links.
will be compiled to the equivalent of:
<a>Spaces <a>between <a>links.
For alternate methods, see the @Component documentation.
*(6) exportAs - name under which the component instance is exported in a template.
*(7) host - map of class property to host element bindings for events, properties and attributes.
*(8) inputs - list of class property names to data-bind as component inputs.
*(9) interpolation?: [string, string]: Custom interpolation markers used in this component's template. Overrides the default encapsulation start and end delimiters (respectively {{ and }})
*(10) moduleId?: string: The module id of the module (file) that contains the component. Needed to be able to resolve relative urls for templates and styles. In CommonJS, this can always be set to module.id, similarly SystemJS exposes __moduleName variable within each module.
*Example
```
@Directive(
{
  selector: 'someDir',
  moduleId: module.id
})
class SomeDir
{
```

}
*(11) outputs - list of class property names that expose output events that others can subscribe to.
*(12) providers - list of providers available to this component and its children.
*(13) queries - configure queries that can be injected into the component.
*(14) selector - css selector that identifies this component in a template.
*(15) styleUrls?: string[]: List of stylesheet urls to be applied to this Angular component's view.
*(16) styles?: string[]: List of inline-defined stylesheets to be applied to this Angular component's view.
*(17) template?: string: Specifies an inline template for an Angular component. Only one of templateUrl or template can be defined per component.
*(18) templateUrl?: string: url to an external Angular component file containing a template for the view. Only one of templateUrl or template can be defined per view.
*(19) viewProviders?: Provider[]: Defines the list of injectable providers objects that are available to this component and its view DOM children.
*Example: Here is an example of a class that can be injected:

```
class Greeter
{
  greet(name:string)
  {
    return 'Hello ' + name + '!';
  }
} //class Greeter
@Directive(
{
  selector: 'needs-greeter'
})
class NeedsGreeter
{
  greeter:Greeter;
  constructor(greeter:Greeter)
  {
    this.greeter = greeter;
  }
}
@Component(
{
  selector: 'greet',
  viewProviders: [Greeter],
  template: `<needs-greeter></needs-greeter>`
})
class HelloWorld
{
}
```

Putting the Pieces together to Create an Angular Application:
*We have the three key pieces of functionality required to build a simple Angular application:
a model, a template, and a component.
*Now we need to bring them together to create the application.
*The first step is to create an Angular module. Through an unfortunate naming choice, there are
two types of module used in Angular development.
*A JavaScript module is a file that contains JavaScript functionality that is used through the import keyword.
*The other type of module is an Angular module, which is used to describe an application or a group of related features.
*Every application has a root module, which provides Angular with the information that it needs to start the application. We created a file called app.module.ts, which is the conventional file name for the root module, in the root application folder and added the code shown below:
//File: app.module.ts:

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { AppComponent } from "./app.component";
@NgModule(
{
  imports: [BrowserModule, FormsModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

*The purpose of the Angular module is to provide configuration information through the properties defined by the @NgModule decorator. The decorator's imports property tells Angular that the application depends on features required to run an application in the browser and that the declarations and bootstrap properties tell Angular about the components in the application and which one should be used to start the application. There is only one component in this simple example application, which is why it is the only value for both properties.

*Angular applications need a bootstrap file, which contains codes required to start the application and load the Angular module. To create the bootstrap file, we added a file called main.ts to the root application folder with the code shown below:

```
//File: main.ts:
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
platformBrowserDynamic().bootstrapModule(AppModule);
```

*Although we focuses on applications that run in a web browser, Angular is intended to work in a range of environments. The code statements in the bootstrap file select the platform that will be used and load the root module, which is the entry point into the application.

*Note: Calling platformBrowserDynamic().bootstrapModule method is for browser-based applications, which is what our focus is. If you are working on different platforms, then you will have to use a different bootstrap method specific to the platform you are working with. The developers of each platform that supports Angular provide details of their platform-specific bootstrap method.

*The final step is to update the HTML file so that the browser will load the application and the modules that contain the Angular framework functionality, as shown below:

```
//File: index.html: Completing the HTML document in the index.html file:
<!DOCTYPE html>
<html>
<head>
  <title>ToDo</title>
  <meta charset="utf-8" />
  <link href="node_modules/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
  <script src="node_modules/classlist.js/classList.min.js"></script>
  <script src="node_modules/core-js/client/shim.min.js"></script>
  <script src="node_modules/zone.js/dist/zone.min.js"></script>
  <script src="node_modules/reflect-metadata/Reflect.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>
  <script>
    var paths =
    {
      "rxjs/*": "node_modules/rxjs/bundles/Rx.min.js",
      "@angular/*": "node_modules/@angular/*"
    }
    var packages = { "app": {} };
    var angularModules = ["common", "compiler", "core", "forms",
                          "platform-browser", "platform-browser-dynamic"];
    angularModules.forEach(function (pkg)
    {
```

```
      packages["@angular/" + pkg] =
      {
        main: "/bundles/" + pkg + ".umd.min.js"
      };
    });
    System.config({ paths: paths, packages: packages });
    System.import("app/main").catch(function(err){ console.error(err); });
  </script>
</head>
<body class="m-a-1">
  <todo-app>Angular placeholder</todo-app>
</body>
</html>
```

*When working with JavaScript modules, you don't add script elements for all the files that are required by the application. Instead, a module loader is provided with the application's bootstrap file and with details of how to load the modules that are specified by the import statements in the code.

*The module loader ensures that all the required modules are loaded and processed before the JavaScript code they contain is executed. The script element in the listing configures the module loader that we use in here, called SystemJS module loader, so that it knows how to find the JavaScript modules that contain the Angular functionality and the custom files in the project. The Angular framework and the example application code will be loaded when the browser processes the HTML file.

*__**The @NgModule Decorator**__:

*Ref: https://angular.io/api/core/NgModule
*npm Package: @angular/core
*Module: import { NgModule } from '@angular/core';
*Source: core/src/metadata/ng_module.ts
*NgModule metadata overview:

```
@NgModule(
{
  providers?: Provider[]
  declarations?: Array<Type<any>|any[]>
  imports?: Array<Type<any>|ModuleWithProviders|any[]>
  exports?: Array<Type<any>|any[]>
  entryComponents?: Array<Type<any>|any[]>
  bootstrap?: Array<Type<any>|any[]>
  schemas?: Array<SchemaMetadata|any[]>
  id?: string
})
```

*Description: NgModule decorator and metadata.
*NgModule metadata properties:
*(1) providers?: Provider[]: The set of injectable objects that are available in the injector of this module.
*Example: Here is an example of a class that can be injected:

```
class Greeter
{
  greet(name:string)
  {
    return 'Hello ' + name + '!';
  }
}
@NgModule(
{
  providers: [Greeter]
})
class HelloWorld
{
```

```
  greeter:Greeter;
 constructor(greeter:Greeter)
  {
    this.greeter = greeter;
  }
}
```
*(2) declarations?: Array<Type<any>|any[]>: Specifies a list of directives/pipes that belong to this module.
*Example:
```
@NgModule(
{
  declarations: [NgFor]
})
class CommonModule
{
}
```
*(3) imports?: Array<Type<any>|ModuleWithProviders|any[]>: Specifies a list of modules whose exported directives/pipes should be available to templates in this module. This can also contain ModuleWithProviders.
*Example:
```
@NgModule(
{
  imports: [CommonModule]
})
class MainModule
{
}
```
*(4) exports?: Array<Type<any>|any[]>: Specifies a list of directives/pipes/modules that can be used within the template of any component that is part of an Angular module that imports this Angular module.
*Example:
```
@NgModule({
 exports: [NgFor]
})
class CommonModule
{
}
```
*(5) entryComponents?: Array<Type<any>|any[]>: Specifies a list of components that should be compiled when this module is defined. For each component listed here, Angular will create a ComponentFactory and store it in the ComponentFactoryResolver.
*(6) bootstrap?: Array<Type<any>|any[]>: Defines the components that should be bootstrapped when this module is bootstrapped. The components listed here will automatically be added to entryComponents.
*(7) schemas?: Array<SchemaMetadata|any[]>: Elements and properties that are not Angular components nor directives have to be declared in the schema. Available schemas:
•NO_ERRORS_SCHEMA: any elements and properties are allowed,
•CUSTOM_ELEMENTS_SCHEMA: any custom elements (tag name has "-") with any properties are allowed.
*(8) id?: string: An opaque ID for this module, e.g. a name or a path. Used to identify modules in getModuleFactory. If left undefined, the NgModule will not be registered with getModuleFactory.

Running the Angular Application: As soon as you save the changes to the index.html file, the browser should reload the page, and your first Angular application will spring to life.
*The browser executed the code in the bootstrap file, which fired up Angular, which in turn processed the HTML document and discovered the todo-app element. The selector property used to define the component matches the todo-app element, which allowed Angular to remove the placeholder content and replace it with the component's template, which was loaded automatically from app.component.html file.
*The template was parsed; the {{ and }} data binding was discovered, and the expression it contains was evaluated, calling the getName and displaying the result.

Adding Features to the Angular Application: Now that the basic structure of the application is in place, we can add the remaining features that we had mocked up with data embedded to static HTML earlier.

Adding a To-Do Table to the Angular Application: In the sections that follow, we add the table containing the list of to-do items and the input element and button for creating new items.
*Angular HTML templates can do more than just display simple data values. We describe the full range of template features later.
*For this application, we are going to use the feature that allows a set of HTML elements to be added to the DOM for each object in an array. The array in this case is the set of to-do items in the data model.
*To begin, we add a method to the component that provides the template with the array of to-do items.
//Adding a Method in the app.component.ts file:

```
import { Component } from "@angular/core";
import { Model } from "./model";
@Component(
{
  selector: "todo-app",
  templateUrl: "app/app.component.html"
})
export class AppComponent
{
  model = new Model();
  getName()
  {
    return this.model.user;
  }
  getTodoItems()
  {
    return this.model.items;
  }
}
```

*The getTodoItems method returns the value of the items property from the Model object.
*The following updates the component's template to take advantage of the new method.
//Displaying the To-Do Items in the app.component.html file:

```
<h3 class="bg-primary p-a-1">{{getName()}}'s To Do List</h3>
<table class="table table-striped table-bordered">
  <thead>
    <tr><th></th><th>Description</th><th>Done</th></tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of getTodoItems(); let i = index">
      <td>{{ i + 1 }}</td>
      <td>{{ item.action }}</td>
      <td [ngSwitch]="item.done">
        <span *ngSwitchCase="true">Yes</span>
        <span *ngSwitchDefault>No</span>
      </td>
    </tr>
  </tbody>
</table>
```

*The additions to the template rely on several different Angular features.
*The first is the *ngFor expression, which is used to repeat a region of content for each item in an array. This is an example of a directive, which is described later.
*The *ngFor expression is applied to an attribute of an element, like this:

```
<tr *ngFor="let item of getTodoItems(); let i = index">
```

*This expression tells Angular to treat the tr element to which it has been applied as a template that should be repeated for every object returned by the component's getTodoItems method.

*The let item part of the expression specifies that each object should be assigned to a variable called item, so that it can be referred to within the template.
*The ngFor expression also keeps track of the index of the current object in the array that is being processed, and this is assigned to a second variable called i:
<tr *ngFor="let item of getTodoItems(); let i = index">
*The result is that the tr element and its contents will be duplicated and inserted into the HTML document for each object returned by the getTodoItems method; for each iteration, the current to-do object can be accessed through the variable called item, and the position of the object in the array can be accessed through the variable called i.
*Within the tr template, there are two data bindings, which can be recognized by the {{ and }} characters, as follows:
<td>{{ i + 1 }}</td>
<td>{{ item.action }}</td>
*These bindings refer to the variables that are created by the *ngFor expression.
*Note: For simple transformations, you can embed your JavaScript expressions directly in bindings like this, but for more complex operations, Angular has a feature called pipes, described later.
*The remaining template expressions in the tr template demonstrate how content can be generated selectively.
<td [ngSwitch]="item.done">
  <span *ngSwitchCase="true">Yes
  <span *ngSwitchDefault>No
</td>
*The [ngSwitch] expression is a conditional statement that is used to insert different sets of elements into the document based on a specified value, which is the item.done property in this case. Nested within the td element are two span elements that have been annotated with *ngSwitchCase and *ngSwitchDefault and that are equivalent to the case and default keywords of a regular JavaScript switch block.
*We describe ngSwitch later, but the result is that the first span element is added to the document when the value of the item.done property is true and the second span element is added to the document when item.done is false. The result is that the true/false value of the item.done property is transformed into span elements containing either Yes or No.
*When you save changes to template, the browser will reload, and table of to-do items will be displayed.
*If you use the browser's F12 developer tools (so called because they are typically opened by pressing the F12 key) and look at the JavaScript console that shows the effect JavaScript calls, you will be able to see the HTML content that the template has generated. You can't do this looking at the page source, which just shows the HTML sent by server and not the changes made by Angular using the DOM API.
*You can see how each to-do object in the model has produced a row in the table that is populated with the local item and i variables and how the switch expression shows Yes or No to indicate whether the task has been completed.
<tr>
  <td>2</td>
  <td>Get Shoes</td>
  <td>No</td>
</tr>
<tr>
  <td>3</td>
  <td>Collect Tickets</td>
  <td>Yes</td>
</tr>

XXXXXStart with Creating a Two-Way Data Binding, p.27.

XXXXXEnd

Cascading Style Sheets (CSS) and the Bootstrap CSS Class Library Framework:
*HTML elements tell the browser what kind of content they represent, but they don't provide any information about how that content should be displayed.

*The information about how to display elements is provided using Cascading Style Sheets (CSS). CSS consists of a comprehensive set of properties that can be used to configure every aspect of an element's appearance and a set of selectors that allow those properties to be applied.
*One of the main problems with CSS is that some browsers interpret properties slightly differently, which can lead to variations in the way that HTML content is displayed on different devices. It can be difficult to track down and correct these problems, and CSS frameworks have emerged to help web app developers style their HTML content in a simple and consistent way.
*One CSS framework that has gained a lot of popularity is Bootstrap, which was originally developed at Twitter but has become a widely used open source project.
*Bootstrap consists of a set of CSS classes that can be applied to elements to style them consistently and some JavaScript code that performs additional enhancement.
*Bootstrap works well across browsers, and it is simple to use. Bootstrap provides a lot more features than the ones used here; see http://getbootstrap.com for full details.
***Applying Basic Bootstrap Classes via the HTML class Attribute**: Bootstrap styles are applied via the HTML class attribute, which is used to group together related elements.
*The class attribute isn't just used to apply CSS styles, but it is the most common use, and it underpins the way that Bootstrap and similar frameworks operate.
*Here is an HTML element with a class attribute, taken from index.html file:
<button class="btn btn-primary m-t-1">Add</button>
*The class attribute assigns the button element to three classes, whose names are separated by spaces: btn, btn-primary, and m-t-1.
*These classes correspond to collections of styles defined by Bootstrap, as described below:

Name	Description
btn	This class applies the basic styling for a button. It can be applied to button or a elements to provide a consistent appearance.
btn-primary	This class applies a style context to provide a visual cue about the purpose of the button.
m-t-1	This class adds a gap between the top of the element and the content that surrounds it.

Table: The three button element classes.

XXXXXStart Using Contextual Classes, p.52.

XXXXXXXEnd

XXXXXXXStart

XXXXXXXEnd

XXXXXXXXXXXXXEnd