

α

asian tech *inc.*

Web Front-End Training Book

Version 1.0

SASS HTML5 **AJAX** PSD HTML5 **IMG** NPM XHTML **SASS** Grid **HTML**
illustrator class PSD AJAX Webpack attribute class Internet-Explorer iphone
element Internet-Explorer IE6 element Mozilla Internet-Explorer Webpack Bootstrap
Grid iphone **Canvas** NPM **PSD**
element PSD **PSD** Google Internet-Explorer **PSD**
XHTML class **PSD** element Bootstrap element **PSD** img NPM
easein **Frontend** Mozilla
Firefox **AJAX**
mootools font **Bootstrap** XHTML **PSD** Adaptive CSS3 Adaptive **AJAX**
SASS Mozilla **PSD** Google Bower Canvas **mootools** font **BOWER** Canvas **SASS**
Canvas Adaptive **PSD** SASS Adaptive **PSD** Mozilla **PSD** **PSD**
easeout **Frontend** asiantech **PSD** Canvas Adaptive **PSD** Canvas **PSD**
CSS3 Yeoman **PSD** font **PSD** **PSD** element **PSD**
frontend **Webpack** **Flat** **PSD** **PSD** mootools **Adaptive** element **PSD**
backbone **Frontend** **PSD** **PSD** **PSD** **PSD** mootools **Adaptive** element **PSD**
Firefox Foundation **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
Bower Grid **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
Flat NPM **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
illustrator **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
SASS **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
HTML **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
NodeJS **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
Linux **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
img **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
HTML **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
illustrator **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
SASS **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
NodeJS **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
HTML **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
Linux **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
XML **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
IE6 **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
jQuery **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
ReactJS **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
internet-Explorer **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
class **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
LESS **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
asiantech **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
JSON **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
IE6 **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
CSS3 **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
HTML **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
Grunt **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
asiantech **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
attribute **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
JSON **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**
Linux **PSD** **PSD** **PSD** **PSD** **PSD** **PSD** **PSD**

Table of Contents

Introduction	1.1
Chapter I: Basic	1.2
Section 0: Developer Environment	1.2.1
Environment	1.2.1.1
Editors and IDEs	1.2.1.2
Toolings	1.2.1.3
Section 1: HTML	1.2.2
Structure	1.2.2.1
Template	1.2.2.2
Template Engines	1.2.2.3
Conventions	1.2.2.4
Section 2: CSS	1.2.3
Selectors	1.2.3.1
Properties	1.2.3.2
Functions	1.2.3.3
@ Rules	1.2.3.4
Media queries	1.2.3.5
Preprocessors	1.2.3.6
Sass (SCSS)	1.2.3.6.1
Variables	1.2.3.6.1.1
Nesting	1.2.3.6.1.2
Partials	1.2.3.6.1.3
Mixins	1.2.3.6.1.4
Inheritance	1.2.3.6.1.5
Operators	1.2.3.6.1.6
Example	1.2.3.6.1.7
Conventions	1.2.3.7
Section 3: JavaScript	1.2.4
Grammar and types	1.2.4.1
Statements and Declaration	1.2.4.2

Expressions and Operators	1.2.4.3
Functions	1.2.4.4
Prototyping	1.2.4.5
Built-in Objects	1.2.4.6
Conventions	1.2.4.7
Section 5: Front-end Frameworks	1.2.5
Chapter II: Web Frameworks	1.3
Section 3: Angular 2	1.3.1
Introduction	1.3.1.1
Prerequisite	1.3.1.2
ECMAScript 6 and TypeScript Features	1.3.1.2.1
ECMAScript 6	1.3.1.2.1.1
Classes	1.3.1.2.1.1.1
Refresher on this	1.3.1.2.1.1.2
Arrow Functions	1.3.1.2.1.1.3
Template Strings	1.3.1.2.1.1.4
Inheritance	1.3.1.2.1.1.5
Constants and Block Scoped Variables	1.3.1.2.1.1.6
...spread and ...rest	1.3.1.2.1.1.7
Destructuring	1.3.1.2.1.1.8
Modules	1.3.1.2.1.1.9
TypeScript	1.3.1.2.1.2
Getting Started	1.3.1.2.1.2.1
Working With tsc	1.3.1.2.1.2.2
Typings	1.3.1.2.1.2.3
Linting	1.3.1.2.1.2.4
TypeScript Features	1.3.1.2.1.2.5
TypeScript Classes	1.3.1.2.1.2.6
Interfaces	1.3.1.2.1.2.7
Shapes	1.3.1.2.1.2.8
Type Inference	1.3.1.2.1.2.9
Decorators	1.3.1.2.1.2.10
Property Decorators	1.3.1.2.1.2.10.1
Class Decorators	1.3.1.2.1.2.10.2

Parameter Decorators	1.3.1.2.1.2.10.3
The JavaScript Toolchain	1.3.1.2.2
Source Control: git	1.3.1.2.2.1
The Command Line	1.3.1.2.2.2
Command Line JavaScript: NodeJS	1.3.1.2.2.2.1
Back-End Code Sharing and Distribution: npm	1.3.1.2.2.3
Module Loading, Bundling and Build Tasks	1.3.1.2.2.4
Bootstrapping	1.3.1.3
Understanding the File Structure	1.3.1.3.1
Bootstrapping Providers	1.3.1.3.2
Components	1.3.1.4
Creating Components	1.3.1.4.1
Application Structure with Components	1.3.1.4.2
Passing Data into a Component	1.3.1.4.2.1
Responding to Component Events	1.3.1.4.2.2
Using Two-Way Data Binding	1.3.1.4.2.3
Accessing Child Components from Template	1.3.1.4.2.4
Projection	1.3.1.4.3
Structuring Applications with Components	1.3.1.4.4
Using Other Components	1.3.1.4.5
Advanced	1.3.1.4.6
Component Lifecycle	1.3.1.4.6.1
Accessing Other Components	1.3.1.4.6.2
View Encapsulation	1.3.1.4.6.3
ElementRef	1.3.1.4.6.4
Directives	1.3.1.5
Attribute Directives	1.3.1.5.1
NgStyle Directives	1.3.1.5.1.1
NgClass Directives	1.3.1.5.1.2
Structural Directives	1.3.1.5.2
NgIf Directives	1.3.1.5.2.1
NgFor Directives	1.3.1.5.2.2
NgSwitch Directives	1.3.1.5.2.3

Combining Directives	1.3.1.5.2.4
Advanced	1.3.1.5.3
Creating an Attribute Directive	1.3.1.5.3.1
Listening to an Element Host	1.3.1.5.3.1.1
Setting Properties in a Directive	1.3.1.5.3.1.2
Creating a Structural Directive	1.3.1.5.3.2
View Containers and Embedded Views	1.3.1.5.3.2.1
Providing Context Variables to Directives	1.3.1.5.3.2.2
Pipes	1.3.1.6
Using Pipes	1.3.1.6.1
Custom Pipes	1.3.1.6.2
Stateful Pipes	1.3.1.6.3
Services	1.3.1.7
Basic	1.3.1.7.1
Observables	1.3.1.7.2
Using Observables	1.3.1.7.2.1
Error Handling	1.3.1.7.2.2
Disposing Subscriptions and Releasing Resources	1.3.1.7.2.3
Observables vs Promises	1.3.1.7.2.4
Using Observables From Other Sources	1.3.1.7.2.5
Observables Array Operations	1.3.1.7.2.6
Combining Streams with flatMap	1.3.1.7.2.7
Cold vs Hot Observables	1.3.1.7.2.8
Summary	1.3.1.7.2.9
Forms	1.3.1.8
Getting Started	1.3.1.8.1
Template-Driven Forms	1.3.1.8.2
Nesting Form Data	1.3.1.8.2.1
Using Template Model Binding	1.3.1.8.2.2
Validating Template-Driven Forms	1.3.1.8.2.3
Model-Driven Forms	1.3.1.8.3
FormBuilder	1.3.1.8.3.1
Validating Model-Driven Forms	1.3.1.8.3.2
FormBuilder Custom Validation	1.3.1.8.3.3

Visual Cues for Users	1.3.1.8.4
Routing	1.3.1.9
Why Routing?	1.3.1.9.1
Configuring Routes	1.3.1.9.2
Redirecting the Router to Another Route	1.3.1.9.3
Defining Links Between Routes	1.3.1.9.4
Dynamically Adding Route Components	1.3.1.9.5
Using Route Parameters	1.3.1.9.6
Defining Child Routes	1.3.1.9.7
Controlling Access to or from a Route	1.3.1.9.8
Passing Optional Parameters to a Route	1.3.1.9.9
Using Auxiliary Routes	1.3.1.9.10
Change Detection	1.3.1.10
How Change Detection Works	1.3.1.10.1
Change Detector Classes	1.3.1.10.2
Change Detection Strategy: OnPush	1.3.1.10.2.1
Enforcing Immutability	1.3.1.10.3
Additional Resources	1.3.1.10.4
Dependency Injection	1.3.1.11
What is DI?	1.3.1.11.1
DI Framework	1.3.1.11.2
Angular 2's DI	1.3.1.11.3
@Inject() and @Injectable	1.3.1.11.3.1
Injection Beyond Classes	1.3.1.11.3.2
The Injector Tree	1.3.1.11.3.3
Modules	1.3.1.12
What is an Angular 2 Module?	1.3.1.12.1
Adding Components, Pipes and Services to a Module	1.3.1.12.2
Creating a Feature Module	1.3.1.12.3
Directive Duplications	1.3.1.12.4
Lazy Loading a Module	1.3.1.12.5
Lazy Loading and the Dependency Injection Tree	1.3.1.12.6
Shared Modules and Dependency Injection	1.3.1.12.7

Sharing the Same Dependency Injection Tree	1.3.1.12.8
Project Setup	1.3.1.13
Webpack	1.3.1.13.1
Installation	1.3.1.13.1.1
Loaders	1.3.1.13.1.2
Plugins	1.3.1.13.1.3
Summary	1.3.1.13.1.4
NPM Scripts Integration	1.3.1.13.2
Angular CLI	1.3.1.13.3
Setup	1.3.1.13.3.1
Creating a New App	1.3.1.13.3.2
Serving the App	1.3.1.13.3.3
Creating Components	1.3.1.13.3.4
Creating Routes	1.3.1.13.3.5
Creating Other Things	1.3.1.13.3.6
Testing	1.3.1.13.3.7
Linting	1.3.1.13.3.8
CLI Command Overview	1.3.1.13.3.9
Adding Third Party Libraries	1.3.1.13.3.10
Integrating an Existing App	1.3.1.13.3.11
Deploying	1.3.1.14
Glossary	1.3.1.15
Other Resources	1.3.1.16

Introduction

Table of Content

- Who we are
- Why we create this book
- More about Asian Tech's Ruby and Front End Training Program
- After reading this book

Who we are

Asian Tech Co., Ltd.

We at Asian Tech want to be at the forefront of the IT revolution in Vietnam. Our engineers work not only for professional and technological development, but also aspire to be part of an internationally expanding economy that is shaping the future of their country.

At our current speed of growth, we see ourselves to be the biggest and most successful company in Vietnam by 2020. But for us, it is about more than just numbers. We want to create a dynamic environment where all our employees have the opportunity to expand their horizons globally and to use their experience at Asian Tech to shape their career path, wherever it may lead.

Why we create this book

This book is made public as part of the Asian Tech Ruby and Front End team's effort to:

- Deliver an organized and free guideline for everyone in the community who is currently and will be doing web development in the future. As the web-development space can easily outgrowth developer's ability to grasp, content in this book has been carefully placed in the most logical and systematic that we could think of.
- Deliver Study material for our training regime here at Asian Tech. If you find this book helpful, feel free to pass it around however please note that what you see here does not include our Teaching material.

More about Asian Tech's Ruby and Front End Training Program

Targeted Audience

Our training courses target to: **Interns** and **Associate Software Engineer**

Definitions for Interns and Associate Software Engineer at AT:

Audience	Definition
Interns	Interns are usually university students, or university graduates who have not yet found employment. Interns are less frequently college students (under 18) or older "career changers".
Associate Software Engineers	Just graduated engineer with major in IT who has less or no working experience. The employee is able to understand and perform work assignments with limited guidance and support from supervisor or manager. The engineer will participate in coding, scripting and executing unit test, and fix bugs for assigned project.

After reading this book

After finishing the content in this book, if you feel curious about us and want to find out about opportunities to work here at Asian Tech feel free to drop by our Career page at <http://asiantech.vn/en/careers>. The back-end language in this book does target Ruby. However if you are not familiar with Ruby, you can still benefit from this book because the Front-end chapter has been compiled very extensively.



What is a Web Developer?

A web developer is a programmer who specializes in, or is specifically engaged in, the development of World Wide Web applications, or distributed network applications that are run over HTTP from a web server to a web browser.

Learning Web Development

Whether you are a complete beginner or not, web development can be challenging — we will hold your hand and provide enough detail for you to feel comfortable and learn the topics properly.

Web development can be divided into three parts:

- *Client-side scripting*, which is code that executes in a web browser and determines what your customers or clients will see when they land on your website.
- *Server-side scripting*, which is code that executes on a web server and powers the behind-the-scenes mechanics of how a website works.
- *Database technology*, which also helps keep a website running smoothly.

Throughout the hard learning time, you may find yourself fit into:

- Front End Development
- Back End Development

But at the beginning of the journey, we encourage and challenge you to become **One Web Developer** by all means!

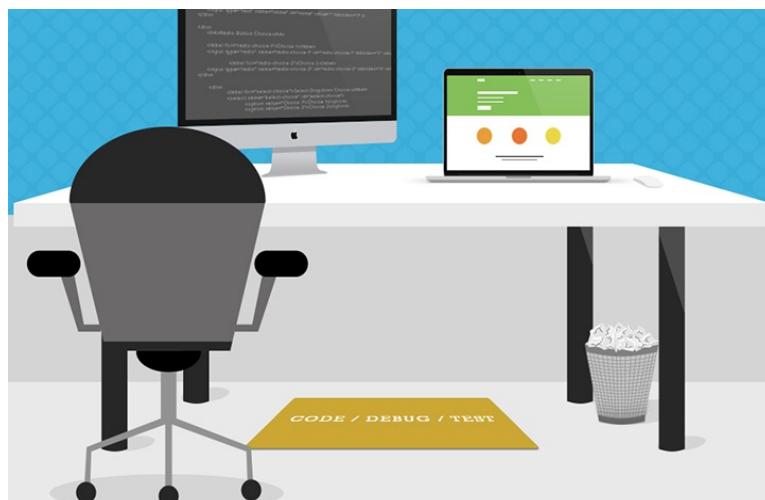
Chapter I: Into Front End Development

Table of Content

- [What is a Front-End Developer?](#)
- [Learning Front-End Development](#)

What is a Front End Developer?

A front-end developer architects and develops websites and applications using web technologies (i.e., HTML, CSS, DOM, and JavaScript), which run on the web platform or act as compilation input for non-web platform environments (i.e., NativeScript).



Learning Front End Development

To embrace modern front-end technology, we need to accept that learning is a part of our job. I am not saying you need to learn everything you're read on the Internet, but if you're interested an open-minded, it's a great start.

Between the lines in a lot of the modern front-end critique I sense a resistance. A resistance to learn and willingness to understand. This attitude will hold you back and leave you behind. The front-end is on a fast track, at it will not wait for the doubters. To jump on it can be frightening, but you will not doubt it. There is amazing thing going on and you can't afford to let fear hold you back.

THE FRONT-END SPECTRUM



Section 5: Developer Environment

Table of Content

1. [Environment](#)
2. [Editors/IDEs](#)
3. [Toolings](#)

Environment

Table of Content

- [What is Environment?](#)
- [Browsers](#)
 - [WebView](#)
- [Runtime: Node.js](#)

What is Environment?

A front-end developer crafts HTML, CSS, and JS that typically runs on the web platform (e.g. a web browser) on one of the following operating systems (aka OSs):

- Windows
- macOS
- Linux (distributed flavors, eg. Ubuntu,...)
- Android
- iOS
- ...

These operating systems typically run on one or more of the following devices:

- Desktop computer
- Laptop / Netbook computer
- Mobile phone
- Tablet
- TV
- Watch
- Things (i.e., anything you can imagine, car, refrigerator, lights, thermostat, etc.)

Generally speaking, front-end technologies can run on the aforementioned operating systems and devices using the following run time scenarios:

- A web browser (examples: Chrome, IE, Safari, Firefox) running on an OS.
- A headless browser (examples phantomJS) driven from a CLI running on an OS.
- A WebView/browser tab (think iframe) embedded within a native application as a runtime with bridge to native APIs. WebView applications typically contain a UI

constructed from web technologies (HTML, CSS, and JS) (examples: Apache Cordova, NW.js, Electron).

- A native application built from web tech that is interpreted at runtime with a bridge to native APIs. The UI will make use of native UI parts (e.g., iOS native controls) not web technologies (examples: NativeScript, React Native).

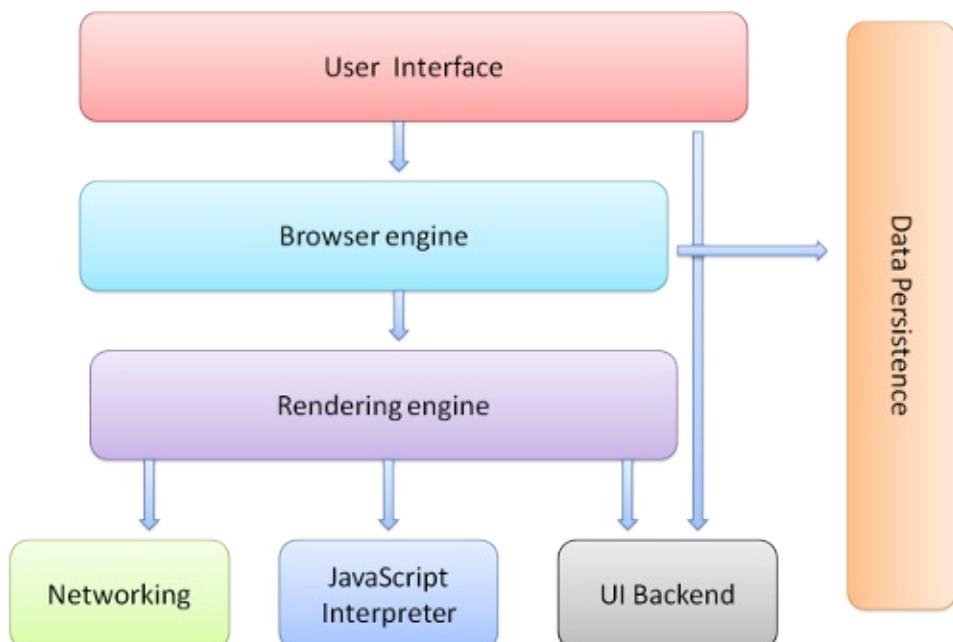
Browsers

The most commonly used browsers (on any device) are:

- Chrome (engine: Blink + V8)
- Firefox (engine: Gecko + SpiderMonkey)
- Internet Explorer/Microsoft Edge (engine: Trident/EdgeHTML + Chakra)
- Safari (engine: Webkit + JavaScriptCore)

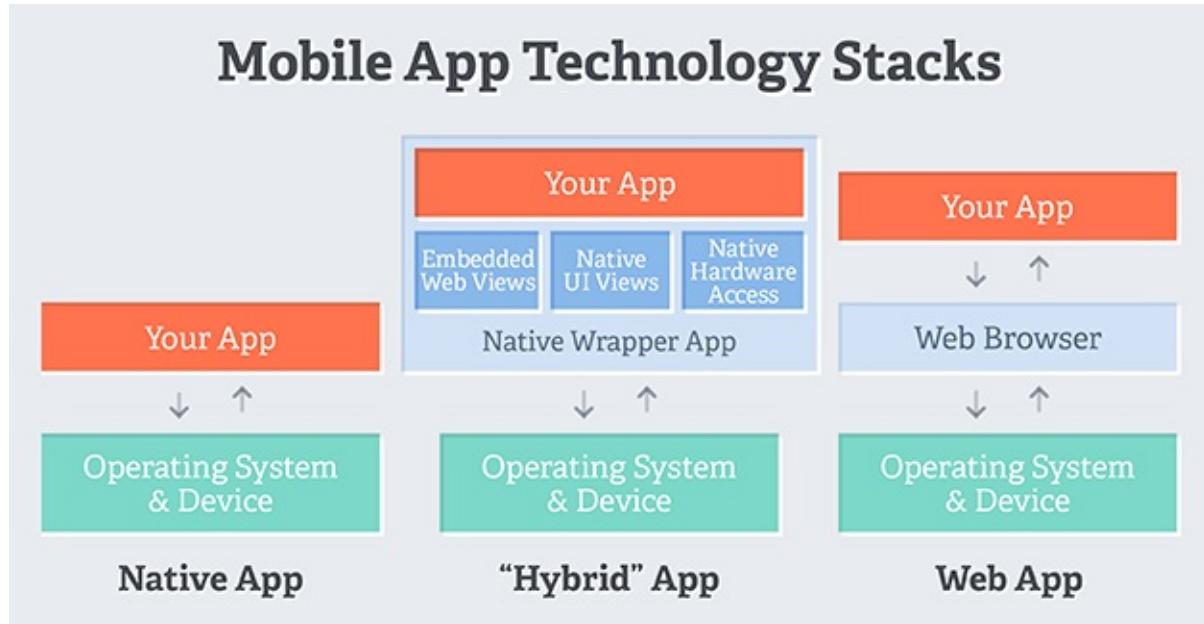
Engine contains two parts:

- A web browser engine (sometimes called layout engine or rendering engine) is a program that renders marked up content (such as HTML, XML, image files, etc.) and formatting information (such as CSS, XSL, etc.).
- A JavaScript engine is a program or library which executes JavaScript code. A JavaScript engine may be a traditional interpreter, or it may utilize just-in-time compilation to bytecode in some manner. Although there are several uses for a JavaScript engine, it is most commonly used in web browsers.



WebView

A WebView is a browser bundled inside of a mobile application producing what is called a hybrid app. Using a webview allows mobile apps to be built using Web technologies (HTML, JavaScript, CSS, etc.) but still package it as a native app and put it in the app store.



It seems so simple. A mini web browser, e.g. a WebView, runs your app, but you can make use of native APIs.

If you have web skills but want to reach a platform on which you have little experience - iOS, Android, Windows, OS X, or Linux - using a WebView is often a compelling option. And it doesn't have to be slow and clunky! Done right, hybrid mobile apps can often perform quite well.

- Android: WebView - system component powered by Chrome that allows Android apps to display web content.
- iOS: UIWebView and WKWebView
 - UIWebView is massive and clunky and leaks memory. It lags behind Mobile Safari, which has the benefit of the Nitro JavaScript engine.
 - WKWebView is the centerpiece of the modern WebKit API introduced in iOS 8 & OS X Yosemite. It replaces UIWebView in UIKit and WebView in AppKit, offering a consistent API across the two platforms.

Runtime: Node.js

Editors/IDEs

Table of Content

- Editors
- IDEs
- Essential Plugins

A source code editor is a text editor program designed specifically for editing source code of computer programs by programmers. It may be a standalone application or it may be built into an integrated development environment (IDE) or web browser. Source code editors are the most fundamental programming tool, as the fundamental job of programmers is to write and edit source code.

Front-end code can minimally be edited with a simple text editing application like Notepad orTextEdit. But, most front-end practitioners use a code editor specifically design for editing a programming language. Minimally, a code editor should have the following qualities (by default or by way of plugins):

- Good documentation on how to use the editor
- Report (i.e., hinting/linting/errors) on the code quality of HTML, CSS, and JavaScript.
- Offer syntax highlighting for HTML, CSS, and JavaScript.
- Offer code completion for HTML, CSS, and JavaScript.
- Be customizable by way of a plug-in architecture.
- Have available a large repository of third-party/community plug-ins that can be used to customize the editor to your liking.
- Be small, simple, and not coupled to the code (i.e., not required to edit the code).

Editors

- Sublime Text

Recommendation: version 3 or higher.

Package Control

- Homepage: <https://packagecontrol.io/>
- Installation: <https://packagecontrol.io/installation>

Sublime Text doesn't have official package manger so, we rely on this community-driven one.

- Atom
- Visual Studio Code

IDEs

- WebStorm
- Adobe Dreamweaver

Plugins

Toolings

Table of Content

- [Package Managers](#)
 - [npm](#)
 - [Bower](#)
- [Task Runners](#)
 - [npm scripts](#)
 - [Gulp](#)
 - [Grunt](#)

Package Managers

npm

Bower

Task Runners

npm scripts

Gulp

Grunt

Section 1: HTML

Table of Content

1. [Introduction](#)
 - [History](#)
 - [Hello world](#)
2. [Structure](#)
3. [Template](#)
4. [Template Engines](#)
5. [Conventions](#)

Introduction

HyperText Markup Language (HTML) is the standard markup language for creating web pages and web applications.

HTML is a markup language. The word markup was used by editors who marked up manuscripts (usually with a blue pencil) when giving instructions for revisions. "Markup" now means something slightly different: a language with specific syntax that instructs a Web browser how to display a page. Once again, HTML separates "content" (words, images, audio, video, and so on) from "presentation" (instructions for displaying each type of content). HTML uses a pre-defined set of elements to define content types. Elements contain one or more "tags" that contain or express content. Tags are enclosed by angle brackets, and the closing tag begins with a forward slash.

Most fundamentally, when you look at a webpage in a Web browser, you see words. But most of the time webpages contain styled text rather than plain text. Nowadays, webpage designers have access to hundreds of different fonts, font sizes, colors, and even alphabets (e.g. Spanish, Japanese, Russian), and browsers can for the most part display them accurately. Webpages may also contain images, video clips, and background music. They may include drop-down menus, search boxes, or links you can follow to access other pages (whether on the same site or another site). Some websites even let visitors customize the page display to accommodate their preferences and challenges (e.g., sight challenges, deafness, or color blindness). Often a page contains content boxes that move (scroll) while the rest of the page remains static.

HTML File	Tim Berners-Lee	HTML5
<pre><!DOCTYPE html> <html> <!-- created 2010-01-01 --> <head> <title>sample</title> </head> <body> <p>Voluptatem accusantium totam rem aperiam.</p> </body> </html></pre> 		

History

In 1980, physicist Tim Berners-Lee, a contractor at CERN, proposed and prototyped ENQUIRE, a system for CERN researchers to use and share documents. In 1989, Berners-Lee wrote a memo proposing an Internet-based hypertext system. Berners-Lee specified HTML and wrote the browser and server software in late 1990. That year, Berners-Lee and CERN data systems engineer Robert Cailliau collaborated on a joint request for funding, but the project was not formally adopted by CERN. In his personal notes from 1990 he listed "some of the many areas in which hypertext is used" and put an encyclopedia first.

The first publicly available description of HTML was a document called "HTML Tags", first mentioned on the Internet by Tim Berners-Lee in late 1991. It describes 18 elements comprising the initial, relatively simple design of HTML. Except for the hyperlink tag, these were strongly influenced by SGMLguid, an in-house Standard Generalized Markup Language (SGML)-based documentation format at CERN. Eleven of these elements still exist in HTML 4.

Since 1996, the HTML specifications have been maintained, with input from commercial software vendors, by the World Wide Web Consortium (W3C).

Edition	Date published	Note
HTML Tags	October 1991	CERN document listing 18 HTML tags, was first mentioned in public.
HTML DTD 1.1	November 1992	An informal draft
HTML 2.0	November 24, 1995	Published as IETF RFC 1866
HTML 3.0	Expired proposal	HTML 3.0 was proposed as a standard to the IETF, but the proposal expired five months later (28 September 1995) without further action.
HTML 3.2	January 14, 1997	Published as a W3C Recommendation. It was the first version developed and standardized exclusively by the W3C, as the IETF had closed its HTML Working Group on September 12, 1996.
HTML 4.0	December 18, 1997	Published as a W3C Recommendation. It offers three variations: - Strict, in which deprecated elements are forbidden. - Transitional, in which deprecated elements are allowed. - Frameset, in which mostly only frame related elements are allowed. Minor update in April 24, 1998
HTML 4.01	December 24, 1999	Update with HTML 4 Errata
HTML 5	October 28, 2014	Modern version of HTML
HTML 5.1	Late 2016	This is an work in-progress recommendation that expected to be released in late 2016. - HTML 5.1 Candidate Recommendation was published on 21 June 2016. - HTML 5.1 Proposed Recommendation was published on 15 September 2016

Hello world

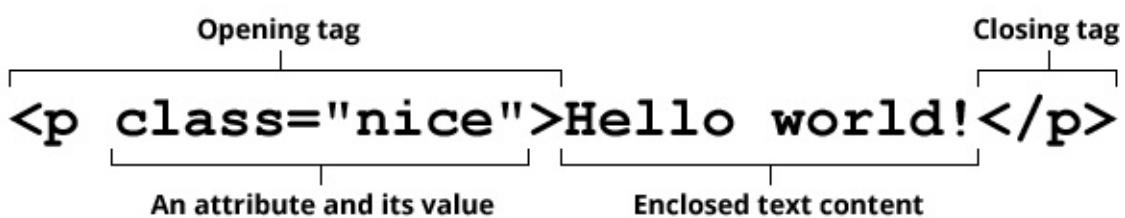
```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Document</title>
</head>
<body>
    <p>Hello world, I am <abbr title="Hyper Text Markup Language">HTML</abbr>.</p>
</body>
</html>
```

Structure

Table of Content

1. Elements
 - Block-level Elements
 - Inline Elements
 - Empty Elements
2. Tags
3. Attributes
4. doctype
5. Example

Anatomy of an HTML element



Elements

HTML (Hypertext Markup Language) elements are usually either "**block-level**" elements or "**inline**" elements.

Block-level Elements

A block-level element occupies the entire space of its parent element (container), thereby creating a "block".

Browsers typically display the block-level element with a newline both before and after the element. You can visualize them as a stack of boxes.

Usage:

- Block-level elements may appear only within a `<body>` element.
- By default, block-level elements begin on new lines.

Example:

```
<p>This paragraph is a block-level element; its background has been colored to display the paragraph's parent element.</p>
```

The screenshot shows a browser's developer tools with the 'HTML' tab selected. On the left, the raw HTML code is displayed: <p>This paragraph is a block-level element; its background has been colored to display the paragraph's parent element.</p>. To the right, the rendered output of this code is shown, where the entire paragraph is highlighted with a green background color.

The following is a list of common usage HTML block level elements (although "block-level" is not technically defined for elements that are new in HTML5).

Element	Description
<address>	The HTML <address> element supplies contact information for its nearest <article> or <body> ancestor; in the latter case, it applies to the whole document.
<article>	The HTML <article> element represents a self-contained composition in a document, page, application, or site, which is intended to be independently distributable or reusable (e.g., in syndication). This could be a forum post, a magazine or newspaper article, a blog entry, an object, or any other independent item of content. Each <article> should be identified, typically by including a heading (<h1> - <h6> element) as a child of the <article> element.
<aside>	The HTML <aside> element represents a section of the page with content connected tangentially to the rest, which could be considered separate from that content. These sections are often represented as sidebars or inserts. They often contain the definitions on the sidebars, such as definitions from the glossary; there may also be other types of information, such as related advertisements; the biography of the author; web applications; profile information or related links on the blog.
<blockquote>	The HTML <blockquote> Element (or HTML Block Quotation Element) indicates that the enclosed text is an extended quotation. Usually, this is rendered visually by indentation (see Notes for how to change it). A URL for the source of the quotation may be given using the cite attribute, while a text representation of the source can be given using the <cite> element.
	The HTML <canvas> Element can be used to draw graphics via

<code><canvas></code>	scripting (usually JavaScript). For example, it can be used to draw graphs, make photo compositions or even perform animations. You may (and should) provide alternate content inside the <code><canvas></code> block. That content will be rendered both on older browsers that don't support canvas and in browsers with JavaScript disabled.
<code><dd></code>	The HTML <code><dd></code> element (HTML Description Element) indicates the description of a term in a description list (<code><dl></code>) element. This element can occur only as a child element of a description list and it must follow a <code><dt></code> element.
<code><div></code>	The HTML <code><div></code> element (or HTML Document Division Element) is the generic container for flow content, which does not inherently represent anything. It can be used to group elements for styling purposes (using the class or id attributes), or because they share attribute values, such as lang. It should be used only when no other semantic element (such as <code><article></code> or <code><nav></code>) is appropriate.
<code><dl></code>	The HTML <code><dl></code> element (or HTML Description List Element) encloses a list of pairs of terms and descriptions. Common uses for this element are to implement a glossary or to display metadata (a list of key-value pairs).
<code><fieldset></code>	The HTML <code><fieldset></code> element is used to group several controls as well as labels (<code><label></code>) within a web form.
<code><figcaption></code>	The HTML <code><figcaption></code> element represents a caption or a legend associated with a figure or an illustration described by the rest of the data of the <code><figure></code> element which is its immediate ancestor which means <code><figcaption></code> can be the first or last element inside a <code><figure></code> block. Also, the HTML Figcaption Element is optional; if not provided, then the parent figure element will have no caption.
<code><figure></code>	The HTML <code><figure></code> element represents self-contained content, frequently with a caption (<code><figcaption></code>), and is typically referenced as a single unit. While it is related to the main flow, its position is independent of the main flow. Usually this is an image, an illustration, a diagram, a code snippet, or a schema that is referenced in the main text, but that can be moved to another page or to an appendix without affecting the main flow.
<code><footer></code>	The HTML <code><footer></code> element represents a footer for its nearest sectioning content or sectioning root element. A footer typically contains information about the author of the section, copyright data or links to related documents.
<code><form></code>	The HTML <code><form></code> element represents a document section that contains interactive controls to submit information to a web server.
<code><h1> , <h2> , <h3> , <h4> , <h5> , <h6></code>	Heading elements implement six levels of document headings, <code><h1></code> is the most important and <code><h6></code> is the least. A heading element briefly describes the topic of the section it introduces. Heading information may be used by user agents, for example, to construct a table of contents for a document automatically.
	The HTML <code><header></code> element represents a group of introductory or

<header>	navigational aids. It may contain some heading elements but also other elements like a logo, wrapped section's header, a search form, and so on.
<hgroup>	The HTML <hgroup> Element (HTML Headings Group Element) represents the heading of a section. It defines a single title that participates in the outline of the document as the heading of the implicit or explicit section that it belongs to.
<hr>	The HTML <hr> element represents a thematic break between paragraph-level elements (for example, a change of scene in a story, or a shift of topic with a section). In previous versions of HTML, it represented a horizontal rule. It may still be displayed as a horizontal rule in visual browsers, but is now defined in semantic terms, rather than presentational terms.
	The HTML element (or HTML List Item Element) is used to represent an item in a list. It must be contained in a parent element: an ordered list (), an unordered list (), or a menu (<menu>). In menus and unordered lists, list items are usually displayed using bullet points. In ordered lists, they are usually displayed with an ascending counter on the left, such as a number or letter.
<main>	The HTML <main> element represents the main content of the <body> of a document or application. The main content area consists of content that is directly related to, or expands upon the central topic of a document or the central functionality of an application. This content should be unique to the document, excluding any content that is repeated across a set of documents such as sidebars, navigation links, copyright information, site logos, and search forms (unless the document's main function is as a search form).
<nav>	The HTML <nav> element (HTML Navigation Element) represents a section of a page that links to other pages or to parts within the page: a section with navigation links.
<noscript>	The HTML <noscript> Element defines a section of html to be inserted if a script type on the page is unsupported or if scripting is currently turned off in the browser.
	The HTML Element (or HTML Ordered List Element) represents an ordered list of items. Typically, ordered-list items are displayed with a preceding numbering, which can be of any form, like numerals, letters or Romans numerals or even simple bullets. This numbered style is not defined in the HTML description of the page, but in its associated CSS, using the list-style-type property.
<output>	The HTML <output> element represents the result of a calculation or user action.
<p>	The HTML <p> element (or HTML Paragraph Element) represents a paragraph of text.
<pre>	The HTML <pre> element (or HTML Preformatted Text) represents preformatted text. Text within this element is typically displayed in a

<code><pre></code>	non-proportional ("monospace") font exactly as it is laid out in the file. Whitespace inside this element is displayed as typed.
<code><section></code>	The HTML <code><section></code> element represents a generic section of a document, i.e., a thematic grouping of content, typically with a heading. Each <code><section></code> should be identified, typically by including a heading (<code><h1></code> - <code><h6></code> element) as a child of the <code><section></code> element.
<code><table></code>	The HTML Table Element (<code><table></code>) represents tabular data - i.e., information expressed via a two dimensional data table.
<code><tfoot></code>	The HTML Table Foot Element (<code><tfoot></code>) defines a set of rows summarizing the columns of the table.
<code></code>	The HTML <code></code> element (or HTML Unordered List Element) represents an unordered list of items, namely a collection of items that do not have a numerical ordering, and their order in the list is meaningless. Typically, unordered-list items are displayed with a bullet, which can be of several forms, like a dot, a circle or a squared. The bullet style is not defined in the HTML description of the page, but in its associated CSS, using the <code>list-style-type</code> property.
<code><video></code>	Use the HTML <code><video></code> element to embed video content in a document. The video element contains one or more video sources. To specify a video source, use either the <code>src</code> attribute or the <code><source></code> element; the browser will choose the most suitable one.

Inline Elements

An inline element occupies only the space bounded by the tags that define the inline element.

Usage:

- Generally, inline elements may contain only data and other inline elements.
- By default, inline elements do not begin with new line and can start anywhere in a line.

Example:

```
<p>This <span>span</span> is an inline element; its background has been colored to display both the beginning and end of the inline element's influence</p>
```

HTML

```
<p>This <span>span</span> is an inline element; its background has been colored to display both the beginning and end of the inline element's influence</p>
```

CSS

JS

This span is an inline element; its background has been colored to display both the beginning and end of the inline element's influence

The following elements are "inline":

- Text enhancement: `b`, `big`, `i`, `small`, `tt`, `strong`, `em`.
- Text semantics: `a`, `bdo`, `br`, `img`, `map`, `object`, `q`, `script`, `span`, `sub`, `sup`, `abbr`, `acronym`, `cite`, `code`, `dfn`, `kbd`, `samp`, `time`, `var`.
- Form elements: `button`, `input`, `label`, `select`, `textarea`.

Empty Elements

HTML elements with no content are called empty elements.

Usage:

- An empty element has no closing [tag](#).
- HTML5 does not require empty elements to be closed.

Example:

```
<span>This span is an inline element.</span><br><span>Its background has been colored to display both the beginning and end of the inline element's influence</span>
```

HTML

```
<span>This span is an inline element.</span><br>Its background has been colored to display both the beginning and end of the inline element's influence.
```

CSS

JS

This span is an inline element
Its background has been colored to display both the beginning and end of the inline element's influence.

The empty elements in HTML are as follows:

Element	Description
<area>	The HTML <code><area></code> element defines a hot-spot region on an image, and optionally associates it with a hypertext link. This element is used only within a <code><map></code> element.
<base>	The HTML <code><base></code> element specifies the base URL to use for all relative URLs contained within a document. There can be only one <code><base></code> element in a document.
 	The HTML element line break <code>
</code> produces a line break in text (carriage-return). It is useful for writing a poem or an address, where the division of lines is significant.
<col>	The HTML Table Column Element (<code><col></code>) defines a column within a table and is used for defining common semantics on all common cells. It is generally found within a <code><colgroup></code> element.
<colgroup>	The HTML Table Column Group Element (<code><colgroup></code>) defines a group of columns within a table.
<command>	The command element represents a command which the user can invoke.
<embed>	The HTML <code><embed></code> Element represents an integration point for an external application or interactive content (in other words, a plug-in).
<hr>	The HTML <code><hr></code> element represents a thematic break between paragraph-level elements (for example, a change of scene in a story, or a shift of topic with a section). In previous versions of HTML, it represented a horizontal rule. It may still be displayed as a horizontal rule in visual browsers, but is now defined in semantic terms, rather than presentational terms.
	The HTML <code></code> element represents an image in the document.
<input>	The HTML element <code><input></code> is used to create interactive controls for web-based forms in order to accept data from the user. How an <code><input></code> works varies considerably depending on the value of its type attribute.
<keygen>	The HTML <code><keygen></code> element exists to facilitate generation of key material, and submission of the public key as part of an HTML form. This mechanism is designed for use with Web-based certificate management systems. It is expected that the <code><keygen></code> element will be used in an HTML form along with other information needed to construct a certificate request, and that the result of the process will be a signed certificate.
<link>	The HTML <code><link></code> element specifies relationships between the current document and an external resource. Possible uses for this element include defining a relational framework for navigation. This Element is most used to link to style sheets.
<meta>	The HTML <code><meta></code> element represents any metadata information that cannot be represented by one of the other HTML meta-related elements (<code><base></code> , <code><link></code> , <code><script></code> , <code><style></code> or <code><title></code>).

<param>	The HTML <param> Element (or HTML Parameter Element) defines parameters for <object> .
<source>	The HTML <source> element specifies multiple media resources for either the <picture> the <audio> or the <video> element. It is an empty element. It is commonly used to serve the same media content in multiple formats supported by different browsers.
<track>	The HTML <track> element is used as a child of the media elements — <audio> and <video> . It lets you specify timed text tracks (or time-based data), for example to automatically handle subtitles. The tracks are formatted in WebVTT format (.vtt files) — Web Video Text Tracks.
<wbr>	The HTML element word break opportunity <wbr> represents a position within text where the browser may optionally break a line, though its line-breaking rules would not otherwise create a break at that location.

Tags

HTML attaches special meaning to anything that starts with the less-than sign (<) and ends with the greater-than sign (>). Such markup is called a **tag**. In HTML a tag is used for creating an element.

Usage:

- *Closing tags* are the same as the *start tag* except with a forward slash (/) immediately after the leading less-than sign (</p>).
- Most elements in HTML are written using both start and closing tags. There are **empty elements** and need no closing tag.
- You must properly nest start and closing tags, that is, write close tags in the opposite order from the start tags.

Example:

Valid	Invalid
I really mean that. 	I really mean that.

Attributes

Attributes provide additional information about *HTML elements*.

Usage:

- All HTML elements can have attributes.
- Attributes provide additional information about an element.

- Attributes are always specified in the start tag.
- Attributes usually consist of 2 parts: **name** and **value** in pairs like `name="value"`.
- You must enclose values containing spaces in quotation marks (either single (') or double (") quotes).
- A few attributes can only have one value. They are **Boolean** attributes and may be shortened by only specifying the attribute name or leaving the attribute value empty.

Example:

Normal	Boolean
<code><p class="foo">bar</p></code>	<code><input required="required"></code> OR <code><input required=""></code> OR <code><input required></code>

The most comprehensive list of HTML `attribute`, please follow this [references](#).

doctype

HTML document must contain a **doctype** declaration as the first line. The **doctype** declaration is not an HTML tag, but rather tells the browser which version of HTML the page is written in.

Usage:

- In HTML5, there is only one declaration and is written like this:

```
<!DOCTYPE html>
```

Example

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>A tiny document</title>
</head>
<body>
  <h1>Main heading in my document</h1>
  <!-- Note that it is "h" + "1", not "h" + the letters "one" -->
  <p>Look Ma, I am coding <abbr title="Hyper Text Markup Language">HTML</abbr>. </p>
</body>
</html>
```


Template

Table of Content

1. Basic
2. Organizing
 - i. Layout
 - ii. Modules

Basic

- In its simplest form, following is an example of an HTML document:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>HTML 101 Template</title>

    <!-- Stylesheet -->
    <link href="path/to/style.css" rel="stylesheet">
  </head>
  <body>
    <h1>Hello, world!</h1>

    <!-- Scripts -->
    <script src="https://path.to/some/external/script.js"></script>
    <script src="path/to/script.js"></script>
  </body>
</html>
```

- A typical HTML document will have following structure:

Visualized	Tear down
<pre data-bbox="181 323 414 781"><!DOCTYPE html> <html> <head> <title>Page title</title> </head> <body> <h1>This is a heading</h1> <p>This is a paragraph.</p> <p>This is another paragraph.</p> </body> </html></pre>	<ul style="list-style-type: none"> - The <code><!DOCTYPE html></code> declaration defines this document to be HTML5. - The <code><html></code> element is the root element of an HTML page. - The <code><head></code> element contains meta information about the document. - The <code><title></code> element specifies a title for the document. - The <code><body></code> element contains the visible page content. - The <code><h1></code> element defines a large heading. - The <code><p></code> element defines a paragraph.

Organizing

HTML5 brings precision to the sectioning and heading features, allowing document outlines to be predictable and used by the browser to improve the user experience.

Layout

Websites often display content in multiple columns (like a magazine or newspaper).

**Header****Featured Story A****Responsive Image****Social Callout**

HTML5 offers new semantic elements that define the different parts of a web page:

- `<header>` - Defines a header for a document or a section
- `<nav>` - Defines a container for navigation links
- `<section>` - Defines a section in a document
- `<article>` - Defines an independent self-contained article
- `<aside>` - Defines content aside from the content (like a sidebar)
- `<footer>` - Defines a footer for a document or a section
- `<details>` - Defines additional details
- `<summary>` - Defines a heading for the `<details>` element

A walkthrough of layout techniques...

- **HTML tables:** an ancient!

The `<table>` element was not designed to be a layout tool! The purpose of the `<table>` element is to display tabular data. **So, do not use tables for your page layout!**
- **CSS Floats:** It is common to do entire web layouts using the CSS float property. Float is easy to learn - you just need to remember how the float and clear properties work.

Disadvantages: Floating elements are tied to the document flow, which may harm the flexibility.
- **CSS Frameworks:** If you want to create your layout fast, you can use a framework. Sounds great. **But how does it work???**
- **CSS Flexbox:** Flexbox is a new layout mode in CSS3. Use of flexbox ensures that

elements behave predictably when the page layout must accommodate different screen sizes and different display devices.

Disadvantages: Does not work in **IE10** and earlier.

Example 1:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>HTML 101 Template</title>
    <!-- Stylesheet -->
    <link href="path/to/style.css" rel="stylesheet">
  </head>
  <body>
    <header>
      <h1>Hello, world!</h1>
    </header>

    <!-- Our content goes here -->

    <footer>
      <a href="http://i.am.here/">
    </footer>
    <!-- Scripts -->
    <script src="https://path.to/some/external/script.js"></script>
    <script src="path/to/script.js"></script>
  </body>
</html>
```

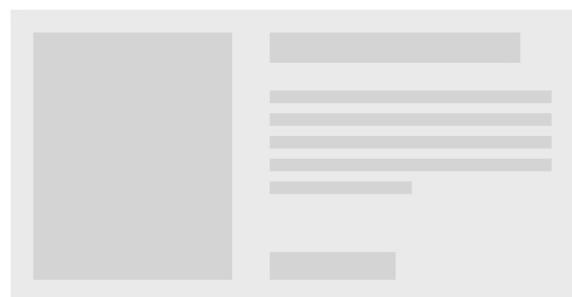
Modules

Modular design is the method of creating a system of self-contained components that can be stacked, rearranged and used or not used, case by case.

- The goal is to be able to easily change or adapt individual design patterns without altering the system as a whole.
- One benefit of the modular system is that you need to design only a single component to address this situation.



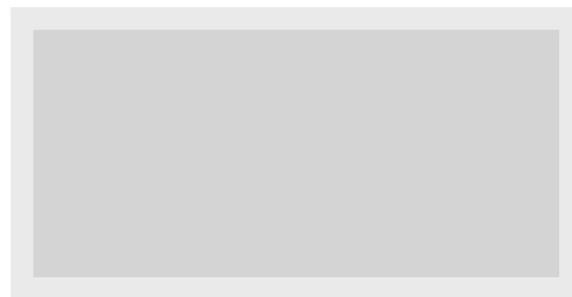
Featured Story A



Featured Story B



General Content Block



Responsive Image

Usage:

- A basic template will serve as your base layout structure and will allow global elements to be shared across all pages.
- Once you determine common use cases, you can start to design individual components according to your needs.
- Designing several variations of each use case might also be helpful.

Example 2:

- Define a module with `<section>`

```

<section>
  <h1>Forest elephants</h1>
  <section>
    <h2>Introduction</h2>
    <p>In this section, we discuss the lesser known forest elephants</p>
  </section>
  <section>
    <h2>Habitat</h2>
    <p>Forest elephants do not live in trees but among them. Let's
      look what scientists are saying in "<cite>The Forest Elephant in Borneo</cite>":</p>
  </p>
  <blockquote>
    <h1>Borneo</h1>
    <p>The forest element lives in Borneo...</p>
  </blockquote>
  </section>
  <aside>
    <p>advertising block</p>
  </aside>
</section>

```



- Implement module into `<body>`

```

<body>
  <h1>Mammals</h1>
  <h2>Whales</h2>
  <p>In this section, we discuss the swimming whales.
    ...this section continues...
  <section>
    <h3>Forest elephants</h3>
    <p>In this section, we discuss the lesser known forest elephants.
      ...this section continues...
    <h3>Mongolian gerbils</h3>
    <p>Hordes of gerbils have spread their range far beyond Mongolia.
      ...this subsection continues...
    <h2>Reptiles</h2>
    <p>Reptiles are animals with cold blood.
      ...this section continues...
  </section>
</body>

```

Put It All Together

With a base template and modular design patterns in place, it's now time to snap all of the LEGO blocks together.



Example 3:

Wondering where the result is? Well, try it out for yourself!

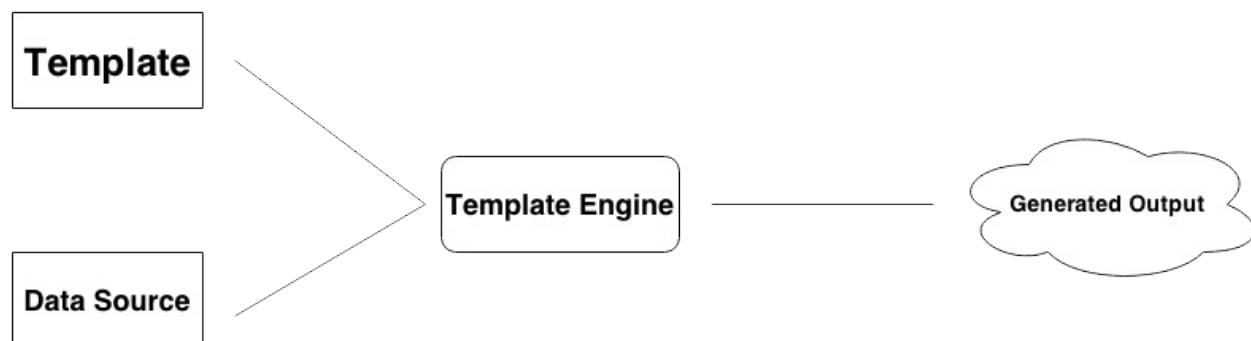
Template Engines

Table of Content

1. [What is template engine?](#)
2. [Popular Engines](#)
 - [mustache.js](#)
 - [Handlebars.js](#)
 - [Pug \(Jade\)](#)
3. [Comparisons](#)

What is template engine?

Template Engines are tools that help us break HTML code into smaller pieces that we can reuse across multiple HTML files. They also give you the power to feed data into variables that help you simplify your code.



Two benefits of template engines

Here are the two benefits that template engines bring:

- It lets you **break HTML code into smaller files**

It's common for a HTML file to contain blocks of code that are repeated across the website. Consider this markup for a second:

```
<body>
  <nav> ... </nav>
  <div class="content"> ... </div>
  <footer> ... </footer>
</body>
```

Much of this code, particularly those within nav and footer, are repeated across multiple pages. Since they are repeated, we can pull them out and place them into smaller files called partials. Just imagine what you would do if you had to change the navigation now. When you use a partial, all you have to do is change code in the navigation partial and all your pages will be updated. Contrast that with having to change the same code across every file the navigation is used on. Which is easier and more effective? This ability to break code up into smaller files helps us write lesser (duplicated) code while at the same time preserve our sanity when code need to be changed.

- It lets you **use data to populate your markup**

```
<div class="gallery">
  <div class="gallery__item">
    
  </div>
  <div class="gallery__item">
    
  </div>
  <div class="gallery__item">
    
  </div>
  <div class="gallery__item">
    
  </div>
  <div class="gallery__item">
    
  </div>
</div>
```

Notice how the `.gallery__item` div was repeated multiple times above? If you had to change the markup of one `.gallery__item`, you'd have to change it in five different places. Now, imagine if you had the ability to write HTML with some sort of logic. You'd probably write something similar to this:

```
<div class="gallery">
  <!-- Some code to loop through the following 5 times: -->
  <div class="gallery__item">
    
  </div>
  <!-- end loop -->
</div>
<!-- Apply some real world code -->
<div class="gallery">

</div>
```

The best part is that you only have to change the markup once and all `.gallery__items` would be updated.

Here, template engines once again gives you the ability to write lesser code, and helps preserve your sanity when code needs to be changed.

Popular Engines

mustache.js

Mustache is a multi-language, logic-less templating system. The mustache.js implementation is just one of many. So once we're used to the (very simple) syntax, we can use it in a variety of programming languages.

Key Points:

- Simple
- No dependencies
- No logic
- No precompiled templates
- Programming language agnostic

Examples:

```
<script id="template" type="x-tmpl-mustache">
  <p>Use the <strong>{{power}}</strong>, {{name}}!</p>
</script>
```

```
//Grab the inline template
var template = document.getElementById('template').innerHTML;

//Parse it (optional, only necessary if template is to be used again)
Mustache.parse(template);

//Render the data into the template
var rendered = Mustache.render(template, {name: "Luke", power: "force"});

//Overwrite the contents of #target with the rendered HTML
document.getElementById('target').innerHTML = rendered;
```

The screenshot shows a browser's developer tools with two tabs: 'HTML' and 'JS'. The 'HTML' tab displays the rendered content: 'Use the **force**, Luke!'. The 'JS' tab shows the executed JavaScript code. The code includes a window.onload event listener that performs the following steps:

- It grabs the inline template from the page.
- It parses the template using Mustache.parse().
- It renders the data into the template using Mustache.render() with the object {name: "Luke", power: "force"}.
- Finally, it overwrites the contents of the #target element with the rendered HTML.

```
1 //Grab the inline template
2 var template = document.getElementById('template').innerHTML;
3
4 //Parse it (optional, only necessary if template is to be used again)
5 Mustache.parse(template);
6
7 //Render the data into the template
8 var rendered = Mustache.render(template, {name: "Luke", power: "force"});
9
10 //Overwrite the contents of #target with the rendered HTML
11 document.getElementById('target').innerHTML = rendered;
12
13 }
```

Use the **force**, Luke!

Summary:

mustache.js is perfect for small projects and quick prototypes where templating complexity is at a minimum. A key thing to note is that we can start off a project with mustache.js and then easily upgrade to Handlebars.js later as the templates are (mostly) the same.

Handlebars.js

Handlebars.js is built on top of Mustache and is mostly compatible with Mustache templates. In a nutshell, it provides everything mustache.js provides, plus it supports **block expressions** and **precompiled templates**.

Key Points:

- Block expressions (helpers)
- Precompiled templates
- No dependencies

Example:

```
<script id="template" type="text/x-handlebars-template">
  <p>Use the <strong>{{power}}</strong>, {{name}}!</p>
</script>
```

```
//Grab the inline template
var template = document.getElementById('template').innerHTML;

//Compile the template
var compiled_template = Handlebars.compile(template);

//Render the data into the template
var rendered = compiled_template({name: "Luke", power: "force"});

//Overwrite the contents of #target with the renderer HTML
document.getElementById('target').innerHTML = rendered;
```

```
1 <div id="target"></div>
2
3 <script id="template" type="text/x-handlebars-template">
4   <p>Use the <strong>{{power}}</strong>, {{name}}!</p>
5 </script>
```

```
1 window.onload = function() {
2   //Grab the inline template
3   var template = document.getElementById('template').innerHTML;
4
5   //Compile the template
6   var compiled_template = Handlebars.compile(template);
7
8   //Render the data into the template
9   var rendered = compiled_template({name: "Luke", power: "force"});
10
11  //Overwrite the contents of #target with the rendered HTML
12  document.getElementById('target').innerHTML = rendered;
13 }
```

Use the **force**, Luke!

Summary:

Handlebars.js is perfect for projects where performance is important and we are not hugely worried about adding 18kb to the weight of the page. It's also the way to go if we want to make use of block expressions.

Pug (Jade)

Pug is an elegant templating engine, primarily used for server-side templating in Node.js. In plain words, Pug gives you a powerful new way to write markup, with a number of advantages over plain HTML.

Key Points:

- Node.js integrated
- Feature rich
- Simple syntax
- No dependencies

Example:

```
- var x = 5;
div
  ul
    - for (var i=1; i<=x; i++) {
      li Hello
    }
```



```

HTML (Jade)
1 - var x = 5;
2 div
3   ul
4     - for (var i=1; i<=x; i++) {
5       li Hello
6     }

```

```

CSS
1 body {
2   padding: 0 20px;
3 }

```

• Hello
• Hello
• Hello
• Hello
• Hello

Summary:

The Pug template engine (for Node.js) is literally enabling developers to write code that looks like paragraphs straight out of a book. Not only does this improve the overall code productivity, it can help to streamline the work on a project that consists of multiple team members.

Comparisons

Key	Handlebars.js	Pug (Jade)
Syntax	Text	Shorthand HTML
Streaming	No	No
Async	No	No
Auto-escape	Yes	Yes
Preprocessor support	No	Yes
Logical	No	Supported
Pre-compile	Yes	Yes (extra JS output)

Conventions

Table of Contents

1. General formatting
2. Default structure
3. Elements
4. Attributes
5. Courtesies
6. Semantic Elements
7. Recommendation

General formatting

- Paragraphs of text should always be placed in a `<p>` tag. Never use multiple `
` tags.
- Items in list form should always be in `` , `` , or `<dl>` . Never use a set of `<div>` or `<p>` .
- Every form input that has text attached should utilize a `<label>` tag. **Especially radio or checkbox elements.**
- Even though quotes around attributes is optional, always put quotes around attributes for readability.
- Avoid writing closing tag comments, like `<!-- /.element -->` . This just adds to page load time. Plus, most editors have indentation guides and open-close tag highlighting.
- Avoid trailing slashes in self-closing elements. For example, `
` , `<hr>` , `` , and `<input>` .
- Don't set `tabindex` manually—rely on the browser to set the order.

Default structure

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta http-equiv="x-ua-compatible" content="ie=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Document</title>
  <link rel="stylesheet" href="assets/css/styles.css">
</head>
<body>
  <div id="wrapper">
    <header id="head">
      <div class="container">
        <nav>...</nav>
      </div>
    </header>
    <div id="page-content">
      <section id="news" class="container">
        <article id="content-article">
          <h2>...</h2>
          <p>...</p>
        </article>
        <aside id="content-table">
          ...
        </aside>
      </section>
      <section id="comments">
        ...
      </section>

    </div>
    <footer>
      <div class="container">
        <p>Copyright © 2016 Asian Tech Co., Ltd.</p>
      </div>
    </footer>
  </div>
  <script src="assets/js/script.js"></script>
</body>
</html>

```

Elements

- Add closing tags to elements that aren't self-closing.
- Don't add the forward slash for self-closing elements.
- Avoid unnecessary markup, such as surrounding a block element in a `<div>`.
- Use lowercase, not uppercase for File Names, Attribute Names, Element Names.

```
<!--Bad-->
<DIV>
<UL>
  <LI>Item 1
  <LI>Item 2
</UL>
</DIV>
<INPUT type="text" />

<!--Good-->
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
<input type="text">
```

```
<!-- Bad -->
<div CLASS="menu">

<!-- Good -->
<div class="menu">
```

- `image.jpg` cannot be accessed as `Image.jpg`
- Use correct HTML5 tags for each content. Avoid using divs for everything.
- Avoid using P, SPAN for structuring

```
<section>      <!-- for each big content block on the same page. -->
<aside>        <!-- for left/right hand side content -->
<article>       <!-- for each item that has title/content structure -->
<h1><h2>...<h6> <!-- for title or text with big fonts -->
<nav>          <!-- for menu/navigation -->
<header>        <!-- for header section -->
<footer>        <!-- for footer section -->
```

Attributes

- Surround values for attributes in double quotes.
- Boolean attributes, such as required, do not need a value.
- Follow this order for attributes:

- `class`
- `id` , `name`
- `data-*`

- `src` , `for` , `type` , `href` , `value`
- `title` , `alt`
- `role` , `aria-*`

```
<!-- Bad -->
<input type='radio' class="input" required='true'>

<!-- Good -->
<input class="input" type="radio" required>

<a href="#" class="..." id="..." data-toggle="modal">
Example link
</a>

<input class="form-control" type="text">

</pre>
```

Courtesies

- Put a single line break between blocks/components.
- Avoid adding comments. An exception would be a closing comment for large blocks.

```
<div class="video-player">
...
</div>

<div class="comment-list">
...
</div></pre>
```

Semantic Elements

- HTML5 offers new semantic elements to define different parts of a web page:

```

<article>      <!-- specifies independent, self-contained content -->
<aside>        <!-- defines some content aside from the content it is placed in (like
                 e.g. a sidebar) -->
<details>       <!-- defines additional details that the user can view or hide -->
<figcaption>   <!-- defines a caption for a `<figure>` element -->
<figure>        <!-- specifies self-contained content, like illustrations, diagrams,
                 photos, code listings, etc -->
<footer>       <!-- specifies a footer for a document or section -->
<header>        <!-- specifies a header for a document or section -->
<main>          <!-- specifies the main content of a document -->
<mark>          <!-- defines marked/highlighted text -->
<nav>           <!-- defines a set of navigation links -->
<section>       <!-- defines a section in a document -->
<summary>       <!-- defines marked/highlighted text -->
<time>          <!-- defines a date/time -->

```

Recommendation

The Difference Between ID and Class

- The simple difference between the two is that while a class can be used repeatedly on a page, an ID must only be used once per page. Therefore, it is appropriate to use an ID on the div element that is marking up the main content on the page, as there will only be one main content section. In contrast, you must use a class to set up alternating row colors on a table, as they are by definition going to be used more than once.
- ID's are unique :
 - Each element can have only one ID.
 - Each page can have only one element with that ID.
- Classes are NOT unique :
 - You can use the same class on multiple elements.
 - You can use multiple classes on the same element.
- ID's have special browser functionality
- Classes have no special abilities in the browser, but ID's do have one very important trick up their sleeve. This is the "hash value" in the URL. If you have a URL like <http://yourdomain.com#comments>, the browser will attempt to locate the element with an ID of "comments" and will automatically scroll the page to show that element. It is important to note here that the browser will scroll whatever element it needs to in order to show that element, so if you did something special like a scrollable DIV area within your regular body, that div will be scrolled too.

- This is an important reason right here why having ID's be absolutely unique is important. So your browser knows where to scroll!
- Elements can have BOTH
- There is nothing stopping you from having both an ID and a Class on a single element. In fact, it is often a very good idea. Take for example the default markup for a WordPress comment list item:

```
<li id="comment-27299" class="item">
```

Do not use inline style attributes

- As much as possible, use css classes instead of using inline style attributes. This has the advantages: Makes it easier to change the look of the UI by changing the stylesheet Makes it possible to reuse the css class in another place, giving a more consistent UI and making it possible to change the look by changing only one place.

```
<! -- Bad -->
<div style="width:100px;align:center;">

<! -- Good -->
<div class="message">
```

Section 2: CSS

Table of Content

1. [Introduction](#)
 - [History](#)
 - [Evolution](#)
 - [Syntax](#)
 - [Terminology](#)
2. [Selectors](#)
3. [Properties](#)
4. [Functions](#)
5. [at-rules](#)
6. [Media queries](#)
7. [Preprocessors](#)
 - [Sass \(SCSS\)](#)
 - [Less](#)
 - [Stylus](#)
8. [CSS Frameworks](#)
9. [Conventions](#)

Introduction

Cascading Style Sheets (CSS) are a stylesheet language used to describe the presentation of a document written in HTML or XML (including XML dialects like SVG or XHTML). CSS describes how the structured elements in the document are to be rendered on screen, on paper, in speech, or on other media. The ability to adjust the document's presentation depending on the output medium is a key feature of CSS.

CSS File	Håkon Wium Lie	CSS3
<pre> h1 { color: white; background: orange; border: 1px solid black; padding: 0 0 0 0; font-weight: bold; } /* begin: seaside-theme */ body { background-color:white; color:black; font-family:Arial,sans-serif; margin: 0 4px 0 0; border: 12px solid; } </pre> 		

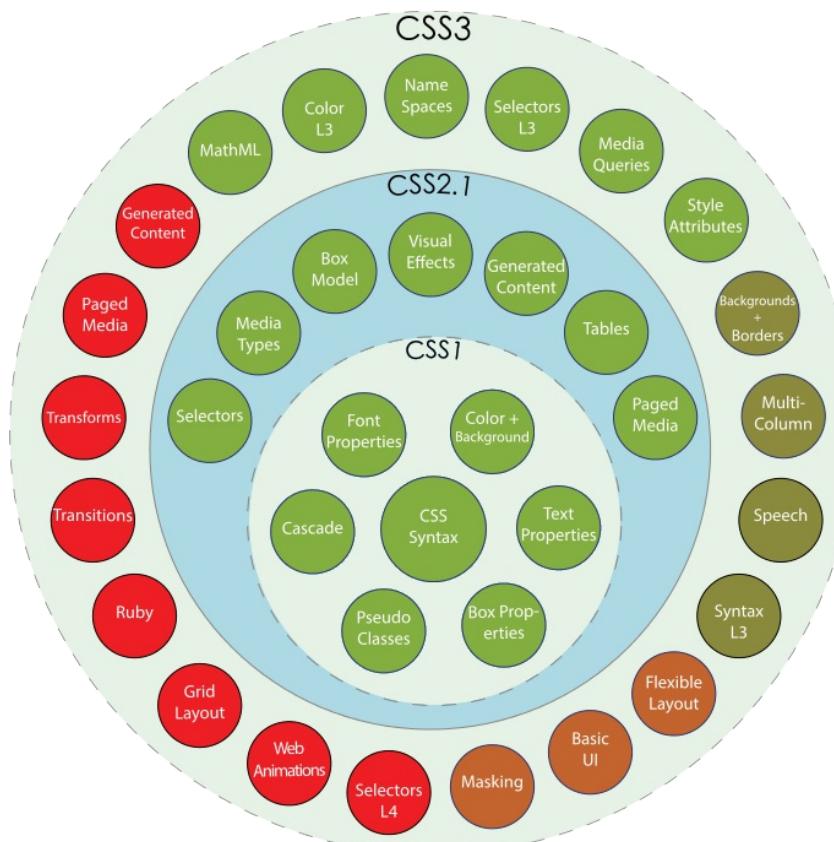
History

CSS is one of the core languages of the open web and has a standardized W3C specification. Developed in levels, CSS1 is now obsolete, CSS2.1 is a recommendation, and CSS3, now split into smaller modules, is progressing on the standardization track.

CSS was first proposed by Håkon Wium Lie on October 10, 1994. At the time, Lie was working with Tim Berners-Lee at CERN. Several other style sheet languages for the web were proposed around the same time, and discussions on public mailing lists and inside World Wide Web Consortium resulted in the first W3C CSS Recommendation (CSS1) being released in 1996. In particular, Bert Bos' proposal was influential; he became co-author of CSS1 and is regarded as co-creator of CSS.

Edition	Date published	Note
CSS 1	December 17, 1996	The first CSS specification to become an official W3C Recommendation is CSS level 1.
CSS 2	May 1998	A superset of CSS 1, CSS 2 includes a number of new capabilities like absolute, relative, and fixed positioning of elements and z-index, the concept of media types, support for aural style sheets (which were later replaced by the CSS 3 speech modules) and bidirectional text, and new font properties such as shadows.
CSS 2.1	June 7, 2011	CSS level 2 revision 1, often referred to as "CSS 2.1", fixes errors in CSS 2, removes poorly supported or not fully interoperable features and adds already implemented browser extensions to the specification.
CSS 3		Unlike CSS 2, which is a large single specification defining various features, CSS 3 is divided into several separate documents called "modules". Each module adds new capabilities or extends features defined in CSS 2, preserving backward compatibility. Work on CSS level 3 started around the time of publication of the original CSS 2 recommendation. The earliest CSS 3 drafts were published in June 1999.

Evolution



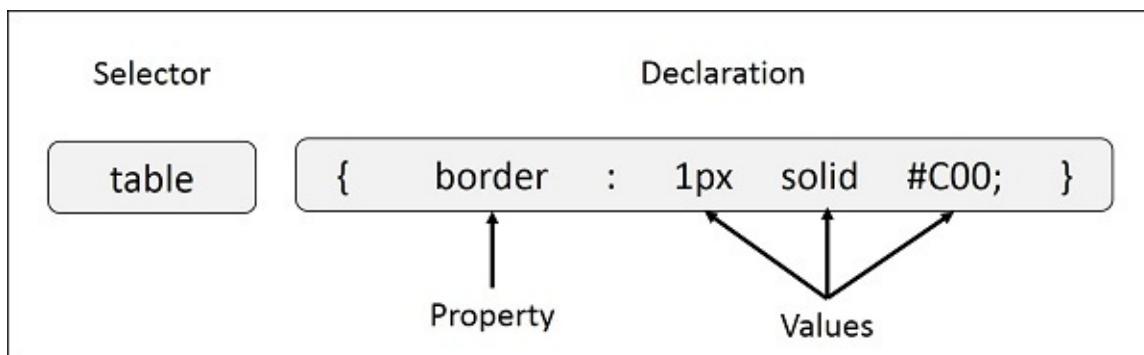
The taxonomy and status of CSS3 modules

Key	Explain
●	Recommendation
●	Candidate Recommendation
●	Last Call
●	Working Draft.

For all CSS3 modules status, please refer to

<https://www.w3.org/Style/CSS/specs.en.html>

Syntax



Terminology

Term	Explanation
Rule set	<p>A rule set is a single section of CSS including the selector, the curly braces, and the different lines with properties and values. The code in the example below comprises one rule set:</p> <pre>/* the rule set starts with the line below */ body { font-family: Arial, sans-serif; color: #555; font-size: 14px; } /* ends with the closing curly brace above */</pre>
	<p>A declaration block is the section of CSS where the property/value pairs appear. In the example below, everything found between the curly braces (not including the comments) is a declaration block:</p>

Declaration block	<pre>body { font-family: Arial, sans-serif; /* starts with this line */ color: #555; font-size: 14px; line-height: 20px; /* ends here, before the closing curly brace */ }</pre>
Declaration	<p>A declaration is generally any single line of CSS that appears between the curly braces, whether shorthand or longhand. In the example below, everything after the first curly brace, and before the last curly brace (not including the comment) is a declaration:</p> <pre>body { font-family: Arial, sans-serif; /* this line is a declaration */ }</pre>
Property	<p>A property is what appears before the colon in any line of CSS. In the example below, the word "width" is the property.</p> <pre>#box { width: 200px; /* the property is "width" (without the colon) */ }</pre>
Value	<p>A value is what appears immediately after the colon in any line of CSS. In the example below, the "200px" is the value.</p> <pre>#box { width: 200px; /* after the colon, without the semi-colon */ }</pre> <p>If you use shorthand, a single declaration could have multiple values. We will cover this later in its own section.</p>
Selector	<p>A selector is the part of the CSS line that selects what element to target with the property/value pair. In the example below "#container #box" is the selector:</p> <pre>/* the selector is everything on the first line */ /* excluding the opening curly brace */ #container #box { width: 200px; }</pre>

Element Type Selector	<p>An element type selector is a selector that targets an element by the tag name. The selector in the example below is an element type selector, because it doesn't use a class, ID, or other selector to apply the given styles. Instead, it directly targets all HTML5 <code><nav></code> elements:</p> <pre data-bbox="430 332 1113 534">/* matches an HTML element by name */ nav { font-family: Arial, sans-serif; color: #555; font-size: 14px; }</pre>
Class Selector	<p>The selector in the example below targets an element by its class name. So every element with a class of "navigation" will receive the styles in question:</p> <pre data-bbox="430 765 1351 945">/* matches HTML element with class="navigation" */ .navigation { width: 960px; margin: 0 auto; }</pre>
ID Selector	<p>The selector in the example below targets an element by its ID. So every element with an ID of "navigation" will receive the styles in question:</p> <pre data-bbox="430 1140 1303 1320">#navigation { width: 960px; margin: 0 auto; }</pre> <p>In HTML, a page will have validation errors if two or more elements share the same ID.</p>
Universal Selector	<p>The universal selector matches any element within the context in which it's placed in a selector. In the example below, the * character is the universal selector:</p> <pre data-bbox="430 1612 1208 1837">/* the asterisk character is the universal selector */ .navigation ul * { width: 100px; float: left; }</pre> <p>Universal selectors are generally discouraged for performance reasons.</p>
	<p>An attribute selector selects an element to style based on an attribute and/or attribute value. The example below targets certain paragraph elements based on the existence of a "style" attribute:</p>

	<pre>/* matches elements with style="[anything]" */ p[style] { color: #1e1e1e; }</pre>
Attribute Selector	<p>The example below targets certain input elements based on the existence of a type attribute with a value of “text”:</p> <pre>/* matches input elements with type="text" */ input[type="text"] { border: solid 1px #ccc;</pre>
	<p>In each example above, everything before the first curly brace is an attribute selector.</p>
Pseudo-class	<p>A pseudo-class works similarly to a regular CSS class, except it's not explicitly declared in the HTML. In the example below, the pseudo-class is added to the anchor element:</p>
Pseudo-element	<p>A pseudo-element is not the same as a pseudo-class. While a pseudo-class matches elements that actually exist, pseudo-elements target “virtual” elements that can change depending on the actual HTML. CSS2 pseudo-elements use a single colon and CSS3 pseudo-elements use a double colon.</p> <pre>/* "first-letter" including the preceding colon is the pseudo-element */ p:first-letter { display: block; float: left; margin: 0 5px 5px 0; } /* CSS3 pseudo-elements have double colons */ ::selection { background: green; }</pre>
	<p>A combinator is the character in a selector that connects two selectors together. There are four types of combinators. These four combinators help create descendant selectors (with a space character), child selectors (with the “>” character), adjacent sibling selectors (with the “+”</p>

Combinator	<p>character), and general sibling selectors (with the “~” character). To dispel any confusion, here are those four combinator in use:</p> <pre> /* In all 4 examples */ /* whatever appears between "div" and "p" is a combinator */ /* in the first example, the combinator is a space character */ div p { color: #222; } div>p { color: #333; } div+p { color: #444; } div~p { color: #555; } </pre>
At-rule	<p>An at-rule is an instruction given in a CSS document using the <code>@</code> character. An at-rule could have a declaration block or a simple string of text. The example below has two different at-rules:</p> <pre> @import url(secondar.css); @media print { #container { width: 500px; } } </pre>
Statement	<p>The at-rule is not just the <code>@media</code> or <code>@import</code> part at the beginning; the entire instruction comprises the complete at-rule.</p> <p>A statement in CSS is any at-rule or rule set. In the example below, there are two statements; one is an at-rule, and the other is a rule set:</p> <pre> /* the at-rule below is a statement */ @import url(secondar.css); /* the entire rule-set below is a statement */ body { font-family: Arial, sans-serif; color: #555; font-size: 14px; } </pre>

Identifier	<p>An identifier can be anything that appears as a property, id, class, keyword value, and at-rule. In the example, below there are four identifiers:</p> <pre>/* "body", "background", "none", and "font-size" are identifiers */ body { background: none; font-size: 14px; } /* "14px" is not an identifier */</pre>
Keyword	<p>A keyword is a value for a property and is somewhat like a reserved word for a particular property. Different properties have different keywords, and all properties allow the keyword “inherit”. In the example below, the value “auto” is a keyword.</p> <pre>#container { height: auto; /* "auto" is a keyword */ }</pre> <p>As an aside, I would argue that !important qualifies as a keyword, based on how keywords are differentiated from values.</p>

Selectors

Table of Content

1. General
 - o Basic Selector
 - o Child Selector
 - o Attribute Selector
 - o Universal Selector
2. Pseudo-classes
3. Pseudo-elements
4. References

General

Basic Selector

Two attributes have special status for CSS. They are `class` and `id`.

Class selectors

Use the class attribute in an element to assign the element to a named class. It is up to you what name you choose for the class. Multiple elements in a document can have the same class value.

In your stylesheet, type a full stop `.` (period) before the class name when you use it in a selector.

ID selectors

Use the id attribute in an element to assign an ID to the element. It is up to you what name you choose for the ID. The ID name must be unique in the document.

In your stylesheet, type a number sign `#` (hash) before the ID when you use it in a selector.

Example:

This HTML tag has both a class attribute and an id attribute:

```
<p class="key" id="principal">
```

The `id` value, *principal*, **must be unique in the document**, but other tags in the document can have the same `class` name, *key*.

In a CSS stylesheet, this rule makes all the elements with class *key* green. (They might not all be `<p>` elements.)

```
.key {  
    color: green;  
}
```

This rule makes the one element with the id *principal* bold:

```
#principal {  
    font-weight: bolder;  
}
```

Child Selector

This selector matches all elements that are the immediate children of a specified element. The combinator in a child selector is a greater-than sign `>`.

Example:

Consider this HTML fragment:

```
<ul>  
    <li>Item 1</li>  
    <li>  
        <ol>  
            <li>Subitem 2A</li>  
            <li>Subitem 2B</li>  
        </ol>  
    </li>  
</ul>
```

Let's try to match elements in the above fragment with the selector below:

```
ul>li {  
    : declarations  
}
```

Common selectors based on relationships:

Selector	Selects
A E	Any E element that is a descendant of an A element (that is: a child, or a child of a child, etc.)
A > E	Any E element that is a child (i.e. direct descendant) of an A element
E:first-child	Any E element that is the first child of its parent
B + E	Any E element that is the next sibling of a B element (that is: the next child of the same parent)

You can also use the symbol * (asterisk) to mean "any element".

Attribute Selector

An attribute selector will match elements on the basis of either the presence of an attribute, or the exact or partial match of an attribute value. Attribute selectors were introduced in CSS2, and CSS3 added a few more.

Attribute selectors are delimited by square brackets; the simplest form of an attribute selector consists of an attribute name surrounded by square brackets:

```
[href] {
  : declarations
}

// 

a[href] {
  : declarations
}
```

Example:

Usage	Explanation
[disabled]	Selects all elements with a <code>disabled</code> attribute.
[type='button']	Selects elements with a <code>button</code> type.
[class~='key']	Selects elements with the class <code>key</code> (but not e.g. <code>keyed</code> , <code>monkey</code> , <code>buckeye</code>). Functionally equivalent to <code>.key</code> .
[lang =es]	Selects elements specified as Spanish. This includes <code>es</code> and <code>es-MX</code> but not <code>eu-ES</code> (which is Basque).
[title*="example" i]	Selects elements whose title contains <code>example</code> , ignoring case. In browsers that don't support the <code>i</code> flag, this selector probably won't match any element.
a[href^="https://"]	Specifies what the attribute's value should start with; in this case, it selects secure links.
img[src\$=".png"]	Selects elements whose value ends with the provided string. Indirectly selects PNG images; any images that are PNGs but whose URL doesn't end in <code>.png</code> (e.g. <code>src="some-image.png?_=cachebusterhash"</code>) won't be selected.

More on Attribute Selector

Case Sensitivity: The value specified in an attribute selector is case sensitive if the attribute value in the markup language is case sensitive. Thus, values for id and class attributes in HTML are case sensitive, while values for lang and type attributes are not.

Default Attributes: Attribute selectors can only match attributes and values that exist in the document tree, and there's no guarantee that a default value specified in a DTD (or similar) can be matched. For instance, in HTML, the default value for a form element's `method` attribute is `"get"`, but you can't rely on a selector like `form[method="get"]` to select a form element with the start tag `<form action="comment.php">`.

Attribute Selectors for IDs: Note that `[id="foo"]` isn't equivalent to `#foo`. Although both selectors would match the same element in HTML, there's a significant difference between their levels of specificity.

Hyphen or No Hyphen?: All supporting browsers allow a hyphen to appear in the value in a selector like `[hreflang|="en"]`. It's unclear whether or not this is illegal, because the CSS specification doesn't contain any guidelines to help us deal with this situation.

Universal Selector

The universal selector matches any element type. It can be implied (and therefore omitted) if it isn't the only component of the simple selector.

Example: The two selector examples shown here are equivalent:

```
*.warning {
  : declarations
}

.warning {
  : declarations
}
```

It's important not to confuse the universal selector with a wildcard character—the universal selector doesn't match "zero or more elements".

Example:

```
<body>
  <div>
    <h1>The <em>Universal</em> Selector</h1>
    <p>We must <em>emphasize</em> the following:</p>
    <ul>
      <li>It's <em>not</em> a wildcard.</li>
      <li>It matches elements regardless of <em>type</em>. </li>
    </ul>
    This is an <em>immediate</em> child of the division.
  </div>
</body>
```

The selector `div * em` will match the following em elements:

- "Universal" in the `h1` element (* matches the `<h1>`)
- "emphasize" in the `p` element (* matches the `<p>`)
- "not" in the first `li` element (* matches the `` or the ``)
- "type" in the second `li` element (* matches the `` or the ``)

However, it won't match the `immediate` element, since that's an immediate child of the `div` element—there's nothing between `<div>` and `` for the `*` to match.

Pseudo-classes

A CSS pseudo-class is a keyword added to selectors that specifies a special state of the element to be selected. For example `:hover` will apply a style when the user hovers over the element specified by the selector.

Pseudo-classes can be used to:

- Style an element when a user mouses over it.

- Style visited and unvisited links differently.
- Style an element when it gets focus.

Syntax:

```
selector:pseudo-class {  
    property:value;  
}
```

A list of available pseudo-classes in [References](#) section.

Pseudo-elements

Just like pseudo-classes, pseudo-elements are added to selectors but instead of describing a special state, they allow you to style certain parts of a document. For example, the `::first-line` pseudo-element targets only the first line of an element specified by the selector.

Pseudo-elements can be used to:

- Style the first letter, or line, of an element.
- Insert content before, or after, the content of an element.

Syntax:

```
selector::pseudo-element {  
    property: value;  
}
```

A list of available pseudo-elements in [References](#) section.

References

General

Selector	Example	Example description	CSS
.class	.intro	Selects all elements with class="intro"	1
#id	#firstname	Selects the element with id="firstname"	1
*	*	Selects all elements	2
element	p	Selects all <p> elements	1
element,element	div, p	Selects all <div> elements and all <p> elements	1
element element	div p	Selects all <p> elements inside <div> elements	1
element>element	div > p	Selects all <p> elements where the parent is a <div> element	2
element+element	div + p	Selects all <p> elements that are placed immediately after <div> elements	2
element1~element2	p ~ ul	Selects every element that are preceded by a <p> element	3
[attribute]	[target]	Selects all elements with a target attribute	2
[attribute=value]	[target=_blank]	Selects all elements with target="_blank"	2
[attribute~=value]	[title~=flower]	Selects all elements with a title attribute containing the word "flower"	2
[attribute =value]	[lang =en]	Selects all elements with a lang attribute value starting with "en"	2
[attribute^=value]	a[href^="https"]	Selects every <a> element whose href attribute value begins with "https"	3
[attribute\$=value]	a[href\$=".pdf"]	Selects every <a> element whose href attribute value ends with ".pdf"	3
[attribute*=value]	a[href*="w3schools"]	Selects every <a> element whose href attribute value contains the substring "w3schools"	3

Pseudo-classes

Selector	Example	Example description
:active	a:active	Selects the active link
:checked	input:checked	Selects every checked <code><input></code> element
:disabled	input:disabled	Selects every disabled <code><input></code> element
:empty	p:empty	Selects every <code><p></code> element that has no children
:enabled	input:enabled	Selects every enabled <code><input></code> element
:first-child	p:first-child	Selects every <code><p></code> elements that is the first child of its parent
:first-of-type	p:first-of-type	Selects every <code><p></code> element that is the first <code><p></code> element of its parent
:focus	input:focus	Selects the <code><input></code> element that has focus
:hover	a:hover	Selects links on mouse over
:in-range	input:in-range	Selects <code><input></code> elements with a value within a specified range
:invalid	input:invalid	Selects all <code><input></code> elements with an invalid value
:lang(language)	p:lang(it)	Selects every <code><p></code> element with a lang attribute value starting with "it"
:last-child	p:last-child	Selects every <code><p></code> elements that is the last child of its parent
:last-of-type	p:last-of-type	Selects every <code><p></code> element that is the last <code><p></code> element of its parent
:link	a:link	Selects all unvisited links
:not(selector)	:not(p)	Selects every element that is not a <code><p></code> element
:nth-child(n)	p:nth-child(2)	Selects every <code><p></code> element that is the second child of its parent
:nth-last-child(n)	p:nth-last-child(2)	Selects every <code><p></code> element that is the second child of its parent, counting from the last child
:nth-last-of-type(n)	p:nth-last-of-type(2)	Selects every <code><p></code> element that is the second <code><p></code> element of its parent, counting from the last child
:nth-of-type(n)	p:nth-of-type(2)	Selects every <code><p></code> element that is the second <code><p></code> element of its parent
		Selects every <code><p></code> element that is the only

		<p> element of its parent
:only-child	p:only-child	Selects every <p> element that is the only child of its parent
:optional	input:optional	Selects <input> elements with no "required" attribute
:out-of-range	input:out-of-range	Selects <input> elements with a value outside a specified range
:read-only	input:read-only	Selects <input> elements with a "readonly" attribute specified
:read-write	input:read-write	Selects <input> elements with no "readonly" attribute
:required	input:required	Selects <input> elements with a "required" attribute specified
:root	root	Selects the document's root element
:target	#news:target	Selects the current active #news element (clicked on a URL containing that anchor name)
:valid	input:valid	Selects all <input> elements with a valid value
:visited	a:visited	Selects all visited links

Pseudo-elements

Selector	Example	Example description
::after	p::after	Insert content after every <p> element
::before	p::before	Insert content before every <p> element
::first-letter	p::first-letter	Selects the first letter of every <p> element
::first-line	p::first-line	Selects the first line of every <p> element
::selection	p::selection	Selects the portion of an element that is selected by a user

Properties & Values

Table of Content

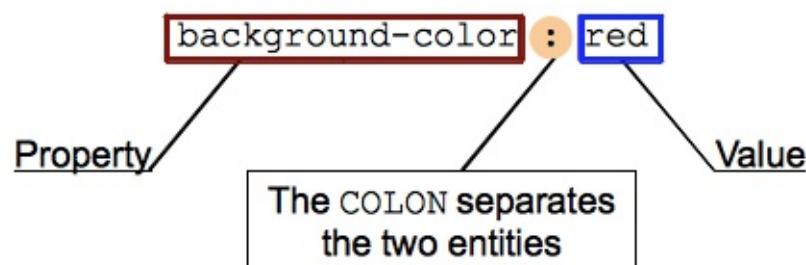
1. What is property?
2. Common Properties
3. Shorthand Properties
4. Properties Values
5. References

What is property?

Properties are defined within selectors by defining a property and a value. They are separated with a colon and delineated with a semi-colon.

CSS properties, associated with values, form declarations. Declaration blocks, assigned with selectors, form rule sets. They are used as follows:

A CSS declaration :



Syntax:

```
// Single
selector {
    property: value;
}

// Multiple
selector {
    property-1: value-1;
    property-2: value-2-1 value-2-2;
}
```

Example:

```
#box {
    width: 200px; /* the property is "width" (without the colon) */
}
```

Common Properties

Text and Fonts

Property	Description	Usage
text-align	The alignment of a block of text and other inline boxes.	text-align: left OR right OR center OR justify OR initial OR inherit;
text-decoration	Underlined, overlined, and struck-through text.	text-decoration: none OR underline OR overline OR line-through OR initial OR inherit;
text-transform	Converts the case of letters — to uppercase, lowercase, or capitalized.	text-transform: none OR capitalize OR uppercase OR lowercase OR initial OR inherit;
vertical-align	The vertical alignment of an inline box or text, or table cell.	vertical-align: baseline OR length OR sub OR super OR top OR text-top OR middle OR bottom OR text-bottom OR initial OR inherit;
white-space	How white space (such as new lines or a sequence of spaces) should be handled.	white-space: normal OR nowrap OR pre OR pre-line OR pre-wrap OR initial OR inherit;
font	Font characteristics. Some, or all, of font name, font size, boldness, italics, small-caps, and line-height.	font: font-style font-variant font-weight font-size/line-height font-family OR caption OR icon OR menu OR message-box OR small-caption OR status-bar OR initial OR inherit;
font-family	Font name. This can be a single font or a comma-separated list, of which a browser will apply the first font it can use (such as one installed on a user's computer).	font-family: family-name/generic-family OR initial OR inherit;
font-size	Font size. Size of a font. Font? The size of it.	font-size: medium OR xx-small OR x-small OR small OR large OR x-large OR xx-large OR smaller OR larger OR length OR initial OR inherit;
font-weight	Bold or light text.	font-weight: normal OR bold OR bolder OR lighter OR number OR initial OR inherit;
font-style	Italic or oblique text.	font-style: normal OR italic OR oblique OR initial OR inherit;

Colors and Backgrounds

Property	Description	Usage
color	Foreground color, primarily setting text color.	color: color OR initial OR inherit;
opacity	Transparency	number OR initial OR inherit;
background	Background color and image characteristics.	background: bg-color bg-image position/bg-size bg-repeat bg-origin bg-clip bg-attachment initial OR inherit;
background-color	The background color of a box.	background-color: color OR transparent OR initial OR inherit;
background-image	The background image of a box.	background-image: url OR none OR initial OR inherit;

The Box Model

Property	Description	Usage
border	The width, style, and color of all four sides of the border of a box.	border: border-width border-style border-color OR initial OR inherit;
border-radius	Rounded corners.	border-radius: 1-4 length OR % / 1-4 length OR % OR initial OR inherit;
box-shadow	Drop-shadows! For boxes!	box-shadow: none OR h-shadow v-shadow blur spread color OR inset OR initial OR inherit;
margin	The margin of a box, outside the border, padding, and content areas.	margin: length OR auto OR initial OR inherit;
padding	The padding of a box, inside the margin and border areas, and outside the content area.	padding: length OR initial OR inherit;

Visual Formatting

Property	Description	Usage
width	The width of the content area of a box.	width: auto OR value OR initial OR inherit;
height	The height of the content area of a box.	height: auto OR length OR initial OR inherit;
min-width	The minimum width of the content area of a box.	min-width: length OR initial OR inherit;
max-width	The maximum width of the content area of a box.	max-width: length OR initial OR inherit;
min-height	The minimum height of the content area of a box.	min-height: length OR initial OR inherit;
max-height	The maximum height of the content area of a box.	max-height: length OR initial OR inherit;
position	How a box's position is calculated.	position: static OR absolute OR fixed OR relative OR initial OR inherit;
top	The top position of a box.	top: auto OR length OR initial OR inherit;
right	The right position of a box.	right: auto OR length OR initial OR inherit;
bottom	The bottom position of a box.	bottom: auto OR length OR initial OR inherit;
left	The left position of a box.	left: auto OR length OR initial OR inherit;
overflow	Determines what should happen to a box's overflow — the portions of content that do not fit inside the box.	overflow: visible OR hidden OR scroll OR auto OR initial OR inherit;
z-index	The placement of a positioned box in the z-axis.	z-index: auto OR number OR initial OR inherit;
float	How a box should float, shifting it to the right or left with surrounding content flowing around it.	float: none OR left OR right OR initial OR inherit;
clear	How a box following a floated box should behave.	clear: none OR left OR right OR both OR initial OR inherit;
display	The fundamental display characteristics of a box.	display: value;
visibility	The visibility, or invisibility, of a box.	visibility: visible OR hidden OR collapse OR initial OR inherit;

Generated Content and Lists

Property	Description	Usage
content	Generated content, placed before or after the content of a box.	content: normal OR none OR counter OR attr OR string OR open-quote OR close-quote OR no-open-quote OR no-close-quote OR url OR initial OR inherit;
counter-increment	Increments a named counter used by generated content.	counter-increment: none OR id OR initial OR inherit;
counter-reset	Resets a named counter used by generated content.	counter-reset: none OR name number OR initial OR inherit;
list-style	The type, image, and position of a list item marker.	list-style: list-style-type list-style-position list-style-image OR initial OR inherit;

Transitions and Transformation

Property	Description	Usage
transition	A gradual change of appearance over time. Allows a link to animate from one color to another when it is hovered over, for example.	transition: property duration timing-function delay OR initial OR inherit;
transform	Manipulates the size, shape, and position of a box and its contents through rotating, skewing, scaling and translating.	transform: none OR transform-functions OR initial OR inherit;

Misc

Property	Description	Usage
cursor	The appearance of the cursor when it passes over a box.	cursor: value;
outline	The width, style, and color of the outline around a box.	outline: outline-color outline-style outline-width OR initial OR inherit;
direction	Writing direction.	direction: ltr OR rtl OR initial OR inherit;

Shorthand Properties

Shorthand properties are CSS properties that let you set the values of several other CSS properties simultaneously. Using a shorthand property, a Web developer can write more concise and often more readable style sheets, saving time and energy.

Edges and Corners

- Shorthands handling properties related to edges of a box, like `border-style`, `margin` or `padding`, always use a consistent **1-to-4-value** syntax representing those edges:

Syntax

```
border: border-width | border-style | color;
// border-top, border-right, border-bottom, border-left apply the same model.

border-radius: 1-4 length|% / 1-4 length|%|initial|inherit;

margin: margin-top | margin-right | margin-bottom | margin-left;

padding: padding-top | padding-right | padding-bottom | padding-left;
```

Example:

Case	Usage
	The 1-value syntax: <code>border-width: 1em</code> — The unique value represents all edges.
	The 2-value syntax: <code>border-width: 1em 2em</code> — The first value represents the vertical, that is top and bottom, edges, the second the horizontal ones, that is the left and right ones.
	The 3-value syntax: <code>border-width: 1em 2em 3em</code> — The first value represents the top edge, the second, the horizontal, that is left and right, ones, and the third value the bottom edge.
	The 4-value syntax: <code>border-width: 1em 2em 3em 4em</code> — The four values represent the top, right, bottom and left edges respectively, always in that order, that is clock-wise starting at the top (The initial letter of Top-Right-Bottom-Left matches the order of the consonant of the word trouble: TRBL).

- Shorthands handling properties related to corners of a box, like `border-radius`, always use a consistent **1-to-4-value** syntax representing those corners:

Case	Usage
	The 1-value syntax : <code>border-radius: 1em</code> — The unique value represents all corners.
	The 2-value syntax : <code>border-radius: 1em 2em</code> — The first value represents the top left and bottom right corner, the second the top right and bottom left ones.
	The 3-value syntax : <code>border-radius: 1em 2em 3em</code> — The first value represents the top left corner, the second the top right and bottom left ones, and the third value the bottom right corner.
	The 4-value syntax : <code>border-radius: 1em 2em 3em 4em</code> — The four values represent the top left, top right, bottom right and bottom left corners respectively, always in that order, that is clock-wise starting at the top left.

Background properties

Syntax:

```
background: background-color | background-image | background-repeat | background-attachment | background-position;
```

Example: A background with the following properties:

```
background-color: #000;
background-image: url(images/bg.gif);
background-repeat: no-repeat;
background-position: top right;
```

Can be shortened to just one declaration:

```
background: #000 url(images/bg.gif) no-repeat top right;
```

Font properties

Syntax

```
font: font-style | font-variant | font-weight | font-size | line-height | font-family;
```

Example: The following declarations:

```
font-style: italic;  
font-weight: bold;  
font-size: .8em;  
line-height: 1.2;  
font-family: Arial, sans-serif;
```

Can be shortened to the following:

```
font: italic bold .8em/1.2 Arial, sans-serif;
```

Shorthand properties: background , font , margin , border , border-top , border-right , border-bottom , border-left , border-width , border-color , border-style , transition , animation , transform , padding , list-style , border-radius .

Properties Values

Distance

- Length: absolute (px, in, pc, pt, cm, mm, q) and relative (em, ex, ch, rem, vw, vh, vmin, vmax)

Numeric

- Percentage: %
- Number: Real number. Using decimal numerals 0–9, this can be a whole number (see integer) or may include numerals following a decimal point. A number may be positive or negative.
- Integer: A whole number, consisting of decimal numerals 0–9. An integer may be positive or negative.

Textual

- String: Quoted string. Any sequence of characters in between single or double quotes.
- URL: url()
- inherit
- initial
- unset

Others

- Color: `#`, `rgb`, `rgba`, `hsl`, `hsla`, `transparent`, other keywords
- Resolution: `dpi`, `dpcm`, `dppx`
- Angle: `deg`, `grad`, `rad`, `turn`
- Time: `s`, `ms`
- Frequency: `Hz`, `kHz`
- Ratio: Valid ratios include `16/9`, `4/3`, `800/600`, and `1/1`.

References

More common properties you should know about: `background`, `background-attachment`, `background-color`, `background-image`, `background-position`, `background-repeat`, `border`, `border-bottom`, `border-bottom-color`, `border-bottom-style`, `border-bottom-width`, `border-color`, `border-left`, `border-left-color`, `border-left-style`, `border-left-width`, `border-right`, `border-right-color`, `border-right-style`, `border-right-width`, `border-style`, `border-top`, `border-top-color`, `border-top-style`, `border-top-width`, `border-width`, `clear`, `clip`, `color`, `cursor`, `display`, `filter`, `font`, `font-family`, `font-size`, `font-variant`, `font-weight`, `height`, `left`, `letter-spacing`, `line-height`, `list-style`, `list-style-image`, `list-style-position`, `list-style-type`, `margin`, `margin-bottom`, `margin-left`, `margin-right`, `margin-top`, `overflow`, `padding`, `padding-bottom`, `padding-left`, `padding-right`, `padding-top`, `page-break-after`, `page-break-before`, `position`, `float`, `text-align`, `text-decoration`, `text-indent`, `text-transform`, `top`, `vertical-align`, `visibility`, `width`, `z-index`.

Properties values that matter:

Keyword	Associated type	Comment
<code>string</code>	<code><string></code>	The attribute value is treated as a CSS <code><string></code> . It is NOT reparsed, and in particular the characters are used as-is instead of CSS escapes being turned into different characters.
<code>color</code>	<code><color></code>	The attribute value is parsed as a hash (3- or 6-value hash) or a keyword. It must be a valid CSS <code><string></code> value. Leading and trailing spaces are stripped.
<code>url</code>	<code><uri></code>	The attribute value is parsed as a string that is used inside a CSS <code>url()</code> function. Relative URL are resolved relatively to the original document, not relatively to the style sheet. Leading and trailing spaces are stripped.
<code>integer</code>	<code><integer></code>	The attribute value is parsed as a CSS <code><integer></code> . If it is not valid, that is not an integer or out of the range accepted by the CSS property, the default value is

		used. Leading and trailing spaces are stripped.
number	<number>	The attribute value is parsed as a CSS <number> . If it is not valid, that is not a number or out of the range accepted by the CSS property, the default value is used. Leading and trailing spaces are stripped.
length	<length>	The attribute value is parsed as a CSS <length> dimension, that is including the unit (e.g. 12.5em). If it is not valid, that is not a length or out of the range accepted by the CSS property, the default value is used. If the given unit is a relative length, attr() computes it to an absolute length. Leading and trailing spaces are stripped.
em, ex, px, rem, vw, vh, vmin, vmax, mm, cm, in, pt, or pc	<length>	The attribute value is parsed as a CSS <number> , that is without the unit (e.g. 12.5), and interpreted as a with the specified unit. If it is not valid, that is not a number or out of the range accepted by the CSS property, the default value is used. If the given unit is a relative length, attr() computes it to an absolute length. Leading and trailing spaces are stripped.
angle	<angle>	The attribute value is parsed as a CSS <angle> dimension, that is including the unit (e.g. 30.5deg). If it is not valid, that is not an angle or out of the range accepted by the CSS property, the default value is used. Leading and trailing spaces are stripped.
deg, grad, rad	<angle>	The attribute value is parsed as a CSS , that is without the unit (e.g. 12.5), and interpreted as an <angle> with the specified unit. If it is not valid, that is not a number or out of the range accepted by the CSS property, the default value is used. Leading and trailing spaces are stripped.
time	<time>	The attribute value is parsed as a CSS <time> dimension, that is including the unit (e.g. 30.5ms). If it is not valid, that is not a time or out of the range accepted by the CSS property, the default value is used. Leading and trailing spaces are stripped.
s, ms	<time>	The attribute value is parsed as a CSS <number> , that is without the unit (e.g. 12.5), and interpreted as an <time> with the specified unit. If it is not valid, that is not a number or out of the range accepted by the CSS property, the default value is used. Leading and trailing spaces are stripped.
frequency	<frequency>	The attribute value is parsed as a CSS <frequency> dimension, that is including the unit (e.g. 30.5kHz). If it is not valid, that is not a frequency or out of the range accepted by the CSS property, the default value is used.
		The attribute value is parsed as a CSS <number> , that

Hz, kHz	<frequency>	is without the unit (e.g. 12.5), and interpreted as a <frequency> with the specified unit. If it is not valid, that is not a number or out of the range accepted by the CSS property, the default value is used. Leading and trailing spaces are stripped.
%	<percentage>	The attribute value is parsed as a CSS <number>, that is without the unit (e.g. 12.5), and interpreted as a %. If it is not valid, that is not a number or out of the range accepted by the CSS property, the default value is used. If the given value is used as a length, attr() computes it to an absolute length. Leading and trailing spaces are stripped.

For all available and in draft properties, please refer to

<https://www.w3.org/Style/CSS/all-properties.en.html>

Functions

Table of Content

1. [What are CSS Functions?](#)

2. [Common Functions](#)

- o [attr\(\)](#)
- o [calc\(\)](#)
- o [url\(\)](#)

3. [References](#)

- o [Further Reading](#)
- o [Functions Index](#)

What are CSS Functions?

Essential Functions

attr()

The `attr()` CSS function is used to retrieve the value of an attribute of the selected element and use it in the style sheet. It can be used on pseudo-elements too and, in this case, the value of the attribute on the pseudo-element's originated element is returned.

The attr() function can be used with any CSS property, but support for properties other than content is experimental.

Syntax:

```
attr( <attr-name> <type-or-unit>? [, <attr-fallback> ]? )  
  
where  
<type-or-unit> = string | integer | color | url | integer | number | length | angle |  
time | frequency | em | ex | px | rem | vw | vh | vmin | vmax | mm | cm | in | pt | pc  
| deg | grad | rad | ms | s | Hz | kHz | %
```

Usage:

```
/* Simple usage */
attr(data-count);
attr(title);

/* With type */
attr(src url);
attr(data-count number);
attr(data-width px);

/* With fallback */
attr(data-count number, 0);
attr(src url, '');
attr(data-width px, inherit);
attr(data-something, 'default');
```

Examples: In CSS

```
p::before {
  content: attr(data-foo) " ";
}
```

In HTML

```
<p data-foo="hello">world</p>
```

Result should be

```
hello world
```

calc()

The `calc()` CSS function can be used anywhere a `<length>` , `<frequency>` , `<angle>` , `<time>` , `<number>` , or `<integer>` is required. With `calc()` , you can perform calculations to determine CSS property values.

It is possible to nest `calc()` function, the internal ones being considered as simple parenthesis `()` .

Syntax:

```
calc( <calc-sum> )

where
<calc-sum> = <calc-product> [ [ '+' | '-' ] <calc-product> ]*

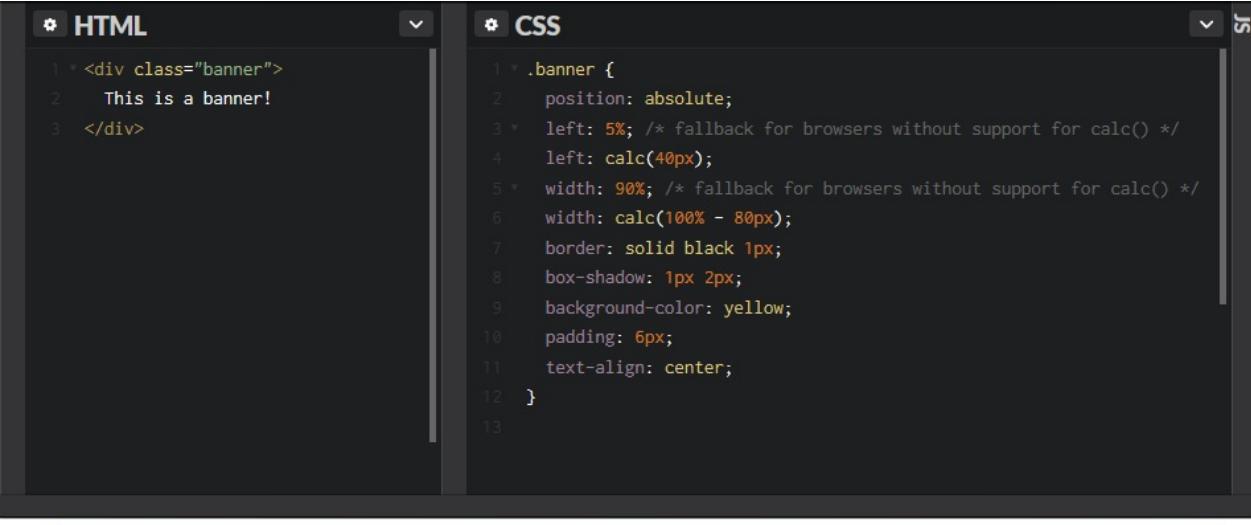
where
<calc-product> = <calc-value> [ '*' <calc-value> | '/' <number> ]*

where
<calc-value> = <number> | <dimension> | <percentage> | ( <calc-sum> )
```

Usage:

```
/* property: calc(expression) */
width: calc(100% - 80px);
```

Example:



The screenshot shows a code editor with two tabs: 'HTML' and 'CSS'. The 'HTML' tab contains the following code:

```
<div class="banner">
  This is a banner!
</div>
```

The 'CSS' tab contains the following code:

```
.banner {
  position: absolute;
  left: 5%; /* fallback for browsers without support for calc() */
  left: calc(40px);
  width: 90%; /* fallback for browsers without support for calc() */
  width: calc(100% - 80px);
  border: solid black 1px;
  box-shadow: 1px 2px;
  background-color: yellow;
  padding: 6px;
  text-align: center;
}
```

Below the code editor, there is a preview window showing a yellow rectangular box with the text "This is a banner!" centered inside it.

url()

The `<url>` CSS data type denotes a pointer to a resource. It has no proper syntax and can only be expressed through the `url()` functional notation.

The URL may be unquoted, or quoted by single or double quotes. Relative URLs are allowed and are relative to the URL of the stylesheet (not to the URL of the web page).

Syntax:

```
<a_css_property>: url("http://mysite.example.com/mycursor.png")  
  
<a_css_property>: url(http://mysite.example.com/mycursor.png)
```

Example:

```
.topbanner {  
    background: url("topbanner.png") #00D no-repeat fixed;  
}  
  
ul {  
    list-style: square url(http://www.example.com/redball.png);  
}
```

References

Further Reading

Functions Index (*including stable and experimental*)

- `annotation()`
- `attr()`
- `blur()`
- `brightness()`
- `calc()`
- `contrast()`
- `drop-shadow()`
- `element()`
- `fit-content()`
- `format()`
- `grayscale()`
- `hsl()`
- `hsla()`
- `hue-rotate()`
- `image()`
- `invert()`
- `linear-gradient()`
- `local()`
- `matrix()`
- `matrix3d()`

Functions

- `minmax()`
- `opacity()`
- `perspective()`
- `radial-gradient()`
- `repeating-linear-gradient()`
- `repeating-radial-gradient()`
- `rgb()`
- `rgba()`
- `rotate()`
- `rotate3d()`
- `rotateX()`
- `rotateY()`
- `rotateZ()`
- `saturate()`
- `scale()`
- `scale3d()`
- `scaleX()`
- `scaleY()`
- `scaleZ()`
- `sepia()`
- `skew()`
- `skewX()`
- `skewY()`
- `symbols()`
- `translate()`
- `translate3d()`
- `translateX()`
- `translateY()`
- `translateZ()`
- `url()`
- `var()`

@ Rules

Table of Content

1. What are @ rules?

2. Essential Rules

- o @charset
- o @import
- o @font-face
- o @media

3. References

- o Further Reading
- o Rules Index

What are @ rules?

The at-rule is a statement that provides CSS with instructions to perform or how to behave. Each statement beginning with an at sign, @, followed by an identifier and includes everything up to the next semi-colon, ; , or the next CSS block, whichever comes first.

General syntax:

```
@[KEYWORD] (RULE);
```

Essential Rules

@charset

This rule defines the character set used by the browser. It comes in handy if the stylesheet contains non-ASCII characters (e.g. UTF-8). Note that a character set that is placed on the HTTP header will override any @charset rule. If several @charset at-rules are defined, only the first one is used, and it cannot be used inside a style attribute on an HTML element or inside the <style> element where the character set of the HTML page is relevant.

Syntax:

```
@charset "<charset>";
```

where:

<charset>

Is a <string> denoting the character encoding to be used. It must be the name of a web-safe character encoding defined in the IANA-registry. If several names are associated with an encoding, only the one marked with preferred must be used.



Example:

```
@charset "UTF-8";      /* Set the encoding of the style sheet to Unicode UTF-8 */
@charset 'iso-8859-15'; /* Set the encoding of the style sheet to Latin-9 (Western European languages, with euro sign) */
@charset "UTF-8";      /* Invalid, there is a character (a space) before the at-rule */
*/
@charset UTF-8;        /* Invalid, without ' or ", the charset is not a CSS <string> */
*/
```

@import

This rule is inserted at the very top of the file and instructs the stylesheet to request and include an external CSS file as if the contents of that file were right where that line is.

Syntax:

```
@import url;
@import url list-of-media-queries;
```

where:

url

Is a <string> or a <uri> representing the location of the resource to import. The URL may be absolute or relative. Note that the URL need not actually specify a file; it can just specify the package name and part, and the appropriate file is chosen automatically (e.g. chrome://communicator/skin/). See here for more information.

list-of-media-queries

Is a comma-separated list of media queries conditioning the application of the CSS rules defined in the linked URL. If the browser does not support any these queries, it does not load the linked resource.

Example:

```
@import url("fineprint.css") print;
@import url("bluish.css") projection, tv;
@import 'custom.css';
@import url("chrome://communicator/skin/");
@import "common.css" screen, projection;
@import url('landscape.css') screen and (orientation:landscape);
```

User agents can avoid retrieving resources for unsupported media types, authors may specify media-dependent `@import` rules. These conditional imports specify comma-separated media queries after the URI. In the absence of any media query, the import is unconditional. Specifying `all` for the medium has the same effect.

@font-face

This rule allows us to load custom fonts on a webpage. There are varying levels of support for custom fonts, but this rule accepts statements that create and serve those fonts. By allowing authors to provide their own fonts, `@font-face` eliminates the need to depend on the limited number of fonts users have installed on their computers.

Syntax:

```
@font-face {
  [ font-family: <family-name>; ] ||
  [ src: [ <uri> [ format(<string>#) ]? | <font-face-name> ]#; ] ||
  [ unicode-range: <urange>#; ] ||
  [ font-variant: <font-variant>; ] ||
  [ font-feature-settings: normal | <feature-tag-value>#; ] ||
  [ font-stretch: <font-stretch>; ] ||
  [ font-weight: <weight>; ] ||
  [ font-style: <style>; ]
}

font-family
  Specifies a name that will be used as the font face value for font properties.

src
  Specifies the resource containing the font data. This can be a URL to a remote font file location or the name of a font on the user's computer.

font-variant
  A font-variant value.

font-stretch
  A font-stretch value.

font-weight
  A font-weight value.

font-style
  A font-style value.

unicode-range
  The range of Unicode code points to be used from the font.
```

Example: In CSS

```

/* Top-level */
@font-face {
    font-family: MyHelvetica;
    src: local("Helvetica Neue Bold"),
         local("HelveticaNeue-Bold"),
         url(MgOpenModernaBold.ttf);
    font-weight: bold;
}

/* Nested */
.className {
    @font-face {
        font-family: MyHelvetica;
        src: local("Helvetica Neue Bold"),
             local("HelveticaNeue-Bold"),
             url(MgOpenModernaBold.ttf);
        font-weight: bold;
    }
}

```

In HTML

```

<html>
<head>
    <title>Web Font Sample</title>
    <style type="text/css" media="screen, print">
        @font-face {
            font-family: "Bitstream Vera Serif Bold";
            src: url("https://mdn.mozilla.org/files/2468/VeraSeBd.ttf");
        }

        body { font-family: "Bitstream Vera Serif Bold", serif }
    </style>
</head>
<body>
    This is Bitstream Vera Serif Bold.
</body>
</html>

```

Notes:

- Web fonts are subject to the same domain restriction (font files must be on the same domain as the page using them), unless HTTP access controls are used to relax this restriction.
- Because there are no defined MIME types for TrueType, OpenType, and Web Open File Format (WOFF) fonts, the MIME type of the file specified is not considered.

- `@font-face` cannot be declared within a CSS selector. For example, the following will not work:

@media

This rule contains conditional statements for targeting styles to specific screens. These statements can include screen sizes, which can be useful for adapting styles to devices.

Syntax:

```
@media <media-query-list> {
  <group-rule-body>
}
```

where

A `<media-query-list>` is composed of a optional media type and/or a number of media features.

Example:

```
@media print {
  body { font-size: 10pt }
}

@media screen {
  body { font-size: 13px }
}

@media screen, print {
  body { line-height: 1.2 }
}

@media only screen
  and (min-device-width: 320px)
  and (max-device-width: 480px)
  and (-webkit-min-device-pixel-ratio: 2) {
  body { line-height: 1.4 }
}
```

You will be covered more about media queries in [next chapter](#).

References

Futher Reading

Rules Index (*including stable and experimental*)

- `@charset`
- `@counter-style`
- `@document`
- `@font-face`
- `@font-feature-values`
- `@import`
- `@keyframes`
- `@media`
- `@namespace`
- `@page`
- `@supports`
- `@viewport`

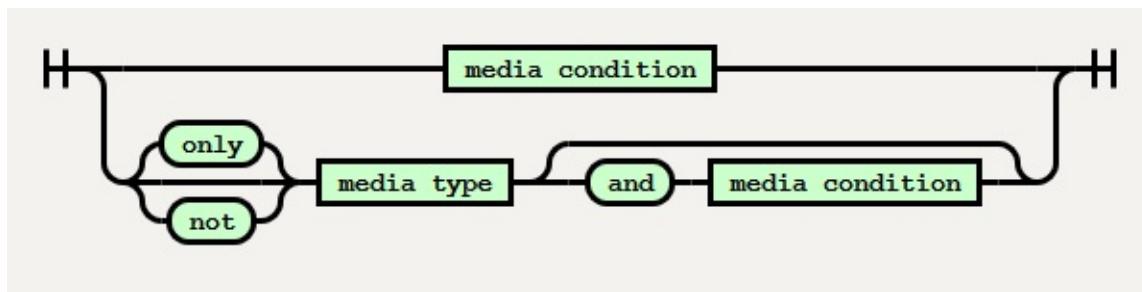
Media queries

Table of Content

1. [What are media queries?](#)
2. [Syntax](#)
3. [Media Types](#)
4. [Media Features](#)
 - [Dimensions](#)
 - [Resolution](#)
 - [Orientation / Aspect Ratio](#)
5. [References](#)
 - [Further Reading](#)
 - [Features Index](#)

What are media queries?

Media Queries is a module of CSS that defines expressions allowing to tailor presentations to a specific range of output devices without changing the content itself.



Syntax

```
<!-- CSS media query on a link element -->
<link rel="stylesheet" media="(max-width: 800px)" href="example.css" />

<!-- CSS media query within a stylesheet -->
<style>
@media (max-width: 600px) {
  .facet_sidebar {
    display: none;
  }
}
</style>
```

A basic media query, a single media feature with the implied all media type, could look like this:

```
@media (min-width: 700px) { ... }
```

Logical operators:

You can compose complex media queries using logical operators, including not, and, and only.

- The `and` operator is used for combining multiple media features together into a single media query, requiring each chained feature to return true in order for the query to be true.

```
@media (min-width: 700px) and (orientation: landscape) { ... }
/* The above media query will only return true if the viewport is 700px wide or wider and the display is in landscape. */

@media tv and (min-width: 700px) and (orientation: landscape) { ... }
/* The above media query will only apply if the media type is TV, the viewport is 700px wide or wider, and the display is in landscape. */
```

- The `not` operator is used to negate an entire media query. It applies to the whole media query and returns true if the media query would otherwise return false and cannot be used to negate an individual feature query.

```
@media not all and (monochrome) { ... }
/* evaluated like this */

@media not (all and (monochrome)) { ... }

@media not screen and (color), print and (color) { ... }
/* evaluated like this */
@media (not (screen and (color))), print and (color) { ... }
```

- The `only` operator is used to apply a style only if the entire query matches, useful for preventing older browsers from applying selected styles.

```
<link rel="stylesheet" media="only screen and (color)" href="example.css">
```

Note: Media queries are case insensitive. Media queries involving unknown media types are always false.

Media Types

Media types provided to be assisted targeting type of devices that has the website run on.

There are four main types available:

- `all` : Suitable for all devices.

```
@media all {  
    body { background: lime; }  
}
```

- `print` : Intended for paged material and for documents viewed on screen in print preview mode. Please consult the section on paged media, and the media section of the Getting Started tutorial for information about formatting issues that are specific to paged media.

```
@media print {  
    body { font-size: 10pt; }  
}
```

- `screen` : Intended primarily for color computer screens.

```
@media screen {  
    body { font-size: 13px; }  
}
```

- `speech` : Intended for speech synthesizers. Note: CSS2 had a similar media type called 'aural' for this purpose. See the appendix on aural style sheets for details.

```
@media speech {  
    img { display: none; }  
}
```

Media Features

Most media features can be prefixed with `min-` or `max-` to express "greater or equal to" or "less than or equal to" constraints. This avoids using the `<` and `>` symbols, which would conflict with HTML and XML. If you use a media feature without specifying a value, the expression resolves to true if the feature's value is non-zero.

Dimensions

You get `height` and `width`, which query against the current browser window height and width. You could use them as-is, but that would probably be rare. Both of them accept min/max prefixes, so more commonly you'd use them as `min-width`, `max-width`, `min-height`, `max-height`.

- `width` can be used to apply styles conditionally based on the width of the viewport. The `width` must be specified as a `<length>` value.

```
/* Exact width */
@media (width: 300px) {...}

/* Viewport width at least */
@media (min-width: 50em) {...}

/* Viewport width at most */
@media (max-width: 1000px) {...}
```

- `height` whose value is the viewport's height as a CSS `<length>`.

```
/* Exact height */
@media (height: 300px) {...}

/* Viewport height at least */
@media (min-height: 50em) {...}

/* Viewport height at most */
@media (max-height: 700px) {...}
```

Resolution

The `resolution` media feature describes the resolution of the output device, i.e. the density of the pixels. When querying devices with non-square pixels, in `min-resolution` queries the least-dense dimension must be compared to the specified value and in `max-resolution` queries the most-dense dimensions must be compared instead. A `resolution` (without a `min-` or `max-` prefix) query never matches a device with non-square pixels.

Example: For printers, this corresponds to the screening resolution (the resolution for printing dots of arbitrary color).

```
/* media query expresses that a style sheet is usable on devices with resolution greater than 300 dots per inch */
@media print and (min-resolution: 300dpi) {...}

/* media query expresses that a style sheet is usable on devices with resolution less than 118 dots per centimeter */
@media print and (max-resolution: 118dpcm) {...}
```

Orientation / Aspect Ratio

- `orientation` media feature is `portrait` when the value of the `height` media feature is greater than or equal to the value of the `width` media feature. Otherwise `orientation` is `landscape`.

```
@media all and (orientation: portrait) {...}
@media all and (orientation: landscape) {...}
```

- `aspect-ratio` media feature is defined as the ratio of the value of the `width` media feature to the value of the `height` media feature.

```
@media screen and (device-aspect-ratio: 16/9) {...}
```

References

Further Reading

Features Index

Value	Description
<code>aspect-ratio</code>	The ratio between the width and the height of the viewport

color	The number of bits per color component for the output device
color-index	The number of colors the device can display
device-aspect-ratio	The ratio between the width and the height of the device
device-height	The height of the device, such as a computer screen
device-width	The width of the device, such as a computer screen
grid	Whether the device is a grid or bitmap
height	The viewport height
max-aspect-ratio	The maximum ratio between the width and the height of the display area
max-color	The maximum number of bits per color component for the output device
max-color-index	The maximum number of colors the device can display
max-device-aspect-ratio	The maximum ratio between the width and the height of the device
max-device-height	The maximum height of the device, such as a computer screen
max-device-width	The maximum width of the device, such as a computer screen
max-height	The maximum height of the display area, such as a browser window
max-monochrome	The maximum number of bits per "color" on a monochrome (greyscale) device
max-resolution	The maximum resolution of the device, using dpi or dpcm
max-width	The maximum width of the display area, such as a browser window
min-aspect-ratio	The minimum ratio between the width and the height of the display area
min-color	The minimum number of bits per color component for the output device
min-color-index	The minimum number of colors the device can display
min-device-aspect-ratio	The minimum ratio between the width and the height of the device
min-device-width	The minimum width of the device, such as a computer screen
min-device-height	The minimum height of the device, such as a computer screen
min-height	The minimum height of the display area, such as a browser window
min-monochrome	The minimum number of bits per "color" on a monochrome (greyscale) device

min-resolution	The minimum resolution of the device, using dpi or dpcm
min-width	The minimum width of the display area, such as a browser window
monochrome	The number of bits per "color" on a monochrome (greyscale) device
orientation	The orientation of the viewport (landscape or portrait mode)
overflow-block	How does the output device handle content that overflows the viewport along the block axis (added in Media Queries Level 4)
overflow-inline	Can content that overflows the viewport along the inline axis be scrolled (added in Media Queries Level 4)
resolution	The resolution of the output device, using dpi or dpcm
scan	The scanning process of the output device
update-frequency	How quickly can the output device modify the appearance of the content (added in Media Queries Level 4)
width	The viewport width

Preprocessors

Table of Content

1. [What is a preprocessor?](#)
2. [Popular Preprocessors](#)
 - [Sass \(SCSS\)](#)
 - [Less](#)
 - [Stylus](#)
3. [Comparison](#)

What is a preprocessor?

Why Pre-Processing CSS?

CSS is primitive and incomplete. Building a function, reusing a definition or inheritance are hard to achieve. For bigger projects, or complex systems, maintenance is a very big problem. On the other hand, web is evolving, new specs are being introduced to HTML as well as CSS. Browsers apply these specs while they are in proposal state with their special vendor prefixes. In some cases (as in background gradient), coding with vendor specific properties become a burden. You have to add all different vendor versions for a single result.

A *preprocessor* is a program that takes one type of data and converts it to another type of data.

CSS preprocessors take code written in the preprocessed language and then convert that code into the same old css we've been writing for years. 3 of the more popular css preprocessors are [Sass](#), [LESS](#), and [Stylus](#).

Like every programming language, pre-processors have different syntax, but hopefully, not too separated. All of them support classic CSS coding and their syntax are like classic CSS. SASS and Stylus have additional styles. In SASS, you can omit curly brackets and semicolon, whereas in Stylus, you can also omit colons.

In general, every CSS Preprocessor supports following features:

- Variables

Variables were all time wanted feature for CSS. Every developer, wanted to define a base color and use it all over the CSS file, in stead of writing the hex or named color in a property each time. Same as `color`, variables needed for `width`, `font-size`, `font-family`, `borders` etc.

- **Nesting**

CSS lacks visual hierarchy while working with child selectors. You have to write selectors and their combinations in separate lines. Nesting provides a visual hierarchy as in the HTML and increases the readability.

- **Mixins**

Mixins are set of definitions that compiles according to some parameters or static rules. With them you can easily write cross-browser background gradients or CSS arrows etc.

- **Extends**

Extends are useful for sharing a generic definition with selectors rather than copying it in. All extended selectors are grouped in compiled CSS.

- **Color Operations**

All three pre-processors have color functions to play with colors. You can lighten the base color or saturate it, even you can mix two or more different colors.

- **If/Else Statements**

Control directives and expressions help to build similar style definitions according to matched conditions or variables.

- **Loops**

Loops are useful when iterating through an array or creating a series of styles as in grid widths.

- **Math**

Math operations can be used for standard arithmetic or unit conversions.

- **Imports**

Rather than using a one large file, separating your codes in small pieces is helpful for expressing your declarations and increasing maintainability and control over the codebase. You can group the similar code chunks in similar folders and import them to main css file.

Popular Preprocessors

Sass (SCSS)

Less

Stylus

Comparison

	Less	Sass	Stylus	PostCSS
Syntax				
Indented Syntax Ability to omit curly braces {} and semicolons ;	v	x	x	x SugarSS
Variables				
basic Ability to declare variables and use them later	x	x	x	x simple-vars
default Default variables are overwritten by regular ones, no matter when they are declared	v	x	x	v
lazy Variable Hoisting, variables can be declared after being used	x	v	v	v
lookup Use the value of a previously declared property elsewhere	v	v	x	x property-lookup, mathjs
scoped Restrict the scope of variables	x	x	x	x nested-vars
Variable Interpolation				
basic Use variables as parts of selectors, properties, or values	x	x	x	x simple-vars
calc Use variables as inside calc() values	x	x	x	x simple-vars
selectors				x simple-

Place a set of selectors into a variable and reuse it	x	x	x	x simple-vars
selectors nested Nest a set of selectors with parent reference below another selector	v	x	x	v
Mixins				
basic Support inclusion of mixins	x	x	x	x mixins
params Mixins that can receive parameters passed to them	x	x	x	x mixins
params-named Mixins that have named placeholders for each parameter passed to them, allows to pass parameters in any order	x	x	x	v
arguments Mixins with an unknown number of arguments passed to them	x	x	x	x mixins
Conditionals				
basic If statement within a declaration	x unusual syntax	x	x	x conditionals
ternary Ternary operator <code>x > 0 ? true : false</code>	v	x unusual syntax	x	v
property Interpolate <code>if</code> statements inside property names	v	x	x	v
selector Interpolate <code>if</code> statements inside selectors	v	x	x requires additional variable	v
Rounding Numbers				
round functions				

Use of Round, Floor, and Ceil functions to convert a decimal to an integer	x	x	x	x mathjs
custom precision Customize the precision for a particular decimal number	x	x requires custom function	x	x mathjs , simple-vars
custom global precision Customize the global precision for calculations resulting in decimal numbers	v	x	v	v
Random Number				
basic Generate a random number with upper and lower limits	x requires custom function	v3.3.0+ x	x requires custom function	x mathjs
cache Append the random number to a resource as a query string to invalidate the cache	x requires custom function	v3.3.0+ x	x requires custom function	x mathjs
color Generate different colors with a random Hue or other color components	x requires custom function	v3.3.0+ x	x requires custom function	x mathjs
Parent Reference				
append Append selector to the parent when nesting <code>&:hover</code> , <code>&.active</code>	x	x	x	x nested
extend name Extend the name of the parent selector <code>&--modifier-class</code>	x	v3.3.0+ x	x	x nested
parent Allows to set a parent of the current nested selector <code>.ie7 &</code>	x	x	x	x nested
prepend Prepend a selector, a nested <code>ol& below .test</code> returns <code>ol.test</code>	x	v3.4.0+ x requires the <code>@at-root</code> directive	x	x nested
multiple				

multiple Ability to more than one parent reference in a selector & + &	x	x	x	x nested
explosion Generate all possible permutations of selectors from a list	x	x requires additional variable	x requires additional variable	v
root Root directive/reference removes parent selectors for everything below it	v	v3.3.0+ x	v0.42.0+ x	x simple-vars , atroot
Placeholder Selectors				
basic Reusable components that are outputted only if used somewhere, also known as silent classes	x	x	x	x extend
Content Directive				
basic Mixins accept style declarations inside it, useful if reusing selectors	x	v3.2.0+ x	v0.41.0+ x	x mixins , nested
media Mixins accept both style declarations and parameters, useful for simpler media queries	x	v3.2.0+ x	v0.41.0+ x	x mixins
placeholder text Ability to reuse the content block in several selectors, useful for placeholder-text mixin	x	v3.2.0+ x	v0.41.0+ x	x mixins
Loops				
basic Loops that can increment values	x need to call the function	x	x	x for , mathjs
intermediate Loops that iterate though items in a list	x need to call the function	x	x	x each
Image Helpers				

dimensions Ability to get the width/height of a given image	v2.2.0+ x	x compass	x	x assets
inline Encode a given image to base64	v1.4.0+ x	x compass	x requires function definition	x assets
Sourcemaps				
basic Generate a source map file that defines a mapping between CSS declarations and the corresponding line of the source file	v1.5.0+ x	v3.3.0+ x	v0.48.0+ x	x
Color Extract				
RGB Extract the Red, Green, or Blue components from a color	x	x	x	v
HSL Extract the Hue, Saturation, or Lightness components from a color	x	x	x	v
HSV Extract the Hue, Saturation, or Value/Brightness components from a color	x	v	x requires custom function	v
alpha Extract the value of the alpha channel from a color	x	x	x	v
luma/luminosity	x	v	v0.47.0+ x	v
luminance	x	v	v	v
contrast	v	v	v0.47.0+ x	v
Color Test				
dark Returns true if the color is dark	x requires custom function	x requires custom function	x	v

light Returns true if the color is light	x requires custom function	x requires custom function	x	v
--	-------------------------------------	----------------------------------	---	---

Sass (SCSS)

Table of Content

1. Overview

2. Syntax

- [Sass](#)
- [SCSS](#)

3. Extending CSS

- [Variables](#)
- [Nesting](#)
- [Partials](#)
- [Mixins](#)
- [Extend/Inheritance](#)
- [Operators](#)

4. AiO Example

5. References

- [Further Reading](#)

Overview

Sass (Syntactically Awesome StyleSheets)

Sass is an extension of CSS that adds power and elegance to the basic language. It allows you to use variables, nested rules, mixins, inline imports, and more, all with a fully CSS-compatible syntax. Sass helps keep large stylesheets well-organized, and get small stylesheets up and running quickly.

Features

- Fully CSS-compatible.
- Language extensions such as variables, nesting, and mixins.
- Many useful functions for manipulating colors and other values.
- Advanced features like control directives for libraries.
- Well-formatted, customizable output.

Syntax

There are two syntaxes available for Sass.

Sass

The first and older syntax, known as the indented syntax (or sometimes just “Sass”), provides a more concise way of writing CSS. It uses indentation rather than brackets to indicate nesting of selectors, and newlines rather than semicolons to separate properties. Some people find this to be easier to read and quicker to write than SCSS.

Files using this syntax have the `.sass` extension.

```
// Variable
!primary-color= hotpink

// Mixin
=border-radius(!radius)
  -webkit-border-radius= !radius
  -moz-border-radius= !radius
  border-radius= !radius

.my-element
  color= !primary-color
  width= 100%
  overflow= hidden

.my-other-element
  +border-radius(5px)
```

SCSS

The second, known as SCSS (Sassy CSS) and used throughout this reference, is an extension of the syntax of CSS. This means that every valid CSS stylesheet is a valid SCSS file with the same meaning. In addition, SCSS understands most CSS hacks and vendor-specific syntax, such as IE’s old filter syntax.

Files using this syntax have the `.scss` extension.

```
// Variable
$primary-color: hotpink;

// Mixin
@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
  -moz-border-radius: $radius;
  border-radius: $radius;
}

.my-element {
  color: $primary-color;
  width: 100%;
  overflow: hidden;
}

.my-other-element {
  @include border-radius(5px);
}
```

Comparisons

SASS	SCSS
The Sass syntax is more concise	SCSS is more expressive
The Sass syntax is easier to read	SCSS encourages proper nesting of rules
The Sass syntax doesn't complain about missing semi-colons	SCSS encourages more modular code with @extend
-	SCSS allows me to write better inline documentation
-	Existing CSS tools often work with SCSS
-	Integration with an existing CSS codebase is much easier
-	SCSS provides a much lower barrier to entry

You will use SCSS as primary syntax through the course and encourage to use for development.

Extending CSS

Variables

[**Nesting**](#)

[**Partials**](#)

[**Mixins**](#)

[**Extend/Inheritance**](#)

[**Operators**](#)

References

Further Reading

Variables

Think of variables as a way to store information that you want to reuse throughout your stylesheet. You can store things like colors, font stacks, or any CSS value you think you'll want to reuse. Sass uses the `$` symbol to make something a variable.

```
$width: 5em;
```

You can then refer to them in properties:

```
#main {
  width: $width;
}
```

Global variable: `!global`

Variables are only available within the level of nested selectors where they're defined. If they're defined outside of any nested selectors, they're available everywhere. They can also be defined with the `!global` flag, in which case they're also available everywhere.

```
#main {
  $width: 5em !global;
  width: $width;
}

#sidebar {
  width: $width;
}
```

Compiled to:

```
#main {
  width: 5em;
}

#sidebar {
  width: 5em;
}
```

Variable defaults: `!default`

You can assign to variables if they aren't already assigned by adding the `!default` flag to the end of the value. This means that if the variable has already been assigned to, it won't be re-assigned, but if it doesn't have a value yet, it will be given one.

```
$content: "First content";
$content: "Second content?" !default;
$new_content: "First time reference" !default;

#main {
  content: $content;
  new-content: $new_content;
}
```

Compiled to:

```
#main {
  content: "First content";
  new-content: "First time reference"; }
```

Variables with `null` values are treated as unassigned by `!default`:

```
$content: null;
$content: "Non-null content" !default;

#main {
  content: $content;
}
```

Compiled to:

```
#main {
  content: "Non-null content"; }
```

Variable names (and all other Sass identifiers) can use hyphens and underscores interchangeably. For example, if you define a variable called `$main-width`, you can access it as `$main_width`, and vice versa.

Nesting

Sass will let you nest your CSS selectors in a way that follows the same visual hierarchy of your HTML.

Sass allows CSS rules to be nested within one another. The inner rule then only applies within the outer rule's selector. This is a great way to organize your CSS and make it more readable.

```
#main p {  
  color: #00ff00;  
  width: 97%;  
  
  .redbox {  
    background-color: #ff0000;  
    color: #000000;  
  }  
}
```

Compiled to:

```
#main p {  
  color: #00ff00;  
  width: 97%; }  
#main p .redbox {  
  background-color: #ff0000;  
  color: #000000; }
```

This helps avoid repetition of parent selectors, and makes complex CSS layouts with lots of nested selectors much simpler.

```
#main {  
  width: 97%;  
  
  p, div {  
    font-size: 2em;  
    a { font-weight: bold; }  
  }  
  
  pre { font-size: 3em; }  
}
```

Compiled to:

```
#main {
  width: 97%; }
#main p, #main div {
  font-size: 2em; }
#main p a, #main div a {
  font-weight: bold; }
#main pre {
  font-size: 3em; }
```

Referencing Parent Selectors: &

Sometimes it's useful to use a nested rule's parent selector in other ways than the default. For instance, you might want to have special styles for when that selector is hovered over or for when the body element has a certain class. In these cases, you can explicitly specify where the parent selector should be inserted using the `&` character.

```
a {
  font-weight: bold;
  text-decoration: none;
  &:hover { text-decoration: underline; }
  body.firefox & { font-weight: normal; }
}
```

Compiled to:

```
a {
  font-weight: bold;
  text-decoration: none; }
a:hover {
  text-decoration: underline; }
body.firefox a {
  font-weight: normal; }
```

- `&` will be replaced with the parent selector as it appears in the CSS. This means that if you have a deeply nested rule, the parent selector will be fully resolved before the `&` is replaced.


```
```scss
#main {
 color: black;
 a {
 font-weight: bold;
 &:hover { color: red; }
 }
}
```

```
Compiled to:
```css
#main {
  color: black; }
#main a {
  font-weight: bold; }
#main a:hover {
  color: red; }
```

- `&` must appear at the beginning of a compound selector, but it can be followed by a suffix that will be added to the parent selector.

```
#main {
  color: black;
  &-sidebar { border: 1px solid; }
}
```

Compiled to:

```
#main {
  color: black; }
#main-sidebar {
  border: 1px solid; }
```

If the parent selector can't have a suffix applied, Sass will throw an error.

Nested Properties

CSS has quite a few properties that are in `namespaces`; for instance, `font-family`, `font-size`, and `font-weight` are all in the font namespace. In CSS, if you want to set a bunch of properties in the same namespace, you have to type it out each time. Sass provides a shortcut for this: just write the namespace once, then nest each of the sub-properties within it.

```
.funky {
  font: {
    family: fantasy;
    size: 30em;
    weight: bold;
  }
}
```

Compiled to:

```
.funky {  
  font-family: fantasy;  
  font-size: 30em;  
  font-weight: bold; }
```

- The property namespace itself can also have a value.

```
.funky {  
  font: 20px/24px fantasy {  
    weight: bold;  
  }  
}
```

Compiled to:

```
.funky {  
  font: 20px/24px fantasy;  
  font-weight: bold;  
}
```

Be aware that overly nested rules will result in over-qualified CSS that could prove hard to maintain and is generally considered bad practice.

Partials

You can create partial Sass files that contain little snippets of CSS that you can include in other Sass files. This is a great way to modularize your CSS and help keep things easier to maintain.

- A partial is simply a Sass file named with a leading underscore. You might name it something like `_partial.scss`.
- The underscore lets Sass know that the file is only a partial file and that it should not be generated into a CSS file.
- Sass partials are used with the `@import` directive.

Import

CSS has an import option that lets you split your CSS into smaller, more maintainable portions.

- `@import` in CSS it creates another HTTP request. Sass builds on top of the current CSS `@import` but instead of requiring an HTTP request, Sass will take the file that you want to import and combine it with the file you're importing into so you can serve a single CSS file to the web browser.
- `@import` takes a filename to import. By default, it looks for a Sass file to import directly, but there are a few circumstances under which it will compile to a CSS `@import` rule:
 - If the file's extension is `.css`.
 - If the filename begins with `http://`.
 - If the filename is a `url()`.
 - If the `@import` has any media queries.
 - If none of the above conditions are met and the extension is `.scss` or `.sass`, then the named Sass or SCSS file will be imported.
 - If there is no extension, Sass will try to find a file with that name and the `.scss` or `.sass` extension and import it.

```
@import "foo.scss";  
  
// or  
  
@import "foo";
```

would both import the file `foo.scss`. Or:

```
@import "foo.css";
@import "foo" screen;
@import "http://foo.com/bar";
@import url(foo);
```

Compile to

```
@import "foo.css";
@import "foo" screen;
@import "http://foo.com/bar";
@import url(foo);
```

- It's also possible to import multiple files in one `@import`.

```
@import "rounded-corners", "text-shadow";
```

would import both the `rounded-corners` and the `text-shadow` files.

Partials and Import

You might have `_colors.scss`, then no `_colors.css` file would be created. You can then import that file without using the underscore.

```
@import "colors";
```

and `_colors.scss` would be imported.

Example:

- You have `_reset.scss` and want to import it in `base.scss`:

```
// _reset.scss

html,
body,
ul,
ol {
  margin: 0;
  padding: 0;
}
```

```
// base.scss

@import 'reset';

body {
  font: 100% Helvetica, sans-serif;
  background-color: #eefefef;
}
```

- When you generate the CSS you'll get:

```
html, body, ul, ol {
  margin: 0;
  padding: 0;
}

body {
  font: 100% Helvetica, sans-serif;
  background-color: #eefefef;
}
```

You may not include a *partial* and a *non-partial* with the same name in the same directory. For example, `_colors.scss` may not exist alongside `colors.scss`.

Mixins

Some things in CSS are a bit tedious to write, especially with CSS3 and the many vendor prefixes that exist. A mixin lets you make groups of CSS declarations that you want to reuse throughout your site. You can even pass in values to make your mixin more flexible.

- Mixins allow you to define styles that can be re-used throughout the stylesheet without needing to resort to non-semantic classes like `.float-left`.
- Mixins can also contain full CSS rules, and anything else allowed elsewhere in a Sass document. They can even take arguments which allows you to produce a wide variety of styles with very few mixins.

Defining a Mixin: `@ mixin`

- Mixins are defined with the `@ mixin` directive.

It's followed by the name of the mixin and optionally the arguments, and a block containing the contents of the mixin.

```
@mixin large-text {
  font: {
    family: Arial;
    size: 20px;
    weight: bold;
  }
  color: #ff0000;
}
```

- Mixins may also contain selectors, possibly mixed with properties.

The selectors can even contain parent references.

```
@mixin clearfix {
  display: inline-block;
  &:after {
    content: ".";
    display: block;
    height: 0;
    clear: both;
    visibility: hidden;
  }
  * html & { height: 1px }
}
```

Mixin names (and all other Sass identifiers) can use hyphens and underscores interchangeably. For example, if you define a mixin called `add-column`, you can include it as `add_column`, and vice versa.

Including a Mixin: `@include`

- Mixins are included in the document with the `@include` directive.
 - This takes the name of a mixin and optionally arguments to pass to it, and includes the styles defined by that mixin into the current rule.

```
.page-title {  
  @include large-text;  
  padding: 4px;  
  margin-top: 10px;  
}
```

Compiled to:

```
.page-title {  
  font-family: Arial;  
  font-size: 20px;  
  font-weight: bold;  
  color: #ff0000;  
  padding: 4px;  
  margin-top: 10px; }
```

- Mixins may also be included outside of any rule (that is, at the root of the document) as long as they don't directly define any properties or use any parent references.

```
@mixin silly-links {  
  a {  
    color: blue;  
    background-color: red;  
  }  
}  
  
@include silly-links;
```

Compiled to:

```
a {  
  color: blue;  
  background-color: red; }
```

- Mixin definitions can also include other mixins.

```

@mixin compound {
  @include highlighted-background;
  @include header-text;
}

@mixin highlighted-background { background-color: #fc0; }
@mixin header-text { font-size: 20px; }

```

From Sass v3.3, mixins may include themselves. Mixins that only define descendent selectors can be safely mixed into the top most level of a document.

Arguments

- The arguments are written as variable names separated by commas, all in parentheses after the name.

When including the mixin, values can be passed in in the same manner.

```

@mixin sexy-border($color, $width) {
  border: {
    color: $color;
    width: $width;
    style: dashed;
  }
}

p { @include sexy-border(blue, 1in); }

```

Compiled to:

```

p {
  border-color: blue;
  border-width: 1in;
  border-style: dashed; }

```

- Mixins can specify default values for their arguments using the normal variable-setting syntax.

When the mixin is included, if it doesn't pass in that argument, the default value will be used instead.

```
@mixin sexy-border($color, $width: 1in) {
  border: {
    color: $color;
    width: $width;
    style: dashed;
  }
}
p { @include sexy-border(blue); }
h1 { @include sexy-border(blue, 2in); }
```

Compiled to:

```
p {
  border-color: blue;
  border-width: 1in;
  border-style: dashed; }

h1 {
  border-color: blue;
  border-width: 2in;
  border-style: dashed; }
```

Keyword Arguments

- Mixins can be included using explicit keyword arguments.

```
p { @include sexy-border($color: blue); }
h1 { @include sexy-border($color: blue, $width: 2in); }
```

- It can make the stylesheet easier to read. It also allows functions to present more flexible interfaces, providing many arguments without becoming difficult to call.
- Named arguments can be passed in any order, and arguments with default values can be omitted. Since the named arguments are variable names, underscores and dashes can be used interchangeably.

Variable Arguments

- Sometimes it makes sense for a mixin or function to take an unknown number of arguments.

If a mixin for creating box shadows might take any number of shadows as arguments, then Sass supports “variable arguments,” which are arguments at the end of a mixin or function declaration that take all leftover arguments and package them up as a list.

These arguments look just like normal arguments, but are followed by ...

```
@mixin box-shadow($shadows...) {
  -moz-box-shadow: $shadows;
  -webkit-box-shadow: $shadows;
  box-shadow: $shadows;
}

.shadows {
  @include box-shadow(0px 4px 5px #666, 2px 6px 10px #999);
}
```

Compiled to:

```
.shadows {
  -moz-box-shadow: 0px 4px 5px #666, 2px 6px 10px #999;
  -webkit-box-shadow: 0px 4px 5px #666, 2px 6px 10px #999;
  box-shadow: 0px 4px 5px #666, 2px 6px 10px #999;
}
```

- Variable arguments contain any keyword arguments passed to the mixin or function.

These can be accessed using the keywords (`$args`) function, which returns them as a map from strings (without `$`) to values.

- Variable arguments can be used when calling a mixin.

Using the same syntax, you can expand a list of values so that each value is passed as a separate argument, or expand a map of values so that each pair is treated as a keyword argument.

```
@mixin colors($text, $background, $border) {
  color: $text;
  background-color: $background;
  border-color: $border;
}

$values: #ff0000, #00ff00, #0000ff;
.primary {
  @include colors($values...);
}

$value-map: (text: #00ff00, background: #0000ff, border: #ff0000);
.secondary {
  @include colors($value-map...);
}
```

Compiled to:

```
.primary {  
  color: #ff0000;  
  background-color: #00ffff;  
  border-color: #0000ff;  
}  
  
.secondary {  
  color: #00ffff;  
  background-color: #0000ff;  
  border-color: #ff0000;  
}
```

You can pass both an argument list and a map as long as the list comes before the map, as in `@include colors($values..., $map...)`.

- You can use variable arguments to wrap a mixin and add additional styles without changing the argument signature of the mixin.

Keyword arguments will get directly passed through to the wrapped mixin.

```
@mixin wrapped-stylish-mixin($args...) {  
  font-weight: bold;  
  @include stylish-mixin($args...);  
}  
  
.stylish {  
  // The $width argument will get passed on to "stylish-mixin" as a keyword  
  @include wrapped-stylish-mixin(#00ffff, $width: 100px);  
}
```

Passing Content Blocks to a Mixin

- It is possible to pass a block of styles to the mixin for placement within the styles included by the mixin.

The styles will appear at the location of any `@content` directives found within the mixin. This makes it possible to define abstractions relating to the construction of selectors and directives.

```
@mixin apply-to-ie6-only {
  * html {
    @content;
  }
}
@include apply-to-ie6-only {
  #logo {
    background-image: url(/logo.gif);
  }
}
```

Generates:

```
* html #logo {
  background-image: url(/logo.gif);
}
```

Note: when the `@content` directive is specified more than once or in a loop, the style block will be duplicated with each invocation.

Variable Scope and Content Blocks

- The block of content passed to a mixin are evaluated in the scope where the block is defined, not in the scope of the mixin.

This means that variables local to the mixin cannot be used within the passed style block and variables will resolve to the global value.

```
$color: white;
@mixin colors($color: blue) {
  background-color: $color;
  @content;
  border-color: $color;
}
.colors {
  @include colors { color: $color; }
}
```

Compiles to:

```
.colors {
  background-color: blue;
  color: white;
  border-color: blue;
}
```

- Additionally, this makes it clear that the variables and mixins that are used within the passed block are related to the other styles around where the block is defined.

```
#sidebar {  
    $sidebar-width: 300px;  
    width: $sidebar-width;  
    @include smartphone {  
        width: $sidebar-width / 3;  
    }  
}
```

Extend/Inheritance

This is one of the most useful features of Sass. Using `@extend` lets you share a set of CSS properties from one selector to another. It helps keep your Sass very DRY.

`@extend` works by inserting the extending selector anywhere in the stylesheet that the extended selector appears.

```
.error {
  border: 1px #f00;
  background-color: #fdd;
}

.error.intrusion {
  background-image: url("/image/hacked.png");
}

.seriousError {
  @extend .error;
  border-width: 3px;
}
```

Compiled to:

```
.error, .seriousError {
  border: 1px #f00;
  background-color: #fdd; }

.error.intrusion, .seriousError.intrusion {
  background-image: url("/image/hacked.png"); }

.seriousError {
  border-width: 3px; }
```

When merging selectors, `@extend` is smart enough to avoid unnecessary duplication, so something like `.seriousError.seriousError` gets translated to `.seriousError`. In addition, it won't produce selectors that can't match anything, like `#main#footer`.

Extending Complex Selectors

Class selectors aren't the only things that can be extended.

It's possible to extend any selector involving only a single element, such as

```
.special.cool, a:hover, or a.user[href^="http://"] .
```

```
.hoverlink {  
    @extend a:hover;  
}
```

- All styles defined for `a:hover` are also applied to `.hoverlink`.

```
.hoverlink {  
    @extend a:hover;  
}  
a:hover {  
    text-decoration: underline;  
}
```

Compiled to:

```
a:hover, .hoverlink {  
    text-decoration: underline; }
```

- Any rule that uses `a:hover` will also work for `.hoverlink`, even if they have other selectors as well.

```
.hoverlink {  
    @extend a:hover;  
}  
.comment a.user:hover {  
    font-weight: bold;  
}
```

Compiled to:

```
.comment a.user:hover, .comment .user.hoverlink {  
    font-weight: bold; }
```

Multiple Extends

A single selector can extend more than one selector.

| This means that it inherits the styles of all the extended selectors.

```
.error {  
  border: 1px #f00;  
  background-color: #fdd;  
}  
.attention {  
  font-size: 3em;  
  background-color: #ff0;  
}  
.seriousError {  
  @extend .error;  
  @extend .attention;  
  border-width: 3px;  
}
```

Compiled to:

```
.error, .seriousError {  
  border: 1px #f00;  
  background-color: #fdd; }  
  
.attention, .seriousError {  
  font-size: 3em;  
  background-color: #ff0; }  
  
.seriousError {  
  border-width: 3px; }
```

In effect, every element with class `.seriousError` also has class `.error` and class `.attention`.

Multiple extends can also be written using a comma-separated list of selectors. For example, `@extend .error, .attention` is the same as `@extend .error; @extend .attention`.

Chaining Extends

It's possible for one selector to extend another selector that in turn extends a third.

```
.error {  
  border: 1px #f00;  
  background-color: #fdd;  
}  
.seriousError {  
  @extend .error;  
  border-width: 3px;  
}  
.criticalError {  
  @extend .seriousError;  
  position: fixed;  
  top: 10%;  
  bottom: 10%;  
  left: 10%;  
  right: 10%;  
}
```

Now everything with class `.seriousError` also has class `.error`, and everything with class `.criticalError` has class `.seriousError` and class `.error`. It's compiled to:

```
.error, .seriousError, .criticalError {  
  border: 1px #f00;  
  background-color: #fdd; }  
  
.seriousError, .criticalError {  
  border-width: 3px; }  
  
.criticalError {  
  position: fixed;  
  top: 10%;  
  bottom: 10%;  
  left: 10%;  
  right: 10%; }
```

Selector Sequences

Selector sequences, such as `.foo .bar` or `.foo + .bar`, currently can't be extended. However, it is possible for nested selectors themselves to use `@extend`.

```
#fake-links .link {
  @extend a;
}

a {
  color: blue;
  &:hover {
    text-decoration: underline;
  }
}
```

Compiled to

```
a, #fake-links .link {
  color: blue; }
a:hover, #fake-links .link:hover {
  text-decoration: underline; }
```

Merging Selector Sequences

Sometimes a selector sequence extends another selector that appears in another sequence. In this case, the two sequences need to be merged.

```
#admin .tabbar a {
  font-weight: bold;
}
#demo .overview .fakelink {
  @extend a;
}
```

While it would technically be possible to generate all selectors that could possibly match either sequence, this would make the stylesheet far too large. The simple example above, for instance, would require ten selectors. Instead, Sass generates only selectors that are likely to be useful.

- When the two sequences being merged have no selectors in common, then two new selectors are generated: one with the first sequence before the second, and one with the second sequence before the first.

```
#admin .tabbar a {
  font-weight: bold;
}
#demo .overview .fakelink {
  @extend a;
}
```

Compiled to:

```
#admin .tabbar a,
#admin .tabbar #demo .overview .fakelink,
#demo .overview #admin .tabbar .fakelink {
  font-weight: bold; }
```

- If the two sequences do share some selectors, then those selectors will be merged together and only the differences (if any still exist) will alternate.

```
#admin .tabbar a {
  font-weight: bold;
}
#admin .overview .fakelink {
  @extend a;
}
```

Compiled to:

```
#admin .tabbar a,
#admin .tabbar .overview .fakelink,
#admin .overview .tabbar .fakelink {
  font-weight: bold; }
```

@extend -Only Selectors

Sometimes you'll write styles for a class that you only ever want to `@extend`, and never want to use directly in your HTML. This is especially true when writing a Sass library, where you may provide styles for users to `@extend` if they need and ignore if they don't.

If you use normal classes for this, you end up creating a lot of extra CSS when the stylesheets are generated, and run the risk of colliding with other classes that are being used in the HTML. That's why Sass supports "placeholder selectors" (for example, `%foo`).

- Placeholder selectors look like `class` and `id` selectors, except the `#` or `.` is replaced by `%`.

```
// This ruleset won't be rendered on its own.
#context a%extreme {
  color: blue;
  font-weight: bold;
  font-size: 2em;
}
```

- Placeholder selectors can be extended.

The extended selectors will be generated, but the base placeholder selector will not.

```
.notice {  
  @extend %extreme;  
}
```

Compiled to:

```
#context a.notice {  
  color: blue;  
  font-weight: bold;  
  font-size: 2em; }
```

The `!optional` Flag

Normally when you extend a selector, it's an error if that `@extend` doesn't work.

Sometimes, though, you want to allow an `@extend` not to produce any new selectors.

To do so, just add the `!optional` flag after the selector.

```
a.important {  
  @extend .notice !optional;  
}
```

`@extend` in Directives

There are some restrictions on the use of `@extend` within directives such as `@media`.
Sass is unable to make CSS rules outside of the `@media` block apply to selectors
inside it without creating a huge amount of stylesheet bloat by copying styles all over
the place. This means that if you use `@extend` within `@media` (or other CSS
directives), you may only extend selectors that appear within the same directive block.

- The following works fine:

```
@media print {  
  .error {  
    border: 1px #f00;  
    background-color: #fdd;  
  }  
  .seriousError {  
    @extend .error;  
    border-width: 3px;  
  }  
}
```

- But this is an error:

```
.error {  
  border: 1px #f00;  
  background-color: #fdd;  
}  
  
@media print {  
  .seriousError {  
    // INVALID EXTEND: .error is used outside of the "@media print" directive  
    @extend .error;  
    border-width: 3px;  
  }  
}
```

Operators

Doing math in your CSS is very helpful. Sass has a handful of standard math operators like `+`, `-`, `*`, `/`, and `%`.

Data types

Sass supports seven main data types:

- numbers (e.g. `1.2`, `13`, `10px`)
- strings of text, with and without quotes (e.g. `"foo"`, `'bar'`, `baz`)
- colors (e.g. `blue`, `#04a3f9`, `rgba(255, 0, 0, 0.5)`)
- booleans (e.g. `true`, `false`)
- nulls (e.g. `null`)
- lists of values, separated by spaces or commas (e.g. `1.5em 1em 0 2em`, `Helvetica, Arial, sans-serif`)
- maps from one value to another (e.g. `(key1: value1, key2: value2)`)

Sass also supports *all other types of CSS property value*, such as Unicode ranges and `!important` declarations. However, it has no special handling for these types. They're treated just like unquoted strings.

Operations

All types support equality operations (`==` and `!=`). In addition, each type has its own operations that it has special support for.

Number Operations

Sass supports the standard arithmetic operations on numbers (addition `+`, subtraction `-`, multiplication `*`, division `/`, and modulo `%`).

Sass math functions preserve units during arithmetic operations. This means that, just like in real life, you cannot work on numbers with incompatible units (such as adding a number with px and em) and two numbers with the same unit that are multiplied together will produce square units (`10px * 10px == 100px * px`). Be Aware that `px * px` is an invalid CSS unit and you will get an error from Sass for attempting to use invalid units in CSS.

Relational operators (`<`, `>`, `<=`, `>=`) are also supported for numbers, and equality operators (`==`, `!=`) are supported for all types.

Division and `/`

CSS allows `/` to appear in property values as a way of separating numbers.

Since Sass is an extension of the CSS property syntax, it must support this, while also allowing `/` to be used for division. This means that by default, if two numbers are separated by `/` in Sass, then they will appear that way in the resulting CSS.

However, there are three situations where the `/` will be interpreted as division. These cover the vast majority of cases where division is actually used. They are:

- If the value, or any part of it, is stored in a variable or returned by a function.
- If the value is surrounded by parentheses, unless those parentheses are outside a list and the value is inside.
- If the value is used as part of another arithmetic expression.

```
p {
  font: 10px/8px;           // Plain CSS, no division
  $width: 1000px;
  width: $width/2;          // Uses a variable, does division
  width: round(1.5)/2;      // Uses a function, does division
  height: (500px/2);        // Uses parentheses, does division
  margin-left: 5px + 8px/2px; // Uses +, does division
  font: (italic bold 10px/8px); // In a list, parentheses don't count
}
```

Compiled to:

```
p {
  font: 10px/8px;
  width: 500px;
  height: 250px;
  margin-left: 9px; }
```

If you want to use variables along with a plain CSS `/`, you can use `#{}#` to insert them.

```
p {
  $font-size: 12px;
  $line-height: 30px;
  font: #{$font-size}/#{$line-height};
}
```

Compiled to:

```
p {
  font: 12px/30px; }
```

Subtraction, Negative Numbers, and -

There are a number of different things - can mean in CSS and in Sass.

It can be:

- A subtraction operator (as in 5px - 3px).
- The beginning of a negative number (as in -3px).
- A unary negation operator (as in -\$var).
- Part of an identifier (as in font-weight).

Most of the time, it's clear which is which, but there are some tricky cases. As a general rule, you're safest if:

- You always include spaces on both sides of - when subtracting.
- You include a space before - but not after for a negative number or a unary negation.
- You wrap a unary negation in parentheses if it's in a space-separated list, as in 10px (- \$var) .

The different meanings of - take precedence in the following order:

- A - as part of an identifier.

This means that a-1 is an unquoted string with value "a-1". The only exception are units; Sass normally allows any valid identifier to be used as an identifier, but identifiers may not contain a hyphen followed by a digit. This means that 5px-3px is the same as 5px - 3px .

- A - between two numbers with no whitespace.

This indicates subtraction, so 1-2 is the same as 1 - 2 .

- A - at the beginning of a literal number.

This indicates a negative number, so 1 -2 is a list containing 1 and -2 .

- A - between two numbers regardless of whitespace.

This indicates subtraction, so 1 -\$var are the same as 1 - \$var .

- A - before a value.

This indicates the unary negation operator; that is, the operator that takes a number and returns its negative.

Color Operations

- All arithmetic operations are supported for color values, where they work piecewise.

This means that the operation is performed on the red, green, and blue components in turn.

```
p {
  color: #010203 + #040506;
}
```

Computes $01 + 04 = 05$, $02 + 05 = 07$, and $03 + 06 = 09$, and is compiled to:

```
p {
  color: #050709; }
```

Often it's more useful to use color functions than to try to use color arithmetic to achieve the same effect.

- Arithmetic operations also work between numbers and colors, also piecewise.

```
p {
  color: #010203 * 2;
```

Computes $01 * 2 = 02$, $02 * 2 = 04$, and $03 * 2 = 06$, and is compiled to:

```
p {
  color: #020406; }
```

Note that colors with an alpha channel (those created with the `rgba` or `hsla` functions) must have the same alpha value in order for color arithmetic to be done with them.

- The arithmetic doesn't affect the alpha value.

```
p {
  color: rgba(255, 0, 0, 0.75) + rgba(0, 255, 0, 0.75); }
```

Compiled to:

```
p {
  color: rgba(255, 255, 0, 0.75); }
```

- The alpha channel of a color can be adjusted using the `opacify` and `transparentize` functions.

```
$translucent-red: rgba(255, 0, 0, 0.5);
p {
  color: opacify($translucent-red, 0.3);
  background-color: transparentize($translucent-red, 0.25);
}
```

Compiled to:

```
p {
  color: rgba(255, 0, 0, 0.8);
  background-color: rgba(255, 0, 0, 0.25); }
```

IE filters require all colors include the alpha layer, and be in the strict format of `#AABBCCDD`. You can more easily convert the color using the `ie_hex_str` function.

```
$translucent-red: rgba(255, 0, 0, 0.5);
$green: #00ff00;
div {
  filter: progid:DXImageTransform.Microsoft.gradient(enabled='false', startColorstr='#{ie-hex-str($green)}', endColorstr='#{ie-hex-str($translucent-red)}');
}
```

Compiled to:

```
div {
  filter: progid:DXImageTransform.Microsoft.gradient(enabled='false', startColorstr='#FF00FF00', endColorstr='#80FF0000');
```

String Operations

The `+` operation can be used to concatenate strings:

```
p {
  cursor: e + -resize;
}
```

Compiled to:

```
p {  
  cursor: e-resize; }
```

Note that if a quoted string is added to an unquoted string (that is, the quoted string is to the left of the `+`), the result is a quoted string.

- If an unquoted string is added to a quoted string (the unquoted string is to the left of the `+`), the result is an unquoted string.

```
p:before {  
  content: "Foo " + Bar;  
  font-family: sans- + "serif";  
}
```

Compiled to:

```
p:before {  
  content: "Foo Bar";  
  font-family: sans-serif; }
```

- By default, if two values are placed next to one another, they are concatenated with a space:

```
p {  
  margin: 3px + 4px auto;  
}
```

Compiled to:

```
p {  
  margin: 7px auto; }
```

- Within a string of text, `#{} style interpolation` can be used to place dynamic values within the string:

```
p:before {  
  content: "I ate #{5 + 10} pies!";  
}
```

Compiled to:

```
p:before {  
  content: "I ate 15 pies!"; }
```

- Null values are treated as empty strings for string interpolation:

```
$value: null;  
p:before {  
  content: "I ate #{value} pies!";  
}
```

Compiled to:

```
p:before {  
  content: "I ate  pies!"; }
```

Boolean Operations

Sass supports `and`, `or`, and `not` operators for boolean values.

Parentheses

Parentheses can be used to affect the order of operations:

```
p {  
  width: 1em + (2em * 3);  
}
```

Compiled to:

```
p {  
  width: 7em; }
```

Example

| Every pieces put together for a working SASS/SCSS.

Conventions

Table of Content

1. [Terminology](#)
 - i. [Rule Declaration](#)
 - ii. [Selectors](#)
 - iii. [Properties](#)
2. [Formatting](#)
3. [Comments](#)
4. [OOCSS and BEM](#)
5. [ID Selectors](#)
6. [JavaScript hooks](#)
7. [Border](#)

Terminology

Rule declaration

A “rule declaration” is the name given to a selector (or a group of selectors) with an accompanying group of properties. Here's an example:

```
.listing {  
  font-size: 18px;  
  line-height: 1.2;  
}
```

Selectors

In a rule declaration, “selectors” are the bits that determine which elements in the DOM tree will be styled by the defined properties. Selectors can match HTML elements, as well as an element's class, ID, or any of its attributes. Here are some examples of selectors:

```
.my-element-class {  
    /* ... */  
}  
  
[aria-hidden] {  
    /* ... */  
}
```

Properties

Finally, properties are what give the selected elements of a rule declaration their style. Properties are key-value pairs, and a rule declaration can contain one or more property declarations. Property declarations look like this:

```
/* some selector */ {  
    background: #f1f1f1;  
    color: #333;  
}
```

CSS

Formatting

- Prefer dashes over camelCasing in class names.
 - Underscores and PascalCasing are okay if you are using BEM (see [OOCSS](#) and [BEM](#) below).
- Do not use ID selectors
- When using multiple selectors in a rule declaration, give each selector its own line.
- Put a space before the opening brace `{` in rule declarations
- In properties, put a space after, but not before, the `:` character.
- Put closing braces `}` of rule declarations on a new line
- Put blank lines between rule declarations

Bad

```
.avatar{  
    border-radius:50%;  
    border:2px solid white; }  
.no, .nope, .not_good {  
    // ...  
}  
#lol-no {  
    // ...  
}
```

Good

```
.avatar {  
border-radius: 50%;  
border: 2px solid white;  
}  
  
.one,  
.selector,  
.per-line {  
// ...  
}
```

Comments

- Prefer line comments (`//` in Sass-land) to block comments.
- Prefer comments on their own line. Avoid end-of-line comments.
- Write detailed comments for code that isn't self-documenting:
 - Uses of z-index
 - Compatibility or browser-specific hacks

OOCSS and BEM

We encourage some combination of OOCSS and BEM for these reasons:

- It helps create clear, strict relationships between CSS and HTML
- It helps us create reusable, composable components
- It allows for less nesting and lower specificity
- It helps in building scalable stylesheets

OOCSS, or “Object Oriented CSS”, is an approach for writing CSS that encourages you to think about your stylesheets as a collection of “objects”: reusable, repeatable snippets that can be used independently throughout a website.

- Nicole Sullivan's [OOCSS wiki](#)

- Smashing Magazine's [Introduction to OOCSS](#)

BEM, or “Block-Element-Modifier”, is a *naming convention* for classes in HTML and CSS. It was originally developed by Yandex with large codebases and scalability in mind, and can serve as a solid set of guidelines for implementing OOCSS.

- CSS Trick's [BEM 101](#)

- Harry Roberts' [introduction to BEM](#)

We recommend a variant of BEM with PascalCased “blocks”, which works particularly well when combined with components (e.g. React). Underscores and dashes are still used for modifiers and children.

Example

```
// ListingCard.jsx
function ListingCard() {
  return (
    <article class="ListingCard ListingCard--featured">

      <h1 class="ListingCard__title">Adorable 2BR in the sunny Mission</h1>

      <div class="ListingCard__content">
        <p>Vestibulum id ligula porta felis euismod semper.</p>
      </div>

    </article>
  );
}
```

```
/* ListingCard.css */
.ListingCard { }
.ListingCard--featured { }
.ListingCard__title { }
.ListingCard__content { }
```

- `.ListingCard` is the “block” and represents the higher-level component
- `.ListingCard__title` is an “element” and represents a descendant of `.ListingCard` that helps compose the block as a whole.
- `.ListingCard--featured` is a “modifier” and represents a different state or variation on the `.ListingCard` block.

ID selectors

While it is possible to select elements by ID in CSS, it should generally be considered an anti-pattern. ID selectors introduce an unnecessarily high level of [specificity](#) to your rule declarations, and they are not reusable.

For more on this subject, read [CSS Wizardry's article](#) on dealing with specificity.

JavaScript hooks

Avoid binding to the same class in both your CSS and JavaScript. Conflating the two often leads to, at a minimum, time wasted during refactoring when a developer must cross-reference each class they are changing, and at its worst, developers being afraid to make changes for fear of breaking functionality.

We recommend creating JavaScript-specific classes to bind to, prefixed with `.js-` :

```
<button class="btn btn-primary js-request-to-book">Request to Book</button>
```

Border

Use `0` instead of `none` to specify that a style has no border.

Bad

```
.foo {  
  border: none;  
}
```

Good

```
.foo {  
  border: 0;  
}
```

Resources & Further Reading

Our coding conventions are heavily inspired by Airbnb's CSS / SASS Style Guide:

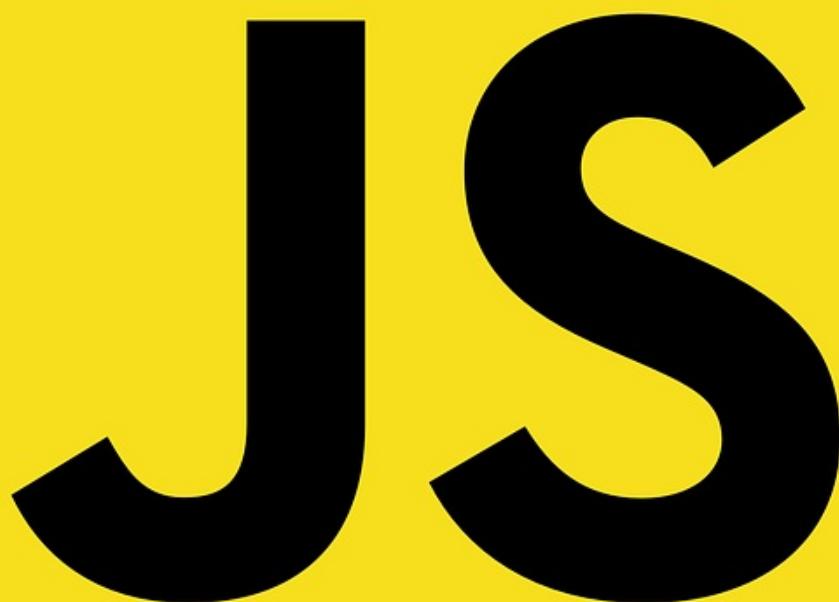
<https://github.com/airbnb/css>

Section 3: JavaScript

Table of Content

1. [Introduction](#)
 - [JavaScript and Java](#)
 - [Today JavaScript](#)
 - [ECMAScript](#)
2. [Grammar and types](#)
3. [Statements and Declaration](#)
4. [Expressions and Operators](#)
5. [Functions](#)
6. [Prototyping](#)
7. [Built-in Objects](#)
8. [Browsers](#)
9. [Debugging](#)
10. [Extended Libraries](#)
 - i. [jQuery](#)
 - ii. [ReactJS](#)
11. [Conventions](#)

Introduction



JavaScript is a high level, dynamic, untyped(!?), and interpreted programming language. It has been standardized in the ECMAScript language specification. Alongside HTML and CSS, it is one of the three essential technologies of World Wide Web content production; the majority of websites employ it and it is supported by all modern web browsers without plug-ins. JavaScript is prototype-based with first-class functions, making it a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. It has an API for working with text, arrays, dates and regular expressions, but does not include any I/O, such as networking, storage or graphics facilities, relying for these upon the host environment in which it is embedded.

JavaScript and Java

JavaScript and Java are similar in some ways but fundamentally different in some others. The JavaScript language resembles Java but does not have Java's static typing and strong type checking. JavaScript follows most Java expression syntax, naming conventions and basic control-flow constructs which was the reason why it was renamed from LiveScript to JavaScript.

JavaScript compared to Java

JavaScript	Java
Object-oriented. No distinction between types of objects. Inheritance is through the prototype mechanism, and properties and methods can be added to any object dynamically.	Class-based. Objects are divided into classes and instances with all inheritance through the class hierarchy. Classes and instances cannot have properties or methods added dynamically.
Variable data types are not declared (dynamic typing).	Variable data types must be declared (static typing).
Cannot automatically write to hard disk.	Can automatically write to hard disk.

Today JavaScript

JavaScript is standardized at Ecma International — the European association for standardizing information and communication systems (ECMA was formerly an acronym for the European Computer Manufacturers Association) to deliver a standardized, international programming language based on JavaScript. This standardized version of JavaScript, called ECMAScript, behaves the same way in all applications that support the standard. Companies can use the open standard language to develop their implementation of JavaScript. The ECMAScript standard is documented in the ECMA-262 specification.

JavaScript is a cross-platform, object-oriented scripting language. It is a small and lightweight language. Inside a host environment (for example, a web browser), JavaScript can be connected to the objects of its environment to provide programmatic control over them. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects; for example:

- **Client-side JavaScript** extends the core language by supplying objects to control a browser and its Document Object Model (DOM). For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation.
- **Server-side JavaScript** extends the core language by supplying objects relevant to running JavaScript on a server. For example, server-side extensions allow an application to communicate with a database, provide continuity of information from one

invocation to another of the application, or perform file manipulations on a server.

ECMAScript

ECMAScript (or ES) is a trademarked scripting-language specification standardized by Ecma International in ECMA-262 and ISOVIEC 16262. It was based on JavaScript, which now tracks ECMAScript. It is commonly used for client-side scripting on the World Wide Web.

Versions

Edition	Date published	Changes from prior edition
1	June 1997	First edition
2	June 1998	Editorial changes to keep the specification fully aligned with ISOVIEC 16262 international standard
3	December 1999	Added regular expressions, better string handling, new control statements, try/catch exception handling, tighter definition of errors, formatting for numeric output and other enhancements.
4	Abandoned	Fourth Edition was abandoned, due to political differences concerning language complexity. Many features proposed for the Fourth Edition have been completely dropped; some are proposed for ECMAScript Harmony.
5	December 2009	Adds "strict mode," a subset intended to provide more thorough error checking and avoid error-prone constructs. Clarifies many ambiguities in the 3rd edition specification, and accommodates behaviour of real-world implementations that differed consistently from that specification. Adds some new features, such as getters and setters, library support for JSON, and more complete reflection on object properties.
5.1	June 2011	This edition 5.1 of the ECMAScript Standard is fully aligned with third edition of the international standard ISOVIEC 16262:2011.
6	June 2015	The Sixth Edition, known as ES6 or ECMAScript 2015, adds significant new syntax for writing complex applications, including classes and modules, but defines them semantically in the same terms as ECMAScript 5 strict mode. Other new features include iterators and for/of loops, Python-style generators and generator expressions, arrow functions, binary data, typed arrays, collections (maps, sets and weak maps), promises, number and math enhancements, reflection, and proxies (metaprogramming for virtual objects and wrappers). As the first "ECMAScript Harmony" specification, it is also known as "ES6 Harmony."
7	June 2016	The Seventh Edition intended to continue the themes of language reform, code isolation, control of effects and library/tool enabling from ES6, includes two new features: the exponentiation operator (**) and Array.prototype.includes.

Hello world

```
function greetMe(yourName) {  
    alert("Hello " + yourName);  
}  
  
greetMe("World");
```

Grammar and Types

Table of Content

- [Basic](#)
- [Comment](#)
- [Data Types](#)
- [Data Structures](#)

Basic

JavaScript borrows most of its syntax from Java, but is also influenced by Awk, Perl and Python.

In JavaScript,

- JavaScript is case-sensitive and uses the Unicode character set.
- Instructions are called statements and are separated by a semicolon (;). It is recommended to always add semicolons to end your statements; it will avoid side effects.
- Spaces, tabs and newline characters are called whitespace.
- The source text of JavaScript scripts gets scanned from left to right and is converted into a sequence of input elements which are tokens, control characters, line terminators, comments or whitespace.

Comment

The syntax of comments is the same as in C++ and in many other languages.

JavaScript has two ways of assigning comments in its code:

- The first way is the `//` comment; this makes all text following it on the same line into a comment.
- The second way is the `/* */` style, which is much more flexible.

Syntax:

```
// a one line comment

/* this is a longer,
   multi-line comment
*/

/* You can't, however, /* nest comments */ SyntaxError */
```

Data Types

JavaScript is a dynamically typed language. That means you don't have to specify the data type of a variable when you declare it, and data types are converted automatically as needed during script execution.

However, the latest ECMAScript standard also defines seven data types:

- Six data types that are *primitives*:
 - **Boolean**. true and false.
 - `null`. A special keyword denoting a null value. Because JavaScript is case-sensitive, null is not the same as Null, NULL, or any other variant.
 - `undefined`. A top-level property whose value is undefined.
 - **Number**. 42 or 3.14159.
 - **String**. "Howdy"
 - **Symbol** (*new in ECMAScript 2015*). A data type whose instances are unique and immutable.
- and **Object**.

A primitive (primitive value, primitive data type) is data that is not an object and has no methods. All primitives are *immutable* (cannot be changed).

Primitive wrapper objects: Except for `null` and `undefined`, all primitive values have object equivalents that wrap around the primitive values:

- `String` for the string primitive.
- `Number` for the number primitive.
- `Boolean` for the Boolean primitive.
- `Symbol` for the Symbol primitive.

Although these data types are a relatively small amount, they enable you to perform useful functions with your applications.

To be technically accurate, the type diagram looks more like this:

- Number

- String
- Boolean
- Symbol (new in Edition 6)
- Object
 - Function
 - Array
 - Date
 - RegExp
- null
- undefined

Objects and **functions** are the other fundamental elements in JavaScript. You can think of objects as named containers for values, and functions as procedures that your application can perform. We will go deep on them in next chapters.

Literals

You use literals to represent values in JavaScript. These are fixed values, not variables, that you literally provide in your script.

- `null` .
- Boolean literal:

```
true  
false
```

Do not confuse the primitive Boolean values `true` and `false` with the *true* and *false* values of the *Boolean object*. The Boolean object is a wrapper around the primitive Boolean data type.

- Array literal:

An array literal is a list of zero or more expressions, each of which represents an array element, enclosed in square brackets ([]). When you create an array using an array literal, it is initialized with the specified values as its elements, and its length is set to the number of arguments specified.

```
var coffees = ["French Roast", "Colombian", "Kona"];
```

Extra commas in array literals : You do not have to specify all elements in an array literal. If you put two commas in a row, the array is created with `undefined` for the unspecified elements. The following example creates the fish array:

```
var fish = ["Lion", , "Angel"]; // fish[0] is "Lion", fish[1] is undefined, and fish[2] is "Angel"
```

- **Integers**

Integers can be expressed in decimal (base 10), hexadecimal (base 16), octal (base 8) and binary (base 2).

- Decimal integer literal consists of a sequence of digits without a leading 0 (zero).
- Leading 0 (zero) on an integer literal, or leading 0o (or 0O) indicates it is in octal. Octal integers can include only the digits 0-7.
- Leading 0x (or 0X) indicates hexadecimal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F.
- Leading 0b (or 0B) indicates binary. Binary integers can include digits only 0 and 1.

```
0, 117 and -345 (decimal, base 10)
015, 0001 and -0077 (octal, base 8)
0x1123, 0x00111 and -0xF1A7 (hexadecimal, "hex" or base 16)
0b11, 0b0011 and -0b11 (binary, base 2)
```

- **Floating-point literals**

A floating-point literal can have the following parts:

- A decimal integer which can be signed (preceded by "+" or "-"),
- A decimal point ("."),
- A fraction (another decimal number),
- An exponent.

The exponent part is an "e" or "E" followed by an integer, which can be signed (preceded by "+" or "-"). A floating-point literal must have at least one digit and either a decimal point or "e" (or "E"). ````js // More succinctly, the syntax is:`

`[(+|-)][digits][.digits][(E|e)(+|-)]digits`

`// For example: 3.1415926 -.123456789 -3.1E+12 .1e-23`

- Object literals

> An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces (`{}`). You should not use an object literal at the beginning of a statement. This will lead to an error or not behave as you expect, because the `{` will be interpreted as the beginning of a block.

```
```js
var sales = "Toyota";

function carTypes(name) {
 if (name === "Honda") {
 return name;
 } else {
 return "Sorry, we don't sell " + name + ".";
 }
}

var car = { myCar: "Saturn", getCar: carTypes("Honda"), special: sales };

console.log(car.myCar); // Saturn
console.log(car.getCar); // Honda
console.log(car.special); // Toyota
```

#### • RegExp literals

A regex literal is a pattern enclosed between slashes. The following is an example of an regex literal.

```
var re = /ab+c/;
```

#### • String literals

A string literal is zero or more characters enclosed in double ("") or single ('') quotation marks. A string must be delimited by quotation marks of the same type; that is, either both single quotation marks or both double quotation marks. The following are examples of string literals:

```
"foo"
'bar'
"1234"
"one line \n another line"
"John's cat"
```

# Statements and Declaration

## Table of Content

1. Declaration
2. Control flow
3. Loop and Iterations

## Declaration

There are three kinds of declarations in JavaScript.

- `var` : Declares a variable, optionally initializing it to a value.
- `let` : (New in ES6) Declares a block scope local variable, optionally initializing it to a value.
- `const` : (New in ES6) Declares a read-only named constant.

## Variables

You use variables as symbolic names for values in your application. The names of variables, called identifiers, conform to certain rules.

A JavaScript identifier must start with a letter, underscore (`_`), or dollar sign (`$`); subsequent characters can also be digits (`0-9`). Because JavaScript is case sensitive, letters include the characters "`A`" through "`z`" (uppercase) and the characters "`a`" through "`z`" (lowercase).

## Declaring variables

You can declare a variable in three ways:

- With the keyword `var`. For example, `var x = 42` .  
This syntax can be used to declare both local and global variables.
- By simply assigning it a value. For example, `x = 42` .  
This always declares a global variable. It generates a strict JavaScript warning.  
**You shouldn't use this variant.**
- With the keyword `let` . For example, `let y = 13` .

This syntax can be used to declare a block scope local variable.

- A variable declared using the var or let statement with no initial value specified has the value undefined .

```
var a; // The value of a is undefined
```

- You can use undefined to determine whether a variable has a value.

```
var input;
if (input === undefined){
 doThis();
} else {
 doThat();
}
```

- The undefined value behaves as false when used in a boolean context.
- The undefined value converts to NaN when used in numeric context.
- When you evaluate a null variable, the null value behaves as 0 in numeric contexts and as false in boolean contexts.

## Variable scope

When you declare a variable outside of any function, it is called a global variable, because it is available to any other code in the current document. When you declare a variable within a function, it is called a local variable, because it is available only within that function.

JavaScript before ECMAScript 2015 does not have block statement scope; rather, a variable declared within a block is local to the function (or global scope) that the block resides within.

```
if (true) {
 var x = 5;
}
console.log(x); // x is 5
```

This behavior changes, when using the let declaration introduced in ECMAScript 2015.

```
if (true) {
 let y = 5;
}
console.log(y); // ReferenceError: y is not defined
```

## Variable hoisting

Another unusual thing about variables in JavaScript is that you can refer to a variable declared later, without getting an exception.

This concept is known as **hoisting**; variables in JavaScript are in a sense "hoisted" or lifted to the top of the function or statement.

- Variables that are hoisted will return a value of undefined.
- If you declare and initialize after you use or refer to this variable, it will still return `undefined`.

### Example:

```
/*
 * Example 1
 */
console.log(x === undefined); // true
var x = 3;

/*
 * Example 2
 */
// will return a value of undefined
var myvar = "my value";

(function() {
 console.log(myvar); // undefined
 var myvar = "local value";
})();
```

Because of hoisting, all `var` statements in a function should be placed as near to the top of the function as possible. **This best practice increases the clarity of the code.**

In ECMAScript 2015, `let` (`const`) will not hoist the variable to the top of the block.

## Control flow

JavaScript supports a compact set of statements, specifically control flow statements, that you can use to incorporate a great deal of interactivity in your application.

When your program contains more than one statement, the statements are executed, predictably, from top to bottom. As a basic example, this program has two statements. The first one asks the user for a number, and the second, which is executed afterward, shows the square of that number.

## Block statement

The most basic statement is a block statement that is used to group statements. The block is delimited by a pair of curly brackets:

```
{
 statement_1;
 statement_2;
 .
 .
 .
 statement_n;
}
```

- Block statements do not define a scope.
- "Standalone" blocks in JavaScript can produce completely different results from what they would produce in C or Java.

```
var x = 1;
{
 var x = 2;
}
console.log(x); // outputs 2
```

This outputs 2 because the `var x` statement within the block is in the same scope as the `var x` statement before the block. In C or Java, the equivalent code would have outputted 1.

## Conditional statements

A conditional statement is a set of commands that executes if a specified condition is true. JavaScript supports two conditional statements: `if...else` and `switch`.

### `if...else` statement

- Use the `if` statement to execute a statement if a logical condition is `true`.
- Use the optional `else` clause to execute a statement if the condition is `false`.

```
if (condition) {
 statement_1; // statement_1 is executed when condition is true
} else {
 statement_2; // statement_2 is executed when condition is false
}
```

- Using `else if` to have multiple conditions tested in sequence.

```
if (condition_1) {
 statement_1;
} else if (condition_2) {
 statement_2;
} else if (condition_n) {
 statement_n;
} else {
 statement_last;
}
```

- The following values evaluate to false (also known as *Falsy values*):

  - false
  - undefined
  - null
  - 0
  - NaN
  - the empty string ( "" )

- All other values, including all objects, evaluate to true when passed to a conditional statement.

## switch statement

A `switch` statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement.

```
switch (expression) {
 case label_1:
 statements_1
 [break];
 case label_2:
 statements_2
 [break];
 ...
 default:
 statements_def
 [break];
}
```

The program first looks for a case clause with a label matching the value of expression and then transfers control to that clause, executing the associated statements.

- If no matching label is found, the program looks for the optional `default` clause, and if found, transfers control to that clause, executing the associated statements.

- If no `default` clause is found, the program continues execution at the statement following the end of switch. By convention, the default clause is the last clause, but it does not need to be so.
- The optional `break` statement associated with each case clause ensures that the program breaks out of `switch` once the matched statement is executed and continues execution at the statement following `switch`.

```
switch (fruitype) {
 case "Oranges":
 console.log("Oranges are $0.59 a pound.");
 break;
 case "Apples":
 console.log("Apples are $0.32 a pound.");
 break;
 case "Bananas":
 console.log("Bananas are $0.48 a pound.");
 break;
 case "Cherries":
 console.log("Cherries are $3.00 a pound.");
 break;
 case "Mangoes":
 console.log("Mangoes are $0.56 a pound.");
 break;
 case "Papayas":
 console.log("Mangoes and papayas are $2.79 a pound.");
 break;
 default:
 console.log("Sorry, we are out of " + fruittype + ".");
}
console.log("Is there anything else you'd like?");
```

## Exception handling statements

You can throw exceptions using the `throw` statement and handle them using the `try...catch` statements.

**throw statement** Use the `throw` statement to throw an exception.

```
throw expression;
```

You may throw any expression, not just expressions of a specific type.

```
throw "Error2"; // String type
throw 42; // Number type
throw true; // Boolean type
throw {toString: function() { return "I'm an object!"; } };
```

**try...catch statement**

- The `try...catch` statement marks a block of statements to try, and specifies one or more responses should an exception be thrown.

```
try {
 throw "myException"; // generates an exception
}
catch (e) {
 // statements to handle any exceptions
 logMyErrors(e); // pass exception object to error handler
}
```

- The `finally` block contains statements to execute after the try and catch blocks execute but before the statements following the `try...catch` statement.

```
openMyFile();
try {
 writeMyFile(theData); //This may throw a error
} catch(e) {
 handleError(e); // If we got a error we handle it
} finally {
 closeMyFile(); // always close the resource
}
```

## Loop and Iterations

Loops offer a quick and easy way to do something repeatedly.

There are many different kinds of loops, but they all essentially do the same thing: they repeat an action some number of times (and it's actually possible that number could be zero).

The statements for loops provided in JavaScript are:

- `for` statement
- `do...while` statement
- `while` statement
- `labeled` statement
- `break` statement
- `continue` statement
- `for...in` statement

### for statement

A for loop repeats until a specified condition evaluates to false. The JavaScript for loop is similar to the Java and C for loop.

```
for ([initialExpression]; [condition]; [incrementExpression])
 statement
```

The `for` statement declares the variable `i` and initializes it to `zero`. It checks that `i` is less than the number of defined length, performs some statements, and increments `i` by one after each pass through the loop.

```
for (var i = 0; i < 10; i++) {
 console.log('I have made ' + i + ' steps.');
}
```

## do...while statement

The `do...while` statement repeats until a specified condition evaluates to false.

```
do
 statement
while (condition);
```

```
var i = 0;
do {
 i += 1;
 console.log(i);
} while (i < 5);
```

## while statement

A `while` statement executes its statements as long as a specified condition evaluates to true.

```
while (condition)
 statement
```

```
var n = 0;
var x = 0;
while (n < 3) {
 n++;
 x += n;
}
```

## **labeled statement**

A `label` provides a statement with an identifier that lets you refer to it elsewhere in your program.

```
label :
 statement
```

The label `markLoop` identifies a while loop.

```
markLoop:
while (theMark == true) {
 doSomething();
}
```

## **break statement**

Use the `break` statement to terminate a loop, `switch`, or in conjunction with a labeled statement.

```
break [label];
```

```
for (var i = 0; i < a.length; i++) {
 if (a[i] == theValue) {
 break;
 }
}

var x = 0;
var z = 0;
labelCancelLoops: while (true) {
 console.log("Outer loops: " + x);
 x += 1;
 z = 1;
 while (true) {
 console.log("Inner loops: " + z);
 z += 1;
 if (z === 10 && x === 10) {
 break labelCancelLoops;
 } else if (z === 10) {
 break;
 }
 }
}
```

## continue statement

The `continue` statement can be used to restart a `while`, `do...while`, `for`, or `label` statement.

```
continue [label];
```

```

var i = 0;
var n = 0;
while (i < 5) {
 i++;
 if (i == 3) {
 continue;
 }
 n += i;
}

checkiandj:
while (i < 4) {
 console.log(i);
 i += 1;
checkj:
 while (j > 4) {
 console.log(j);
 j -= 1;
 if ((j % 2) == 0) {
 continue checkj;
 }
 console.log(j + " is odd.");
 }
 console.log("i = " + i);
 console.log("j = " + j);
}

```

## for...in statement

The `for...in` statement iterates a specified variable over all the enumerable properties of an object. For each distinct property, JavaScript executes the specified statements.

```

for (variable in object) {
 statements
}

function dump_props(obj, obj_name) {
 var result = "";
 for (var i in obj) {
 result += obj_name + "." + i + " = " + obj[i] + "
";
 }
 result += "<hr>";
 return result;
}

```



# Expressions and Operators

## Table of Content

1. [Arithmetic Operators](#)
2. [Assignment Operators](#)
3. [String Operators](#)
4. [Comparison and Logical Operators](#)

## Arithmetic operators

An arithmetic operator takes numerical values (either literals or variables) as their operands and returns a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (\*), and division (/). These operators work as they do in most other programming languages when used with floating point numbers (in particular, note that division by zero produces Infinity).

```
1 / 2; // 0.5
1 / 2 == 1.0 / 2.0; // this is true
```

Operator	Description	Example
Remainder ( % )	Binary operator. Returns the integer remainder of dividing the two operands.	12 % 5 returns 2.
Increment ( ++ )	Unary operator. Adds one to its operand. If used as a prefix operator (++x), returns the value of its operand after adding one; if used as a postfix operator (x++), returns the value of its operand before adding one.	If x is 3, then ++x sets x to 4 and returns 4, whereas x++ returns 3 and, only then, sets x to 4.
Decrement ( -- )	Unary operator. Subtracts one from its operand. The return value is analogous to that for the increment operator.	If x is 3, then --x sets x to 2 and returns 2, whereas x-- returns 3 and, only then, sets x to 2.
Unary negation ( - )	Unary operator. Returns the negation of its operand.	If x is 3, then -x returns -3.
Unary plus ( + )	Unary operator. Attempts to convert the operand to a number, if it is not already.	+"3" returns 3. +true returns 1.

## Assignment operators

An assignment operator assigns a value to its left operand based on the value of its right operand. The simple assignment operator is equal (=), which assigns the value of its right operand to its left operand. That is,  $x = y$  assigns the value of y to x.

Name	Shorthand operator	Meaning
Assignment	<code>x = y</code>	<code>x = y</code>
Addition assignment	<code>x += y</code>	<code>x = x + y</code>
Subtraction assignment	<code>x -= y</code>	<code>x = x - y</code>
Multiplication assignment	<code>x *= y</code>	<code>x = x * y</code>
Division assignment	<code>x /= y</code>	<code>x = x / y</code>
Remainder assignment	<code>x %= y</code>	<code>x = x % y</code>
Exponentiation assignment	<code>x **= y</code>	<code>x = x ** y</code>
Left shift assignment	<code>x &lt;= y</code>	<code>x = x &lt;&lt; y</code>
Right shift assignment	<code>x &gt;= y</code>	<code>x = x &gt;&gt; y</code>
Unsigned right shift assignment	<code>&gt;&gt;&gt;= y</code>	<code>x = x &gt;&gt;&gt; y</code>
Bitwise AND assignment	<code>x &amp;= y</code>	<code>x = x &amp; y</code>
Bitwise XOR assignment	<code>x ^= y</code>	<code>x = x ^ y</code>
Bitwise OR assignment	<code>x  = y</code>	<code>x = x   y</code>

## String operators

In addition to the comparison operators, which can be used on string values, the concatenation operator (+) concatenates two string values together, returning another string that is the union of the two operand strings.

```
console.log("my " + "string"); // console logs the string "my string".
```

The shorthand assignment operator += can also be used to concatenate strings.

```
var mystring = "alpha";
mystring += "bet"; // evaluates to "alphabet" and assigns this value to mystring.
```

## Comparison operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true. The operands can be numerical, string, logical, or object values.

```
var var1 = 3;
var var2 = 4;
```

Operator	Description	Examples returning true
Equal ( == )	Returns true if the operands are equal.	<code>3 == var1</code> <code>"3" == var1</code> <code>3 == '3'</code>
Not equal ( != )	Returns true if the operands are not equal.	<code>var1 != 4</code> <code>var2 != "3"</code>
Strict equal ( === )	Returns true if the operands are equal and of the same type. See also Object.is and sameness in JS.	<code>3 === var1</code>
Strict not equal ( !== )	Returns true if the operands are of the same type but not equal, or are of different type.	<code>var1 !== "3"</code> <code>3 !== '3'</code>
Greater than ( > )	Returns true if the left operand is greater than the right operand.	<code>var2 &gt; var1</code> <code>"12" &gt; 2</code>
Greater than or equal ( >= )	Returns true if the left operand is greater than or equal to the right operand.	<code>var2 &gt;= var1</code> <code>var1 &gt;= 3</code>
Less than ( < )	Returns true if the left operand is less than the right operand.	<code>var1 &lt; var2</code> <code>"2" &lt; 12</code>
Less than or equal ( <= )	Returns true if the left operand is less than or equal to the right operand.	<code>var1 &lt; var2</code> <code>"2" &lt; 12</code>

## Logical operators

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value.

Operator	Usage	Description
Logical AND ( && )	<code>expr1 &amp;&amp; expr2</code>	Returns <code>expr1</code> if it can be converted to false; otherwise, returns <code>expr2</code> . Thus, when used with Boolean values, <code>&amp;&amp;</code> returns true if both operands are true; otherwise, returns false.
Logical OR (    )	<code>expr1    expr2</code>	Returns <code>expr1</code> if it can be converted to true; otherwise, returns <code>expr2</code> . Thus, when used with Boolean values, <code>  </code> returns true if either operand is true; if both are false, returns false.
Logical NOT ( ! )	<code>!expr</code>	Returns false if its single operand can be converted to true; otherwise, returns true.

The following code shows examples of the `&&` (logical AND) operator.

```
var a1 = true && true; // t && t returns true
var a2 = true && false; // t && f returns false
var a3 = false && true; // f && t returns false
var a4 = false && (3 == 4); // f && f returns false
var a5 = "Cat" && "Dog"; // t && t returns Dog
var a6 = false && "Cat"; // f && t returns false
var a7 = "Cat" && false; // t && f returns false
```

The following code shows examples of the || (logical OR) operator.

```
var o1 = true || true; // t || t returns true
var o2 = false || true; // f || t returns true
var o3 = true || false; // t || f returns true
var o4 = false || (3 == 4); // f || f returns false
var o5 = "Cat" || "Dog"; // t || t returns Cat
var o6 = false || "Cat"; // f || t returns Cat
var o7 = "Cat" || false; // t || f returns Cat
```

The following code shows examples of the ! (logical NOT) operator.

```
var n1 = !true; // !t returns false
var n2 = !false; // !f returns true
var n3 = !"Cat"; // !t returns false
```

# Functions

## Table of Content

1. Defining a function
2. Parameters and Scopes
3. The Function stack
4. Predefined Functions

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure—a set of statements that performs a task or calculates a value. To use a function, you must define it somewhere in the scope from which you wish to call it.

There are two more or less natural ways for functions to be introduced into programs.

- The first is that you find yourself writing very similar code multiple times.

We want to avoid doing that since having more code means more space for mistakes to hide and more material to read for people trying to understand the program. So we take the repeated functionality, find a good name for it, and put it into a function.

- The second way is that you find you need some functionality that you haven't written yet and that sounds like it deserves its own function.

You'll start by naming the function, and you'll then write its body. You might even start writing code that uses the function before you actually define the function itself.

How difficult it is to find a good name for a function is a good indication of how clear a concept it is that you're trying to wrap.

## Defining functions

### Function declarations

A function definition (also called a function declaration, or function statement) consists of the function keyword, followed by:

- The name of the function.

- A list of arguments to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly brackets, `{ }`.

```
function square(number) {
 return number * number;
}
```

## Function expressions

- While the function declaration above is syntactically a statement, functions can also be created by a function expression. Such a function can be anonymous; it does not have to have a name.

```
var square = function (number) { return number * number };
var x = square(4) // x gets the value 16
```

- However, a name can be provided with a function expression and can be used inside the function to refer to itself, or in a debugger to identify the function in stack traces:

```
var factorial = function fac(n) { return n < 2 ? 1 : n * fac(n-1) };
console.log(factorial(3));
```

- Function expressions are convenient when passing a function as an argument to another function.

```
function map(f, a) {
 var result = [], // Create a new Array
 i;
 for (i = 0; i != a.length; i++)
 result[i] = f(a[i]);
 return result;
}

var multiply = function (x) {return x * x * x}; // Expression function.
map(multiply, [0, 1, 2, 5, 10]); // [0, 1, 8, 125, 1000].
```

- In JavaScript, a function can be defined based on a condition.

```
var myFunc;
if (num === 0) {
 myFunc = function (theObject) {
 theObject.make = "Toyota"
 }
}
```

## Anonymous functions

An anonymous function is a function that is not given an identifier. Anonymous functions are mostly used for passing functions as a parameter to another function.

```
function () {
 console.log('hi');
}; // anonymous function, but no way to invoke it

// create a function that can invoke our anonymous function
var sayHi = function (f) {
 f(); // invoke anonymous function
}

// pass an anonymous function as parameter
sayHi(function () {
 console.log('hi');
}); // log 'hi'
```

## Self-invoking function expression

A function expression (really any function except one created from the `Function()` constructor) can be immediately invoked after definition by using the parentheses operator.

```
var sayWord = function () {
 console.log('Word 2 yo mo!');
}
()
;
// logs 'Word 2 yo mo!'
```

## Self-invoking anonymous function statements

It's possible to create an anonymous function statement that is self-invoked. This is called a self-invoking anonymous function.

```
// most commonly used/seen in the wild
(function(msg) {
 console.log(msg);
})('Hi');

// slightly different but achieving the same thing:
(function(msg) {
 console.log(msg)
}('Hi'));

// the shortest possible solution
function sayHi(msg) {
 console.log(msg);
}('Hi');

// And, this does NOT work!
function sayHi() {
 console.log('hi');
}();
```

According to the ECMAScript standard, the parentheses around the function (or anything that transforms the function into an expression) are required if the function is to be invoked immediately.

## Parameters and scopes

### Parameters

- The parameters to a function behave like regular variables, but their initial values are given by the caller of the function, not the code in the function itself.
- Parameters are vehicles for passing values into the scope of a function when it is invoked.

```
var addFunction = function(number1, number2) {
 var sum = number1 + number2;
 return sum;
}
console.log(addFunction(
3, 3
));
// logs 6
```

- It is perfectly legal in JavaScript to omit parameters even if the function has been defined to accept these arguments.

The missing parameters are simply given the value of `undefined`. Of course, by leaving out values for the parameters, the function might not work properly. - If you pass a function unexpected parameters (those not defined when the function was created), no error will occur, unless you validate acceptable values by for the function.

- And it's possible to access these parameters from the `arguments` object, which is available to all functions.

## Function scope

Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined.

```
// The following variables are defined in the global scope
var num1 = 20,
 num2 = 3,
 name = "Chamahk";

// This function is defined in the global scope
function multiply() {
 return num1 * num2;
}

multiply(); // Returns 60

// A nested function example
function getScore () {
 var num1 = 2,
 num2 = 3;

 function add() {
 return name + " scored " + (num1 + num2);
 }

 return add();
}

getScore(); // Returns "Chamahk scored 5"
```

## this & arguments

Inside the scope/body of all functions is available the `this` and `arguments` values.

- The `arguments` object is an array-like object containing all of the parameters being passed to the function.

```
var add = function () {
 return arguments[0] + arguments[1];
};
console.log(add(4, 4)); // returns 8
```

- The `this` keyword, passed to all functions, is a reference to the object that contains the function.

As you might expect, functions contained within objects as properties (i.e. methods) can use this to gain a reference to the "parent" object. When a function is defined in the global scope, the value of this is the global object.

```
var myObject1 = {
 name: 'myObject1',
 myMethod: function () {
 console.log(this);
 }
};

myObject1.myMethod(); // logs 'myObject1'

var myObject2 = function () {
 console.log(this);
};

myObject2(); // logs window
```

## Redefining function parameters

A function's parameters can be redefined inside the function either directly, or by using the `arguments` array.

```
var foo = false;
var bar = false;

var myFunction = function(foo, bar) {
 arguments[0] = true;
 bar = true;
 console.log(arguments[0], bar); // logs true true
}
myFunction();
```

Notice that the value of the `bar` parameter can redefine using the `arguments` index or by directly reassigning a new value to the parameter.

# The Function stack

## Cancel function execution (a.k.a return a function before it is done)

Functions can be cancelled at any time during invocation by using the `return` keyword with or without a value.

```
var add = function (x, y) {
 // If the parameters are not numbers, return error.
 if (typeof x !== 'number' || typeof y !== 'number') {
 return 'pass in numbers';
 }

 return x + y;
}

console.log(add(3,3)); // logs 6
console.log(add('2','2'));
```

The take away here is that you can cancel a function's execution by using the `return` keyword at any point in the execution of the function.

## Recursion

It is perfectly okay for a function to call itself, as long as it takes care not to overflow the stack. A function that calls itself is called *recursive*.

In some ways, recursion is analogous to a loop. Both execute the same code multiple times, and both require a condition (to avoid an infinite loop, or rather, infinite recursion in this case).

```
// Consider the following loop
var x = 0;
while (x < 10) { // "x < 10" is the loop condition
 // do stuff
 x++;
}

// It can be converted to
function loop(x) {
 if (x >= 10) // "x >= 10" is the exit condition (equivalent to "!(x < 10)")
 return;
 // do stuff
 loop(x + 1); // the recursive call
}
loop(0);
```

In typical JavaScript implementations, it's about 10 times slower than the looping version. Running through a simple loop is a lot cheaper than calling a function multiple times.

## Closures

You can nest a function within a function. The nested (inner) function is private to its containing (outer) function. It also forms a *closure*.

A closure is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

Since a nested function is a closure, this means that a nested function can "inherit" the arguments and variables of its containing function.

- The inner function can be accessed only from statements in the outer function.
- The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.

```
function addSquares(a,b) {
 function square(x) {
 return x * x;
 }
 return square(a) + square(b);
}
a = addSquares(2,3); // returns 13
b = addSquares(3,4); // returns 25
c = addSquares(4,5); // returns 41
```

Since the inner function forms a closure, you can call the outer function and specify arguments for both the outer and inner function:

```
function outside(x) {
 function inside(y) {
 return x + y;
 }
 return inside;
}
fn_inside = outside(3); // Think of it like: give me a function that adds 3 to whatever you give it
result = fn_inside(5); // returns 8

result1 = outside(3)(5); // returns 8
```

## Predefined functions

JavaScript has several top-level, built-in functions:

Functions	Explanation
<code>eval()</code>	The <code>eval()</code> method evaluates JavaScript code represented as a string.
<code>isFinite()</code>	The global <code>isFinite()</code> function determines whether the passed value is a finite number. If needed, the parameter is first converted to a number.
<code>isNaN()</code>	The <code>isNaN()</code> function determines whether a value is NaN or not. Note: coercion inside the <code>isNaN</code> function has interesting rules; you may alternatively want to use <code>Number.isNaN()</code> , as defined in ECMAScript 6, or you can use <code>typeof</code> to determine if the value is Not-A-Number.
<code>parseFloat()</code>	The <code>parseFloat()</code> function parses a string argument and returns a floating point number.
<code>parseInt()</code>	The <code>parseInt()</code> function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems).
<code>decodeURI()</code>	The <code>decodeURI()</code> function decodes a Uniform Resource Identifier (URI) previously created by <code>encodeURI</code> or by a similar routine.
<code>decodeURIComponent()</code>	The <code>decodeURIComponent()</code> method decodes a Uniform Resource Identifier (URI) component previously created by <code>encodeURIComponent</code> or by a similar routine.
<code>encodeURI()</code>	The <code>encodeURI()</code> method encodes a Uniform Resource Identifier (URI) by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character (will only be four escape sequences for characters composed of two "surrogate" characters).
<code>encodeURIComponent()</code>	The <code>encodeURIComponent()</code> method encodes a Uniform Resource Identifier (URI) component by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character (will only be four escape sequences for characters composed of two "surrogate" characters).

# Prototyping

## Table of Content

1. Re-introduction
2. Prototype-based programming
  - Namespace
  - The class
  - The object (class instance)
  - The constructor
  - The property (object attribute)
  - The method
  - Inheritance
  - Abstraction
3. Function Prototype Property
4. Terminology
  - Encapsulation
  - Polymorphism

## Re-introduction

### Why a re-introduction?

Because JavaScript is notorious for being the world's most misunderstood programming language. It is often derided as being a toy, but beneath its layer of deceptive simplicity, powerful language features await. JavaScript is now used by an incredible number of high-profile applications, showing that deeper knowledge of this technology is an important skill for any web or mobile developer.

Unlike most programming languages, the JavaScript language has no concept of input or output.

JavaScript is an object-oriented dynamic language with types and operators, standard built-in objects, and methods. Its syntax is based on the Java and C languages — so many structures from those languages apply to JavaScript as well.

## Prototype-based programming

Prototype-based programming is an OOP model that doesn't use classes, but rather it first accomplishes the behavior of any class and then reuses it (equivalent to inheritance in class-based languages) by decorating (or expanding upon) existing prototype objects. (Also called classless, prototype-oriented, or instance-based programming.)

JavaScript is a prototype-based programming language (probably prototype-based scripting language is more correct definition).

## Namespace

A namespace is a container which allows developers to bundle up functionality under a unique, application-specific name.

In JavaScript a namespace is just another object containing methods, properties, and objects.

**Note:** It's important to note that in JavaScript, there's no language-level difference between regular objects and namespaces. This differs from many other object-oriented languages, and can be a point of confusion for new JavaScript programmers.

Namespace in JavaScript is simple: create one global object, and all variables, methods, and functions become properties of that object.

```
// global namespace
var MYAPP = MYAPP || {};

// sub namespace
MYAPP.event = {};
```

## The class

- JavaScript is a prototype-based language and contains no `class` statement.
- JavaScript uses functions as constructors for classes.
- Defining a class is as easy as defining a function.

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

## The object (class instance)

To create a new instance of an object `obj` we use the statement `new obj`, assigning the result (which is of type `obj`) to a variable to access it later.

```
var person1 = new Person();
var person2 = new Person();
```

## The constructor

The constructor is called at the moment of instantiation (the moment when the object instance is created). The constructor is a method of the class.

- In JavaScript the function serves as the constructor of the object, therefore there is no need to explicitly define a constructor method. Every action declared in the class gets executed at the time of instantiation.
- The constructor is used to set the object's properties or to call methods to prepare the object for use.

```
// The constructor of the class Person logs a message when a Person is instantiated.

var Person = function () {
 console.log('instance created');
};

var person1 = new Person();
var person2 = new Person();
```

## The property (object attribute)

Properties are variables contained in the class; every instance of the object has those properties. Properties are set in the constructor (function) of the class so that they are created on each instance.

The keyword `this`, which refers to the current object, lets you work with properties from within the class.

```
var Person = function (firstName) {
 this.firstName = firstName;
 console.log('Person instantiated');
};

var person1 = new Person('Alice');
var person2 = new Person('Bob');

// Show the firstName properties of the objects
console.log('person1 is ' + person1.firstName); // logs "person1 is Alice"
console.log('person2 is ' + person2.firstName); // logs "person2 is Bob"
```

## The methods

Methods are functions (and defined like functions), but otherwise follow the same logic as properties. Calling a method is similar to accessing a property, but you add `()` at the end of the method name, possibly with arguments.

- To define a method, assign a function to a named property of the class's `prototype` property.

```
var Person = function (firstName) {
 this.firstName = firstName;
};

Person.prototype.sayHello = function() {
 console.log("Hello, I'm " + this.firstName);
};

var person1 = new Person("Alice");
var person2 = new Person("Bob");

// call the Person sayHello method.
person1.sayHello(); // logs "Hello, I'm Alice"
person2.sayHello(); // logs "Hello, I'm Bob"
```

- Methods are regular function objects bound to an object as a property, which means you can invoke methods "out of the context".

```

var Person = function (firstName) {
 this.firstName = firstName;
};

Person.prototype.sayHello = function() {
 console.log("Hello, I'm " + this.firstName);
};

var person1 = new Person("Alice");
var person2 = new Person("Bob");
var helloFunction = person1.sayHello;

// logs "Hello, I'm Alice"
person1.sayHello();

// logs "Hello, I'm Bob"
person2.sayHello();

// logs "Hello, I'm undefined" (or fails
// with a TypeError in strict mode)
helloFunction();

// logs true
console.log(helloFunction === person1.sayHello);

// logs true
console.log(helloFunction === Person.prototype.sayHello);

// logs "Hello, I'm Alice"
helloFunction.call(person1);

```

## Inheritance

Inheritance is a way to create a class as a specialized version of one or more classes .

The specialized class is commonly called the child, and the other class is commonly called the parent.

- JavaScript only supports single inheritance.
- In JavaScript you do this by assigning an instance of the parent class to the child class, and then specializing it.

```

// Define the Person constructor
var Person = function(firstName) {
 this.firstName = firstName;
};

// Add a couple of methods to Person.prototype
Person.prototype.walk = function(){
 console.log("I am walking!");
}

```

```

};

Person.prototype.sayHello = function(){
 console.log("Hello, I'm " + this.firstName);
};

// Define the Student constructor
function Student(firstName, subject) {
 // Call the parent constructor, making sure (using call)
 // that "this" is set correctly during the call
 Person.call(this, firstName);

 // Initialize our Student-specific properties
 this.subject = subject;
}

// Create a Student.prototype object that inherits from Person.prototype.
// Note: A common error here is to use "new Person()" to create the
// Student.prototype. That's incorrect for several reasons, not least
// that we don't have anything to give Person for the "firstName"
// argument. The correct place to call Person is above, where we call
// it from Student.
Student.prototype = Object.create(Person.prototype); // See note below

// Set the "constructor" property to refer to Student
Student.prototype.constructor = Student;

// Replace the "sayHello" method
Student.prototype.sayHello = function(){
 console.log("Hello, I'm " + this.firstName + ". I'm studying "
 + this.subject + ".");
};

// Add a "sayGoodBye" method
Student.prototype.sayGoodBye = function(){
 console.log("Goodbye!");
};

// Example usage:
var student1 = new Student("Janet", "Applied Physics");
student1.sayHello(); // "Hello, I'm Janet. I'm studying Applied Physics."
student1.walk(); // "I am walking!"
student1.sayGoodBye(); // "Goodbye!"

// Check that instanceof works correctly
console.log(student1 instanceof Person); // true
console.log(student1 instanceof Student); // true

```

## Abstraction

Abstraction is a mechanism that allows you to model the current part of the working problem, either by inheritance (specialization) or composition.

- JavaScript achieves specialization by inheritance, and composition by letting class instances be the values of other objects' attributes.
- The JavaScript Function class inherits from the Object class (this demonstrates specialization of the model) and the Function.prototype property is an instance of Object (this demonstrates composition).

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

// logs "foo is a Function: true"
console.log('foo is a Function: ' + (foo instanceof Function));

// logs "foo.prototype is an Object: true"
console.log('foo.prototype is an Object: ' + (foo.prototype instanceof Object));
```

## Function Prototype Property

### Conceptual overview of the

`prototype` chain

The `prototype` property is an object created by JavaScript for every `Function()` instance. Specifically, it links object instances created with the `new` keyword back to the constructor function that created them. This is done so that instances can share, or inherit, common methods and properties. Importantly, the sharing occurs during property lookup.

A prototype object is created for every function, regardless of whether you intend to use that function as a constructor.

### Why care about the

`prototype` property?

You should care about the `prototype` property for four reasons:

- Reason 1: The first reason is that the `prototype` property is used by the native constructor functions (e.g. `Object()`, `Array()`, `Function()`, etc.) to allow constructor instances to inherit properties and methods.

It is the mechanism that JavaScript itself uses to allow object instances to inherit properties and methods from the constructor function's `prototype` property. If you want to understand JavaScript better, you need to understand how JavaScript itself leverages the `prototype` object.

- Reason 2: When creating user-defined constructor functions, you can orchestrate inheritance the same way JavaScript native objects do.
- Reason 3: You might really dislike prototypal inheritance or prefer another pattern for object inheritance, but the reality is that someday you might have to edit or manage someone else's code who thought prototypal inheritance was the bee's knees.

When this happens, you should be aware of how prototypal inheritance works, as well as how it can be replicated by developers who make use of custom constructor functions.

- Reason 4: By using prototypal inheritance, you can create efficient object instances that all leverage the same methods.

## `Prototype`

`is standard on all Function() ` instances`

- All functions are created from a `Function()` constructor, even if you do not directly invoke the `Function()` constructor (e.g. `var add = new Function('x', 'y', 'return x + z');` ) and instead use the literal notation (e.g. `var add = function(x,y){return x + z};` ).
- When a function instance is created, it is always given a `prototype` property, which is an empty object.

```
var myFunction = function() {};
console.log(
myFunction.prototype
);
// logs object{}
console.log(typeof
myFunction.prototype
);
// logs 'object'
```

Make sure you completely understand that the `prototype` property is coming from the `Function()` constructor.

## The default

`prototype` property is an `Object()` object

All this `prototype` talk can get a bit heavy. Truly, `prototype` is just an empty object property called "prototype" created behind the scenes by JavaScript and made available by invoking the `Function()` constructor.

```
var myFunction = function() {};
myFunction.prototype = {};
// add the prototype property and set it to an empty object
console.log(
myFunction.prototype
);
// logs an empty object
```

**Note:** The value of a prototype property can be set to any of the complex values (i.e. objects) available in JavaScript. JavaScript will ignore any prototype property set to a primitive value.

## Terminology

### Encapsulation

A technique which involves bundling the data and the methods that use the data together.

### Polymorphism

Poly means "many" and morphism means "forms". Different classes might define the same method or property.

# Built-in Objects

## Table of Content

1. Fundamental Objects
2. Numbers and Dates
3. Text Processing
4. Indexed Collections
5. Structured Data

## Fundamental objects

These are the fundamental, basic objects upon which all other objects are based. This includes objects that represent general objects, functions, and errors.

### Object

The `Object` constructor creates an object wrapper for the given value. If the value is `null` or `undefined`, it will create and return an empty object, otherwise, it will return an object of a Type that corresponds to the given value. If the value is an object already, it will return the value.

#### Syntax

```
// Object initialiser or literal
{ [nameValuePair1[, nameValuePair2[, ...nameValuePairN]]] }

// Called as a constructor
new Object([value])
```

#### Parameters

- `nameValuePair1` , `nameValuePair2` , ... `nameValuePairN` : Pairs of names (strings) and values (any value) where the name is separated from the value by a colon.
- `value` : Any value.

### Function

Function objects created with the `Function` constructor are parsed when the function is created. This is less efficient than declaring a function with a function expression or function statement and calling it within your code, because such functions are parsed with the rest of the code.

All arguments passed to the function are treated as the names of the identifiers of the parameters in the function to be created, in the order in which they are passed.

## Syntax

```
new Function ([arg1[, arg2[, ...argN]]], functionBody)
```

## Parameters

- `arg1, arg2, ... argN` : Names to be used by the function as formal argument names. Each must be a string that corresponds to a valid JavaScript identifier or a list of such strings separated with a comma; for example "x", "theValue", or "a,b".
- `functionBody` : A string containing the JavaScript statements comprising the function definition.

## Boolean

The `Boolean` object is an object wrapper for a boolean value. The value passed as the first parameter is converted to a boolean value, if necessary. If value is omitted or is `0`, `-0`, `null`, `false`, `Nan`, `undefined`, or the empty string (`""`), the object has an initial value of `false`. All other values, including any object or the string `"false"`, create an object with an initial value of `true`.

## Syntax

```
new Boolean([value])
```

## Parameters

- `value` : Optional. The initial value of the Boolean object.

## Error

The `Error` constructor creates an error object. Instances of Error objects are thrown when runtime errors occur. The Error object can also be used as a base object for user-defined exceptions.

## Syntax

```
new Error([message])
```

## Parameters

- `message` : Optional. Human-readable description of the error.

# Numbers and dates

These are the base objects representing numbers, dates, and mathematical calculations.

## Number

The `Number` JavaScript object is a wrapper object allowing you to work with numerical values. A `Number` object is created using the `Number()` constructor.

The primary uses for the `Number` object are:

- If the argument cannot be converted into a number, it returns `Nan`.
- In a non-constructor context (i.e., without the `new` operator), `Number` can be used to perform a type conversion.

## Syntax

```
new Number(value);
```

## Parameters

- `value` : The numeric value of the object being created.

## Math

`Math` is a built-in object that has properties and methods for mathematical constants and functions. Not a function object.

Unlike the other global objects, `Math` is not a constructor. All properties and methods of `Math` are static. You refer to the constant pi as `Math.PI` and you call the sine function as `Math.sin(x)`, where `x` is the method's argument. Constants are defined with the full precision of real numbers in JavaScript.

## Date

Creates a JavaScript `Date` instance that represents a single moment in time. `Date` objects are based on a time value that is the number of milliseconds since 1 January, 1970 UTC.

- If no arguments are provided, the constructor creates a JavaScript Date object for the current date and time according to system settings.
- If at least two arguments are supplied, missing arguments are either set to 1 (if day is missing) or 0 for all others.
- The JavaScript date is based on a time value that is milliseconds since midnight 01 January, 1970 UTC. A day holds 86,400,000 milliseconds. The JavaScript Date object range is -100,000,000 days to 100,000,000 days relative to 01 January, 1970 UTC.
- The JavaScript Date object provides uniform behavior across platforms. The time value can be passed between systems to create a date that represents the same moment in time.
- The JavaScript Date object supports a number of UTC (universal) methods, as well as local time methods. UTC, also known as Greenwich Mean Time (GMT), refers to the time as set by the World Time Standard. The local time is the time known to the computer where JavaScript is executed.
- Invoking JavaScript Date as a function (i.e., without the new operator) will return a string representing the current date and time.

## Syntax

```
new Date();
new Date(value);
new Date(dateString);
new Date(year, month[, date[, hours[, minutes[, seconds[, milliseconds]]]]]);
```

## Parameters

- `value` : Integer value representing the number of milliseconds since 1 January 1970 00:00:00 UTC, with leap seconds ignored (Unix Epoch; but consider that most Unix time stamp functions count in seconds).
- `dateString` : String value representing a date. The string should be in a format recognized by the `Date.parse()` method.
- `year` : Integer value representing the year. Values from 0 to 99 map to the years 1900 to 1999.
- `month` : Integer value representing the month, beginning with 0 for January to 11 for December.
- `date` : Optional. Integer value representing the day of the month.
- `hours` : Optional. Integer value representing the hour of the day.
- `minutes` : Optional. Integer value representing the minute segment of a time.
- `seconds` : Optional. Integer value representing the second segment of a time.

- `milliseconds` : Optional. Integer value representing the millisecond segment of a time.

## Text processing

These objects represent strings and support manipulating them.

### String

The `String` global object is a constructor for strings, or a sequence of characters.

Strings are useful for holding data that can be represented in text form. Some of the most-used operations on strings are to check their `length`, to build and concatenate them using the `+` and `+=` string operators, checking for the existence or location of substrings with the `indexOf()` method, or extracting substrings with the `substring()` method.

#### Syntax

String literals take the forms:

```
'string text'
"string text"
"中文 español deutsch English हिन्दी العربية português बাংলা русский 日本語 மூலம் 한국어 தமிழ்
עברית"
```

Strings can also be created using the `String` global object directly:

```
String(thing)
```

#### Parameters

- `thing` : Anything to be converted to a string.

## RegExp

The `RegExp` constructor creates a regular expression object for matching text with a pattern.

#### Syntax

Literal and constructor notations are possible:

```
/pattern	flags
new RegExp(pattern[, flags])
```

## Parameters

- `pattern` : The text of the regular expression.
- `flags` : If specified, flags can have any combination of the following values:
  - `g` : global match; find all matches rather than stopping after the first match
  - `i` : ignore case
  - `m` : multiline; treat beginning and end characters (^ and \$) as working over multiple lines (i.e., match the beginning or end of each line (delimited by \n or \r), not only the very beginning or end of the whole input string)
  - `u` : unicode; treat pattern as a sequence of unicode code points
  - `y` : sticky; matches only from the index indicated by the lastIndex property of this regular expression in the target string (and does not attempt to match from any later indexes).

# Indexed collections

These objects represent collections of data which are ordered by an index value. This includes (typed) arrays and array-like constructs.

## Array

The JavaScript `Array` object is a global object that is used in the construction of arrays; which are high-level, list-like objects.

Arrays are list-like objects whose prototype has methods to perform traversal and mutation operations. Neither the length of a JavaScript array nor the types of its elements are fixed. Since an array's length can change at any time, and data can be stored at non-contiguous locations in the array, JavaScript arrays are not guaranteed to be dense; this depends on how the programmer chooses to use them. In general, these are convenient characteristics; but if these features are not desirable for your particular use, you might consider using typed arrays.

### Syntax

```
[element0, element1, ..., elementN]
new Array(element0, element1[, ...[, elementN]])
new Array(arrayLength)
```

## Parameters

- `elementN` : A JavaScript array is initialized with the given elements, except in the case

where a single argument is passed to the Array constructor and that argument is a number (see the arrayLength parameter below). Note that this special case only applies to JavaScript arrays created with the Array constructor, not array literals created with the bracket syntax.

- `arrayLength` : If the only argument passed to the Array constructor is an integer between 0 and 2<sup>32</sup>-1 (inclusive), this returns a new JavaScript array with length set to that number. If the argument is any other number, a RangeError exception is thrown.
- Create an Array

```
var fruits = ["Apple", "Banana"];

console.log(fruits.length);
// 2
```

- Access (index into) an Array item  
```js var first = fruits[0]; // Apple

```
var last = fruits[fruits.length - 1]; // Banana
```

- Loop over an Array
```js  
fruits.forEach(function (item, index, array) {  
 console.log(item, index);  
});  
// Apple 0  
// Banana 1

- Add to the end of an Array

```
var newLength = fruits.push("Orange");
// ["Apple", "Banana", "Orange"]
```

- Remove from the end of an Array

```
var last = fruits.pop(); // remove Orange (from the end)
// ["Apple", "Banana"];
```

- Remove from the front of an Array

```
var first = fruits.shift(); // remove Apple from the front
// ["Banana"];
```

- Add to the front of an Array

```
var newLength = fruits.unshift("Strawberry") // add to the front
// ["Strawberry", "Banana"];
```

- Find the index of an item in the Array ```js fruits.push("Mango"); // ["Strawberry", "Banana", "Mango"]`

```
var pos = fruits.indexOf("Banana"); // 1
```

- Remove an item by Index Position

```
``js
var removedItem = fruits.splice(pos, 1); // this is how to remove an item,
// ["Strawberry", "Mango"]
```

- Remove items from an Index Position ```js var removedItems = fruits.splice(pos, n); // this is how to remove items, n defines the number of items to be removed,`

```
// from that position onward to the end of a
array.
```

```
// let, n = 1;
```

```
// ["Strawberry"]
```

- Copy an Array

```
``js
var shallowCopy = fruits.slice(); // this is how to make a copy
// ["Strawberry"]
```

## Structured data

These objects represent and interact with structured data buffers and data coded using JavaScript Object Notation (JSON).

### JSON

The JSON object contains methods for parsing JavaScript Object Notation (JSON) and converting values to JSON. It can't be called or constructed, and aside from its two method properties it has no interesting functionality of its own.



# Conventions

## Table of Content

1. Types
2. Objects
3. Arrays
4. Strings
5. Functions
6. Properties
7. Variables
8. Hoisting
9. Comparison Operators & Equality
10. Blocks
11. Comments
12. Whitespace
13. Commas
14. Semicolons
15. Type Casting & Coercion
16. Naming Conventions
17. Accessors
18. Constructors
19. Events
20. Modules
21. jQuery
22. Testing

## Types

- **Primitives:** When you access a primitive type you work directly on its value.

- string
- number
- boolean
- null
- undefined

```
var foo = 1;
var bar = foo;

bar = 9;

console.log(foo, bar); // => 1, 9
```

- **Complex:** When you access a complex type you work on a reference to its value.

- object
- array
- function

```
var foo = [1, 2];
var bar = foo;

bar[0] = 9;

console.log(foo[0], bar[0]); // => 9, 9
```

↑ back to top

## Objects

- Use the literal syntax for object creation.

```
// bad
var item = new Object();

// good
var item = {};
```

- Don't use [reserved words](#) as keys. It won't work in IE8. [More info.](#)

```
// bad
var superman = {
 default: { clark: 'kent' },
 private: true
};

// good
var superman = {
 defaults: { clark: 'kent' },
 hidden: true
};
```

- Use readable synonyms in place of reserved words.

```
// bad
var superman = {
 class: 'alien'
};

// bad
var superman = {
 klass: 'alien'
};

// good
var superman = {
 type: 'alien'
};
```

[↑ back to top](#)

## Arrays

- Use the literal syntax for array creation.

```
// bad
var items = new Array();

// good
var items = [];
```

- Use `Array#push` instead of direct assignment to add items to an array.

```
```javascript
var someStack = [];
```

```
// bad
someStack[someStack.length] = 'abracadabra';

// good
someStack.push('abracadabra');
...;
```

- When you need to copy an array use `Array#slice`. [jsPerf](#)

```

var len = items.length;
var itemsCopy = [];
var i;

// bad
for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}

// good
itemsCopy = items.slice();

```

- To convert an array-like object to an array, use `Array#slice`.

```

function trigger() {
  var args = Array.prototype.slice.call(arguments);
  ...
}

```

[↑ back to top](#)

Strings

- Use single quotes `' '` for strings.

```

// bad
var name = "Bob Parr";

// good
var name = 'Bob Parr';

// bad
var fullName = "Bob " + this.lastName;

// good
var fullName = 'Bob ' + this.lastName;

```

- Strings longer than 100 characters should be written across multiple lines using string concatenation.
- Note: If overused, long strings with concatenation could impact performance. [jsPerf](#) & [Discussion](#).

```
// bad
var errorMessage = 'This is a super long error that was thrown because of Batman.
When you stop to think about how Batman had anything to do with this, you would get nowhere fast.';

// bad
var errorMessage = 'This is a super long error that was thrown because \
of Batman. When you stop to think about how Batman had anything to do \
with this, you would get nowhere \
fast.';

// good
var errorMessage = 'This is a super long error that was thrown because ' +
'of Batman. When you stop to think about how Batman had anything to do ' +
'with this, you would get nowhere fast.';
```

- When programmatically building up a string, use `Array#join` instead of string concatenation. Mostly for IE: [jsPerf](#).

```

var items;
var messages;
var length;
var i;

messages = [{ 
    state: 'success',
    message: 'This one worked.'
}, { 
    state: 'success',
    message: 'This one worked as well.'
}, { 
    state: 'error',
    message: 'This one did not work.'
}];

length = messages.length;

// bad
function inbox(messages) {
  items = '<ul>';

  for (i = 0; i < length; i++) {
    items += '<li>' + messages[i].message + '</li>';
  }

  return items + '</ul>';
}

// good
function inbox(messages) {
  items = [];

  for (i = 0; i < length; i++) {
    // use direct assignment in this case because we're micro-optimizing.
    items[i] = '<li>' + messages[i].message + '</li>';
  }

  return '<ul>' + items.join('') + '</ul>';
}

```

[↑ back to top](#)

Functions

- Function expressions:

```
// anonymous function expression
var anonymous = function () {
    return true;
};

// named function expression
var named = function named() {
    return true;
};

// immediately-invoked function expression (IIFE)
(function () {
    console.log('Welcome to the Internet. Please follow me.');
}());
```

- Never declare a function in a non-function block (if, while, etc). Assign the function to a variable instead. Browsers will allow you to do it, but they all interpret it differently, which is bad news bears.
- **Note:** ECMA-262 defines a `block` as a list of statements. A function declaration is not a statement. [Read ECMA-262's note on this issue](#).

```
// bad
if (currentUser) {
    function test() {
        console.log('Nope.');
    }
}

// good
var test;
if (currentUser) {
    test = function test() {
        console.log('Yup.');
    };
}
```

- Never name a parameter `arguments`. This will take precedence over the `arguments` object that is given to every function scope.

```
// bad
function nope(name, options, arguments) {
  // ...stuff...
}

// good
function yup(name, options, args) {
  // ...stuff...
}
```

[↑ back to top](#)

Properties

- Use dot notation when accessing properties.

```
var luke = {
  jedi: true,
  age: 28
};

// bad
var isJedi = luke['jedi'];

// good
var isJedi = luke.jedi;
```

- Use subscript notation `[]` when accessing properties with a variable.

```
var luke = {
  jedi: true,
  age: 28
};

function getProp(prop) {
  return luke[prop];
}

var isJedi = getProp('jedi');
```

[↑ back to top](#)

Variables

- Always use `var` to declare variables. Not doing so will result in global variables. We want to avoid polluting the global namespace. Captain Planet warned us of that.

```
// bad
superPower = new SuperPower();

// good
var superPower = new SuperPower();
```

- Use one `var` declaration per variable. It's easier to add new variable declarations this way, and you never have to worry about swapping out a `; for a ,` or introducing punctuation-only diffs.

```
// bad
var items = getItems(),
    goSportsTeam = true,
    dragonball = 'z';

// bad
// (compare to above, and try to spot the mistake)
var items = getItems(),
    goSportsTeam = true;
    dragonball = 'z';

// good
var items = getItems();
var goSportsTeam = true;
var dragonball = 'z';
```

- Declare unassigned variables last. This is helpful when later on you might need to assign a variable depending on one of the previous assigned variables.

```
// bad
var i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// bad
var i;
var items = getItems();
var dragonball;
var goSportsTeam = true;
var len;

// good
var items = getItems();
var goSportsTeam = true;
var dragonball;
var length;
var i;
```

- Assign variables at the top of their scope. This helps avoid issues with variable declaration and assignment hoisting related issues.

```
// bad
function () {
    test();
    console.log('doing stuff..');

    //...other stuff..

    var name = getName();

    if (name === 'test') {
        return false;
    }

    return name;
}

// good
function () {
    var name = getName();

    test();
    console.log('doing stuff..');

    //...other stuff..

    if (name === 'test') {
        return false;
    }
}
```

```
    return name;
}

// bad - unnecessary function call
function () {
    var name = getName();

    if (!arguments.length) {
        return false;
    }

    this.setFirstName(name);

    return true;
}

// good
function () {
    var name;

    if (!arguments.length) {
        return false;
    }

    name = getName();
    this.setFirstName(name);

    return true;
}
```

[↑ back to top](#)

Hoisting

- Variable declarations get hoisted to the top of their scope, but their assignment does not.

```
// we know this wouldn't work (assuming there
// is no notDefined global variable)
function example() {
    console.log(notDefined); // => throws a ReferenceError
}

// creating a variable declaration after you
// reference the variable will work due to
// variable hoisting. Note: the assignment
// value of `true` is not hoisted.
function example() {
    console.log(declaredButNotAssigned); // => undefined
    var declaredButNotAssigned = true;
}

// The interpreter is hoisting the variable
// declaration to the top of the scope,
// which means our example could be rewritten as:
function example() {
    var declaredButNotAssigned;
    console.log(declaredButNotAssigned); // => undefined
    declaredButNotAssigned = true;
}
```

- Anonymous function expressions hoist their variable name, but not the function assignment.

```
function example() {
    console.log(anonymous); // => undefined

    anonymous(); // => TypeError anonymous is not a function

    var anonymous = function () {
        console.log('anonymous function expression');
    };
}
```

- Named function expressions hoist the variable name, not the function name or the function body.

```

function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  superPower(); // => ReferenceError superPower is not defined

  var named = function superPower() {
    console.log('Flying');
  };
}

// the same is true when the function name
// is the same as the variable name.
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  var named = function named() {
    console.log('named');
  };
}

```

- Function declarations hoist their name and the function body.

```

function example() {
  superPower(); // => Flying

  function superPower() {
    console.log('Flying');
  }
}

```

- For more information refer to [JavaScript Scoping & Hoisting by Ben Cherry](#).

[↑ back to top](#)

Comparison Operators & Equality

- Use `==` and `!=` over `==` and `!=`.
- Conditional statements such as the `if` statement evaluate their expression using coercion with the `ToBoolean` abstract method and always follow these simple rules:
 - **Objects** evaluate to **true**
 - **Undefined** evaluates to **false**

- **Null** evaluates to **false**
- **Booleans** evaluate to **the value of the boolean**
- **Numbers** evaluate to **false** if **+0, -0, or NaN**, otherwise **true**
- **Strings** evaluate to **false** if an empty string `''`, otherwise **true**

```
if ([0]) {
  // true
  // An array is an object, objects evaluate to true
}
```

- Use shortcuts.

```
// bad
if (name !== '') {
  // ...stuff...
}

// good
if (name) {
  // ...stuff...
}

// bad
if (collection.length > 0) {
  // ...stuff...
}

// good
if (collection.length) {
  // ...stuff...
}
```

- For more information see [Truth Equality and JavaScript](#) by Angus Croll.

[↑ back to top](#)

Blocks

- Use braces with all multi-line blocks.

```
// bad
if (test)
    return false;

// good
if (test) return false;

// good
if (test) {
    return false;
}

// bad
function () { return false; }

// good
function () {
    return false;
}
```

- If you're using multi-line blocks with `if` and `else`, put `else` on the same line as your `if` block's closing brace.

```
// bad
if (test) {
    thing1();
    thing2();
}
else {
    thing3();
}

// good
if (test) {
    thing1();
    thing2();
} else {
    thing3();
}
```

[↑ back to top](#)

Comments

- Use `/** ... */` for multi-line comments. Include a description, specify types and values for all parameters and return values.

```
// bad
// make() returns a new element
// based on the passed in tag name
//
// @param {String} tag
// @return {Element} element
function make(tag) {

    // ...stuff...

    return element;
}

// good
/**
 * make() returns a new element
 * based on the passed in tag name
 *
 * @param {String} tag
 * @return {Element} element
 */
function make(tag) {

    // ...stuff...

    return element;
}
```

- Use `//` for single line comments. Place single line comments on a newline above the subject of the comment. Put an empty line before the comment.

```
// bad
var active = true; // is current tab

// good
// is current tab
var active = true;

// bad
function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  var type = this._type || 'no type';

  return type;
}

// good
function getType() {
  console.log('fetching type...');

  // set the default type to 'no type'
  var type = this._type || 'no type';

  return type;
}
```

- Prefixing your comments with `FIXME` or `TODO` helps other developers quickly understand if you're pointing out a problem that needs to be revisited, or if you're suggesting a solution to the problem that needs to be implemented. These are different than regular comments because they are actionable. The actions are `FIXME -- need to figure this out` or `TODO -- need to implement`.
- Use `// FIXME:` to annotate problems.

```
function Calculator() {

  // FIXME: shouldn't use a global here
  total = 0;

  return this;
}
```

- Use `// TODO:` to annotate solutions to problems.

```
function Calculator() {  
  
    // TODO: total should be configurable by an options param  
    this.total = 0;  
  
    return this;  
}
```

[↑ back to top](#)

Whitespace

- Use soft tabs set to 2 spaces.

```
// bad  
function () {  
    ...var name;  
}  
  
// bad  
function () {  
    .var name;  
}  
  
// good  
function () {  
    ..var name;  
}
```

- Place 1 space before the leading brace.

```
// bad
function test(){
  console.log('test');
}

// good
function test() {
  console.log('test');
}

// bad
dog.set('attr',{
  age: '1 year',
  breed: 'Bernese Mountain Dog'
});

// good
dog.set('attr', {
  age: '1 year',
  breed: 'Bernese Mountain Dog'
});
```

- Place 1 space before the opening parenthesis in control statements (`if`, `while` etc.).
Place no space before the argument list in function calls and declarations.

```
// bad
if(isJedi) {
  fight ();
}

// good
if (isJedi) {
  fight();
}

// bad
function fight () {
  console.log ('Swoosh!');
}

// good
function fight() {
  console.log('Swoosh!');
}
```

- Set off operators with spaces.

```
// bad  
var x=y+5;  
  
// good  
var x = y + 5;
```

- End files with a single newline character.

```
// bad  
(function (global) {  
    // ...stuff...  
})(this);
```

```
// bad  
(function (global) {  
    // ...stuff...  
})(this);↵
```

```
// good  
(function (global) {  
    // ...stuff...  
})(this);↵
```

- Use indentation when making long method chains. Use a leading dot, which emphasizes that the line is a method call, not a new statement.

```
// bad
$( '#items' ).find( '.selected' ).highlight().end().find( '.open' ).updateCount();  
  
// bad
$( '#items' ).  
  find( '.selected' ).  
    highlight().  
    end().  
  find( '.open' ).  
    updateCount();  
  
// good
$( '#items' )  
  .find( '.selected' )  
    .highlight()  
    .end()  
  .find( '.open' )  
    .updateCount();  
  
// bad
var leds = stage.selectAll( '.led' ).data(data).enter().append( 'svg:svg' ).classed('led', true)  
  .attr('width', (radius + margin) * 2).append('svg:g')  
  .attr('transform', 'translate(' + (radius + margin) + ', ' + (radius + margin)  
+ ')')  
  .call(tron.led);  
  
// good
var leds = stage.selectAll( '.led' )  
  .data(data)  
  .enter().append('svg:svg')  
    .classed('led', true)  
    .attr('width', (radius + margin) * 2)  
  .append('svg:g')  
    .attr('transform', 'translate(' + (radius + margin) + ', ' + (radius + margin)  
+ ')')  
  .call(tron.led);
```

- Leave a blank line after blocks and before the next statement

```
// bad
if (foo) {
    return bar;
}

return baz;

// good
if (foo) {
    return bar;
}

return baz;

// bad
var obj = {
    foo: function () {
    },
    bar: function () {
    }
};
return obj;

// good
var obj = {
    foo: function () {
    },

    bar: function () {
    }
};

return obj;
```

[↑ back to top](#)

Commas

- Leading commas: **Nope.**

```
// bad
var story = [
  once
, upon
, aTime
];

// good
var story = [
  once,
  upon,
  aTime
];

// bad
var hero = {
  firstName: 'Bob'
, lastName: 'Parr'
, heroName: 'Mr. Incredible'
, superPower: 'strength'
};

// good
var hero = {
  firstName: 'Bob',
  lastName: 'Parr',
  heroName: 'Mr. Incredible',
  superPower: 'strength'
};
```

- Additional trailing comma: **Nope**. This can cause problems with IE6/7 and IE9 if it's in quirksmode. Also, in some implementations of ES3 would add length to an array if it had an additional trailing comma. This was clarified in ES5 ([source](#)):

Edition 5 clarifies the fact that a trailing comma at the end of an ArrayInitialiser does not add to the length of the array. This is not a semantic change from Edition 3 but some implementations may have previously misinterpreted this.

```
// bad
var hero = {
  firstName: 'Kevin',
  lastName: 'Flynn',
};

var heroes = [
  'Batman',
  'Superman',
];

// good
var hero = {
  firstName: 'Kevin',
  lastName: 'Flynn'
};

var heroes = [
  'Batman',
  'Superman'
];
```

[↑ back to top](#)

Semicolons

- **Yup.**

```
// bad
(function () {
  var name = 'Skywalker'
  return name
})()

// good
(function () {
  var name = 'Skywalker';
  return name;
})();

// good (guards against the function becoming an argument when two files with IIFE
// s are concatenated)
;(function () {
  var name = 'Skywalker';
  return name;
})();
```

[Read more.](#)

[↑ back to top](#)

Type Casting & Coercion

- Perform type coercion at the beginning of the statement.
- Strings:

```
// => this.reviewScore = 9;

// bad
var totalScore = this.reviewScore + '';

// good
var totalScore = '' + this.reviewScore;

// bad
var totalScore = '' + this.reviewScore + ' total score';

// good
var totalScore = this.reviewScore + ' total score';
```

- Use `parseInt` for Numbers and always with a radix for type casting.

```
var inputValue = '4';

// bad
var val = new Number(inputValue);

// bad
var val = +inputValue;

// bad
var val = inputValue >> 0;

// bad
var val = parseInt(inputValue);

// good
var val = Number(inputValue);

// good
var val = parseInt(inputValue, 10);
```

- If for whatever reason you are doing something wild and `parseInt` is your bottleneck and need to use Bitshift for [performance reasons](#), leave a comment explaining why and what you're doing.

```
// good
/**
 * parseInt was the reason my code was slow.
 * Bitshifting the String to coerce it to a
 * Number made it a lot faster.
 */
var val = inputValue >> 0;
```

- **Note:** Be careful when using bitshift operations. Numbers are represented as [64-bit values](#), but Bitshift operations always return a 32-bit integer ([source](#)). Bitshift can lead to unexpected behavior for integer values larger than 32 bits. [Discussion](#). Largest signed 32-bit Int is 2,147,483,647:

```
2147483647 >> 0 //=> 2147483647
2147483648 >> 0 //=> -2147483648
2147483649 >> 0 //=> -2147483647
```

- Booleans:

```
var age = 0;

// bad
var hasAge = new Boolean(age);

// good
var hasAge = Boolean(age);

// good
var hasAge = !!age;
```

[↑ back to top](#)

Naming Conventions

- Avoid single letter names. Be descriptive with your naming.

```
// bad
function q() {
  // ...stuff...
}

// good
function query() {
  // ..stuff..
}
```

- Use camelCase when naming objects, functions, and instances.

```
// bad
var OBJEcttssss = {};
var this_is_my_object = {};
var o = {};
function c() {}

// good
var thisIsMyObject = {};
function thisIsMyFunction() {}
```

- Use PascalCase when naming constructors or classes.

```
// bad
function user(options) {
  this.name = options.name;
}

var bad = new user({
  name: 'nope'
});

// good
function User(options) {
  this.name = options.name;
}

var good = new User({
  name: 'yup'
});
```

- Use a leading underscore `_` when naming private properties.

```
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';

// good
this._firstName = 'Panda';
```

- When saving a reference to `this` use `_this`.

```
// bad
function () {
  var self = this;
  return function () {
    console.log(self);
  };
}

// bad
function () {
  var that = this;
  return function () {
    console.log(that);
  };
}

// good
function () {
  var _this = this;
  return function () {
    console.log(_this);
  };
}
```

- Name your functions. This is helpful for stack traces.

```
// bad
var log = function (msg) {
  console.log(msg);
};

// good
var log = function log(msg) {
  console.log(msg);
};
```

- **Note:** IE8 and below exhibit some quirks with named function expressions. See <http://kangax.github.io/nfe/> for more info.

- If your file exports a single class, your filename should be exactly the name of the class.

```
// file contents
class CheckBox {
    // ...
}

module.exports = CheckBox;

// in some other file
// bad
var CheckBox = require('./checkBox');

// bad
var CheckBox = require('./check_box');

// good
var CheckBox = require('./CheckBox');
```

[↑ back to top](#)

Accessors

- Accessor functions for properties are not required.
- If you do make accessor functions use `getVal()` and `setVal('hello')`.

```
// bad
dragon.age();

// good
dragon.getAge();

// bad
dragon.age(25);

// good
dragon.setAge(25);
```

- If the property is a boolean, use `isVal()` or `hasVal()`.

```
// bad
if (!dragon.age()) {
  return false;
}

// good
if (!dragon.hasAge()) {
  return false;
}
```

- It's okay to create get() and set() functions, but be consistent.

```
function Jedi(options) {
  options || (options = {});
  var lightsaber = options.lightsaber || 'blue';
  this.set('lightsaber', lightsaber);
}

Jedi.prototype.set = function set(key, val) {
  this[key] = val;
};

Jedi.prototype.get = function get(key) {
  return this[key];
};
```

[↑ back to top](#)

Constructors

- Assign methods to the prototype object, instead of overwriting the prototype with a new object. Overwriting the prototype makes inheritance impossible: by resetting the prototype you'll overwrite the base!

```
function Jedi() {
  console.log('new jedi');
}

// bad
Jedi.prototype = {
  fight: function fight() {
    console.log('fighting');
  },

  block: function block() {
    console.log('blocking');
  }
};

// good
Jedi.prototype.fight = function fight() {
  console.log('fighting');
};

Jedi.prototype.block = function block() {
  console.log('blocking');
};
```

- Methods can return `this` to help with method chaining.

```
// bad
Jedi.prototype.jump = function jump() {
  this.jumping = true;
  return true;
};

Jedi.prototype.setHeight = function setHeight(height) {
  this.height = height;
};

var luke = new Jedi();
luke.jump(); // => true
luke.setHeight(20); // => undefined

// good
Jedi.prototype.jump = function jump() {
  this.jumping = true;
  return this;
};

Jedi.prototype.setHeight = function setHeight(height) {
  this.height = height;
  return this;
};

var luke = new Jedi();

luke.jump()
.setHeight(20);
```

- It's okay to write a custom `toString()` method, just make sure it works successfully and causes no side effects.

```
function Jedi(options) {
  options || (options = {});
  this.name = options.name || 'no name';
}

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

Jedi.prototype.toString = function toString() {
  return 'Jedi - ' + this.getName();
};
```

[↑ back to top](#)

Events

- When attaching data payloads to events (whether DOM events or something more proprietary like Backbone events), pass a hash instead of a raw value. This allows a subsequent contributor to add more data to the event payload without finding and updating every handler for the event. For example, instead of:

```
// bad
$(this).trigger('listingUpdated', listing.id);

...
$(this).on('listingUpdated', function (e, listingId) {
  // do something with listingId
});
```

prefer:

```
// good
$(this).trigger('listingUpdated', { listingId : listing.id });

...
$(this).on('listingUpdated', function (e, data) {
  // do something with data.listingId
});
```

[↑ back to top](#)

Modules

- The module should start with a `!`. This ensures that if a malformed module forgets to include a final semicolon there aren't errors in production when the scripts get concatenated. [Explanation](#)
- The file should be named with camelCase, live in a folder with the same name, and match the name of the single export.
- Add a method called `noConflict()` that sets the exported module to the previous version and returns this one.
- Always declare `'use strict';` at the top of the module.

```
// fancyInput/fancyInput.js

!function (global) {
  'use strict';

  var previousFancyInput = global.FancyInput;

  function FancyInput(options) {
    this.options = options || {};
  }

  FancyInput.noConflict = function noConflict() {
    global.FancyInput = previousFancyInput;
    return FancyInput;
  };

  global.FancyInput = FancyInput;
}(this);
```

[↑ back to top](#)

jQuery

- Prefix jQuery object variables with a `$`.

```
// bad
var sidebar = $('.sidebar');

// good
var $sidebar = $('.sidebar');
```

- Cache jQuery lookups.

```
// bad
function setSidebar() {
  $('.sidebar').hide();

  // ...stuff...

  $('.sidebar').css({
    'background-color': 'pink'
  });
}

// good
function setSidebar() {
  var $sidebar = $('.sidebar');
  $sidebar.hide();

  // ...stuff...

  $sidebar.css({
    'background-color': 'pink'
  });
}
```

- For DOM queries use Cascading `$('.sidebar ul')` or parent > child `$('.sidebar > ul')`. [jsPerf](#)
- Use `find` with scoped jQuery object queries.

```
// bad
$('.ul', '.sidebar').hide();

// bad
$('.sidebar').find('ul').hide();

// good
$('.sidebar ul').hide();

// good
$('.sidebar > ul').hide();

// good
$sidebar.find('ul').hide();
```

[⬆ back to top](#)

Testing

- **Yup.**

```
function () {  
    return true;  
}
```

[↑ back to top](#)

Front-end Frameworks

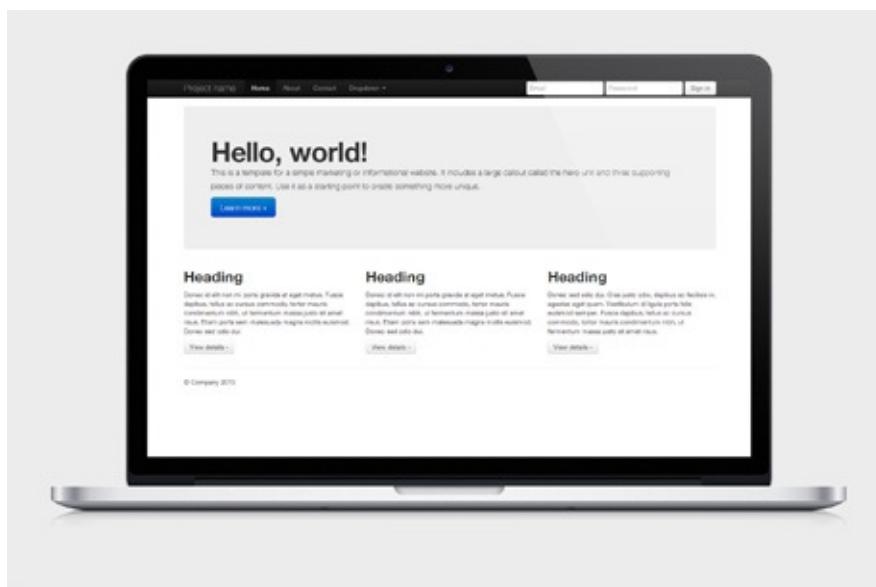
Table of Content

- [What is Front-end Framework?](#)
- [Popular Frameworks](#)
 - [Bootstrap](#)
 - [Foundation](#)
 - [Semantic UI](#)
- [Further Reading](#)

What is a Front-end Framework?

A framework is a standardized set of concepts, practices and criteria for dealing with a common type of problem, which can be used as a reference to help us approach and resolve new problems of a similar nature.

In the world of web design, to give a more straightforward definition, a framework is defined as a package made up of a structure of files and folders of standardized code (HTML, CSS, JS documents etc.) which can be used to support the development of websites, as a basis to start building a site.



Most websites share a very similar (not to say identical) structure. The aim of frameworks is to provide a common structure so that developers don't have to redo it from scratch and can reuse the code provided. In this way, frameworks allow us to cut out much of the work and

save a lot of time.

Front-end Frameworks (or CSS Frameworks)

Frontend frameworks usually consist of a package made up of a structure of files and folders of standardized code (HTML, CSS, JS documents etc.)

The usual components are:

- CSS source code to create a grid: this allows the developer to position the different elements that make up the site design in a simple and versatile fashion.
- Typography style definitions for HTML elements.
- Solutions for cases of browser incompatibility so the site displays correctly in all browsers.
- Creation of standard CSS classes which can be used to style advanced components of the user interface.

Popular Frameworks

Selection of frameworks

Within CSS frameworks, we can draw a distinction between two types of framework according to their complexity: simple frameworks and complete frameworks. This distinction is subjective, and doesn't mean one is better than the others but rather that they give different solutions depending on the level of complexity and/or flexibility required.

- Simple frameworks

These are often called simply “grid systems” but are frameworks nonetheless.

They offer style sheets with column systems to facilitate the distribution of different elements according to the established design. The most recognized is [960 Grid System](#).

- Complete frameworks, *here we mainly talk about this*

They usually offer complete frameworks with configurable features like styled-typography, sets of forms, buttons, icons and other reusable components built to provide navigation, alerts, popovers, and more, images frames, HTML templates, custom settings, etc.

Bootstrap

Bootstrap is the undisputed leader among the available frameworks today. Given its huge popularity, which is still growing every day, you can be sure that this wonderful toolkit won't fail you, or leave you alone on your way to building successful websites.

- Extras/Add-ons: None bundled, but many third-party plug-ins are available.
- Documentation: Good.
- Customization: Basic GUI Customizer. Unfortunately you need to input the color values manually, because there is no color picker available.

Notes on Bootstrap: The main strength of Bootstrap is its huge popularity. Technically, it's not necessarily better than the others in the list, but it offers many more resources (articles and tutorials, third-party plug-ins and extensions, theme builders, and so on) than the other four frameworks combined. In short, Bootstrap is everywhere. And this is the main reason people continue to choose it.

Foundation

Foundation is the second big player in this comparison. With a solid company like ZURB backing it, this framework has a truly strong ... well... foundation. After all, Foundation is used on many big websites including Facebook, Mozilla, Ebay, Yahoo!, and National Geographic, to name a few.

- Extras/Add-ons: Yes .
- Documentation: Good. Many additional resources are available.
- Customization: No GUI customizer, only manual customization.

Notes on Foundation : Foundation is a truly professional framework with business support, training, and consulting offered. It also provides many resources to help you learn and use the framework faster and easier.

Semantic UI

Semantic UI is an ongoing effort to make building websites much more semantic. It utilizes natural language principles, thus making the code much more readable and understandable.

- Extras/Add-ons: No .
- Documentation: Very good. Semantic offers a very well organized documentation, plus a separate website that offers guides for getting started, customizing and creating themes.
- Customization: No GUI customizer, only manual customization.

Notes on Semantic UI : Semantic is the most innovative and full-featured framework among those discussed here. The overall structure of the framework and the naming conventions, in terms of clear logic and semantics of its classes also surpasses the others.

Further Reading

Advantages and Disadvantages to Using a Framework

There are many advantages to using a CSS framework. One of the biggest is probably not having to start every project or website completely from scratch. Having a good base or foundation can save you a lot of time.

Here are some other advantages:

- Easier to maintain code with snippets and libraries
- More organized and easy to setup
- Easier decisions that are already made for you
- Responsive media queries and browser compatibility fixes already included
- Most are free and open source
- Stable and well tested code by hundreds of developers
- Usually the ability to contribute on Github
- Get regular updates, bug fixes, and new feature

Now that all sounds great, but there can also be some possible disadvantages to using a framework:

- Sometimes requires lots of tweaking to make it work, could end up costing you more time
- Requires documentation for changes when updating
- Missing a feature you need, in which you have to introduce another 3rd party asset
- Possibly unnecessary code for your unique situation
- Might make collaboration harder

Chapter II: Web Frameworks

Table of Content

Angular 2

AngularJS is by far the most popular JavaScript framework available today for creating web applications. And now Angular 2 and TypeScript are bringing true object oriented web development to the mainstream, in a syntax that is strikingly close to Java 8.

According to Google engineering director Brad Green, 1.3 million developers use AngularJS and 300 thousand are already using the soon to be released Angular 2. After working with Angular 2 for the last 10 months I believe its impact on the JavaScript community will rival that of the Spring framework on Java.

In this article I'll provide a high-level overview of the Angular 2 framework.

At the end of 2014 Google announced that Angular 2 would be a complete rewrite of AngularJS, and they even created a new language "AtScript" that was meant to be used for writing Angular 2 applications.

But then Microsoft agreed to add support for decorators (a.k.a. annotations) to their TypeScript language (a strict superset of JavaScript), and so that emerged as the language for the development of the Angular 2 framework itself, and the recommended language for developing applications using the AngularJS framework.

You can also develop Angular 2 apps in JavaScript (both ECMAScript 5 and 6) and in Dart.

In addition, the Angular team integrated yet another Microsoft product - the RxJS library of reactive JavaScript extensions, into the Angular 2 framework.

Why Angular 2?

There are many front-end JavaScript frameworks to choose from today, each with its own set of trade-offs. Many people were happy with the functionality that Angular 1.x afforded them. Angular 2 improved on that functionality and made it faster, more scalable and more modern. Organizations that found value in Angular 1.x will find more value in Angular 2.

Angular 2's Advantages

The first release of Angular provided programmers with the tools to develop and architect large scale JavaScript applications, but its age has revealed a number of flaws and sharp edges. Angular 2 was built on five years of community feedback.

Angular 2 Is Easier

The new Angular 2 codebase is more modern, more capable and easier for new programmers to learn than Angular 1.x, while also being easier for project veterans to work with.

With Angular 1, programmers had to understand the differences between Controllers, Services, Factories, Providers and other concepts that could be confusing, especially for new programmers.

Angular 2 is a more streamlined framework that allows programmers to focus on simply building JavaScript classes. Views and controllers are replaced with components, which can be described as a refined version of directives. Even experienced Angular programmers are not always aware of all the capabilities of Angular 1.x directives. Angular 2 components are considerably easier to read, and their API features less jargon than Angular 1.x's directives. Additionally, to help ease the transition to Angular 2, the Angular team has added a `.component` method to Angular 1.5, which has been [back-ported by community member Todd Motto to v1.3](#).

TypeScript

Angular 2 was written in TypeScript, a superset of JavaScript that implements many new ES2016+ features.

By focusing on making the framework easier for computers to process, Angular 2 allows for a much richer development ecosystem. Programmers using sophisticated text editors (or IDEs) will notice dramatic improvements with auto-completion and type suggestions. These improvements help to reduce the cognitive burden of learning Angular 2. Fortunately for traditional ES5 JavaScript programmers this does *not* mean that development must be done in TypeScript or ES2015: programmers can still write vanilla JavaScript that runs without transpilation.

Familiarity

Despite being a complete rewrite, Angular 2 has retained many of its core concepts and conventions with Angular 1.x, e.g. a streamlined, "native JS" implementation of dependency injection. This means that programmers who are already proficient with Angular will have an easier time migrating to Angular 2 than another library like React or framework like Ember.

Performance and Mobile

Angular 2 was designed for mobile from the ground up. Aside from limited processing power, mobile devices have other features and limitations that separate them from traditional computers. Touch interfaces, limited screen real estate and mobile hardware have all been considered in Angular 2.

Desktop computers will also see dramatic improvements in performance and responsiveness.

Angular 2, like React and other modern frameworks, can leverage performance gains by rendering HTML on the server or even in a web worker. Depending on application/site design this isomorphic rendering can make a user's experience *feel* even more instantaneous.

The quest for performance does not end with pre-rendering. Angular 2 makes itself portable to native mobile by integrating with [NativeScript](#), an open source library that bridges JavaScript and mobile. Additionally, the Ionic team is working on an Angular 2 version of their product, providing *another* way to leverage native device features with Angular 2.

Project Architecture and Maintenance

The first iteration of Angular provided web programmers with a highly flexible framework for developing applications. This was a dramatic shift for many web programmers, and while that framework was helpful, it became evident that it was often too flexible. Over time, best practices evolved, and a community-driven structure was endorsed.

Angular 1.x tried to work around various browser limitations related to JavaScript. This was done by introducing a module system that made use of dependency injection. This system was novel, but unfortunately had issues with tooling, notably minification and static analysis.

Angular 2.x makes use of the ES2015 module system, and modern packaging tools like webpack or SystemJS. Modules are far less coupled to the "Angular way", and it's easier to write more generic JavaScript and plug it into Angular. The removal of minification workarounds and the addition of rigid prescriptions make maintaining existing applications simpler. The new module system also makes it easier to develop effective tooling that can reason better about larger projects.

New Features

Some of the other interesting features in Angular 2 are:

- Form Builder
- Change Detection
- Templating
- Routing
- Annotations
- Observables
- Shadow DOM

Differences Between Angular 1 & 2

Note that "Transitional Architecture" refers to a style of Angular 1 application written in a way that mimics Angular 2's component style, but with controllers and directives instead of TypeScript classes.

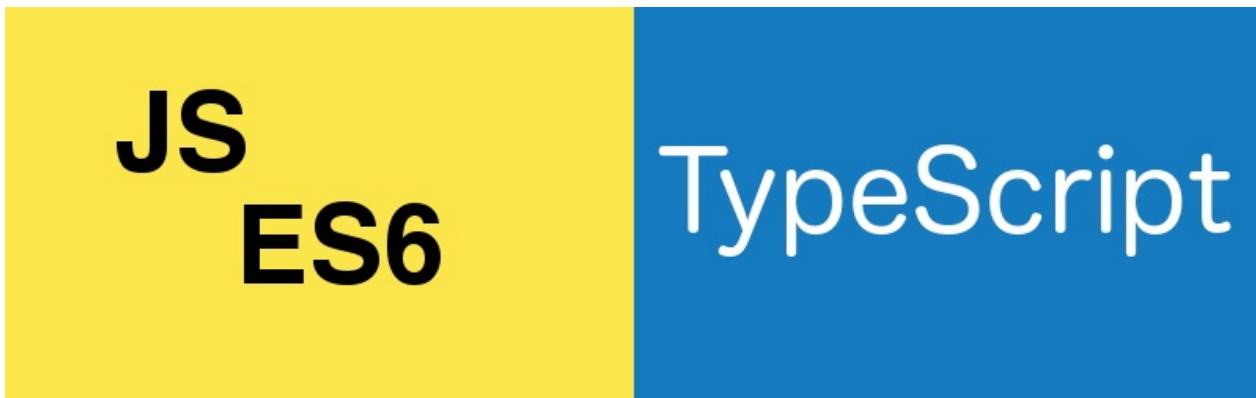
	Old School Angular 1.x	Angular 1.x Best Practices	Transitional Architecture	Angular 2
Nested scopes (<code>\$scope</code> watches)	Used heavily	Avoided	Avoided	Gone
Directives vs controllers	Use as alternatives	Used together	Directives as components	Component directives
Controller and service implementation	Functions	Functions	ES6 classes	ES6 classes
Module system	Angular's modules	Angular's modules	ES6 modules	ES6 modules
Transpiler required	No	No	TypeScript	TypeScript

Prerequisite

Topic

- The Language: Extending JavaScript
 - ECMAScript 6
 - TypeScript
- The Tooling

ECMAScript 6 and TypeScript Features



The language we usually call "JavaScript" is formally known as "ECMAScript". The new version of JavaScript, known as "ES6", offers a number of new features that extend the power of the language.

ES6 is not widely supported in today's browsers, so it needs to be transpiled to ES5. You can choose between several transpilers, but we'll be using TypeScript, which is what the Angular team uses to write Angular 2. Angular 2 makes use of a number of features of ES6 and TypeScript.

ECMAScript 6

Table of Content

- Classes
- Refresher on this
- Arrow Functions
- Template Strings
- Inheritance
- Constants and Block Scoped Variables
- `...spread and ...rest`
- Destructuring
- Modules

JavaScript was created in 1995, but the language is still thriving today. There are subsets, supersets, current versions and the latest version ES6 that brings a lot of new features.

Some of the highlights:

- Classes
- Arrow Functions
- Template Strings
- Inheritance
- Constants and Block Scoped Variables
- Spread and Rest operators
- Destructuring
- Modules

Classes

Classes are a new feature in ES6, used to describe the blueprint of an object and make EcmaScript's prototypical inheritance model function more like a traditional class-based language.

```
class Hamburger {  
  constructor() {  
    // This is the constructor.  
  }  
  listToppings() {  
    // This is a method.  
  }  
}
```

Traditional class-based languages often reserve the word `this` to reference the current (runtime) instance of the class. In Javascript `this` refers to the calling context and therefore can change to be something other than the object.

Object

An object is an instance of a class which is created using the `new` operator. When using a dot notation to access a method on the object, `this` will refer to the object to the left of the dot.

```
let burger = new Hamburger();  
burger.listToppings();
```

In the snippet above, whenever `this` is used from inside class `Hamburger`, it will refer to object `burger`.

Changing Caller Context

JavaScript code can *optionally* supply `this` to a method at call time using one of the following.

```
Function.prototype.call(object [,arg, ...])
Function.prototype.bind(object [,arg, ...])
Function.prototype.apply(object [,argsArray])
```

A Refresher on `this`

Inside a JavaScript class we'll be using `this` keyword to refer to the instance of the class. E.g., consider this case:

```
class Toppings {  
  ...  
  
  formatToppings() { /* implementation details */ }  
  
  list() {  
    return this.formatToppings(this.toppings);  
  }  
}
```

Here `this` refers to an instance of the `Toppings` class. As long as the `list` method is called using dot notation, like `myToppings.list()`, then `this.formatToppings(this.toppings)` invokes the `formatToppings()` method defined on the instance of the class. This will also ensure that inside `formatToppings`, `this` refers to the same instance.

However, `this` can also refer to other things. There are two basic cases that you should remember.

1. Method invocation:

```
someObject.someMethod();
```

Here, `this` used inside `someMethod` will refer to `someObject`, which is usually what you want.

2. Function invocation:

```
someFunction();
```

Here, `this` used inside `someFunction` can refer to different things depending on whether we are in "strict" mode or not. Without using the "strict" mode, `this` refers to the context in which `someFunction()` was called. This rarely what you want, and it can be confusing when `this` is not what you were expecting, because of where the function was called from. In "strict" mode, `this` would be `undefined`, which is slightly less confusing.

[View Example](#)

One of the implications is that you cannot easily detach a method from its object. Consider this example:

```
var log = console.log;
log('Hello');
```

In many browsers this will give you an error. That's because `log` expects `this` to refer to `console`, but the reference was lost when the function was detached from `console`.

This can be fixed by setting `this` explicitly. One way to do this is by using `bind()` method, which allows you to specify the value to use for `this` inside the bound function.

```
var log = console.log.bind(console);
log('Hello');
```

You can also achieve the same using `Function.call` and `Function.apply`, but we won't discuss this here.

Another instance where `this` can be confusing is with respect to anonymous functions, or functions declared within other functions. Consider the following:

```
class ServerRequest {
  notify() {
    ...
  }
  fetch() {
    getFromServer(function callback(err, data) {
      this.notify(); // this is not going to work
    });
  }
}
```

In the above case `this` will *not* point to the expected object: in "strict" mode it will be `undefined`. This leads to another ES6 feature - arrow functions, which will be covered next.

Arrow Functions

ES6 offers some new syntax for dealing with `this`: "arrow functions". Arrow functions also make higher order functions much easier to work with.

The new "fat arrow" notation can be used to define anonymous functions in a simpler way.

Consider the following example:

```
items.forEach(function(x) {
  console.log(x);
  incrementedItems.push(x+1);
});
```

This can be rewritten as an "arrow function" using the following syntax:

```
items.forEach((x) => {
  console.log(x);
  incrementedItems.push(x+1);
});
```

Functions that calculate a single expression and return its values can be defined even simpler:

```
incrementedItems = items.map((x) => x+1);
```

The latter is *almost* equivalent to the following:

```
incrementedItems = items.map(function (x) {
  return x+1;
});
```

There is one important difference, however: arrow functions do not set a local copy of `this`, `arguments`, `super`, or `new.target`. When `this` is used inside an arrow function JavaScript uses the `this` from the outer scope. Consider the following example:

```

class Toppings {
  constructor(toppings) {
    this.toppings = Array.isArray(toppings) ? toppings : [];
  }
  outputList() {
    this.toppings.forEach(function(topping, i) {
      console.log(topping, i + '/' + this.toppings.length); // no this
    })
  }
}

var ctrl = new Toppings(['cheese', 'lettuce']);

ctrl.outputList();

```

Let's try this code on ES6 Fiddle (<http://www.es6fiddle.net/>). As we see, this gives us an error, since `this` is undefined inside the anonymous function.

Now, let's change the method to use the arrow function:

```

class Toppings {
  constructor(toppings) {
    this.toppings = Array.isArray(toppings) ? toppings : [];
  }
  outputList() {
    this.toppings
      .forEach((topping, i) => console
        .log(topping, i + '/' + this.toppings.length)) // `this` works!
  }
}

var ctrl = new Toppings(['cheese', 'lettuce']);

```

Here `this` inside the arrow function refers to the instance variable.

Warning arrow functions do *not* have their own `arguments` variable, which can be confusing to veteran JavaScript programmers. `super` and `new.target` are also scoped from the outer enclosure.

Template Strings

In traditional JavaScript, text that is enclosed within matching " or ' marks is considered a string. Text within double or single quotes can only be on one line. There was no way to insert data into these strings. This resulted in a lot of ugly concatenation code that looked like:

```
var name = 'Sam';
var age = 42;

console.log('hello my name is ' + name + ' I am ' + age + ' years old');
```

ES6 introduces a new type of string literal that is marked with back ticks (`). These string literals *can* include newlines, and there is a string interpolation for inserting variables into strings:

```
var name = 'Sam';
var age = 42;

console.log(`hello my name is ${name}, and I am ${age} years old`);
```

There are all sorts of places where this kind of string can come in handy, and front-end web development is one of them.

Inheritance

JavaScript's inheritance works differently from inheritance in other languages, which can be very confusing. ES6 classes provide a syntactic sugar attempting to alleviate the issues with using prototypical inheritance present in ES5.

To illustrate this, let's image we have a zoo application where types of birds are created. In classical inheritance, we define a base class and then subclass it to create a derived class.

Subclassing

The example code below shows how to derive `Penguin` from `Bird` using the **extends** keyword. Also pay attention to the **super** keyword used in the subclass constructor of `Penguin`, it is used to pass the argument to the base class `Bird`'s constructor.

The `Bird` class defines the method `walk` which is inherited by the `Penguin` class and is available for use by instance of `Penguin` objects. Likewise the `Penguin` class defines the method `swim` which is not available to `Bird` objects. Inheritance works top-down from base class to its subclass.

Object Initialization

The class constructor is called when an object is created using the **new** operator, it will be called before the object is fully created. A consturctor is used to pass in arguments to initialize the newly created object.

The order of object creation starts from its base class and then moves down to any subclass(es).

```
// Base Class : ES6
class Bird {
  constructor(weight, height) {
    this.weight = weight;
    this.height = height;
  }

  walk() {
    console.log('walk!');
  }
}

// Subclass
class Penguin extends Bird {
  constructor(weight, height) {
    super(weight, height);
  }

  swim() {
    console.log('swim!');
  }
}

// Penguin object
let penguin = new Penguin(...);
penguin.walk(); //walk!
penguin.swim(); //swim!
```

Below we show how prototypal inheritance was done before class was introduced to JavaScript.

```
// JavaScript classical inheritance.

// Bird constructor
function Bird(weight, height) {
  this.weight = weight;
  this.height = height;
}

// Add method to Bird prototype.
Bird.prototype.walk = function() {
  console.log("walk!");
};

// Penguin constructor.
function Penguin(weight, height) {
  Bird.call(this, weight, height);
}

// Prototypal inheritance (Penguin is-a Bird).
Penguin.prototype = Object.create( Bird.prototype );
Penguin.prototype.constructor = Penguin;

// Add method to Penguin prototype.
Penguin.prototype.swim = function() {
  console.log("swim!");
};

// Create a Penguin object.
let penguin = new Penguin(50,10);

// Calls method on Bird, since it's not defined by Penguin.
penguin.walk(); // walk!

// Calls method on Penguin.
penguin.swim(); // swim!
```

Constants and Block Scoped Variables

ES6 introduces the concept of block scoping. Block scoping will be familiar to programmers from other languages like C, Java, or even PHP. In ES5 JavaScript and earlier, `var`s are scoped to `function`s, and they can "see" outside their functions to the outer context.

```
var five = 5;
var threeAlso = three; // error

function scope1() {
  var three = 3;
  var fiveAlso = five; // == 5
  var sevenAlso = seven; // error
}

function scope2() {
  var seven = 7;
  var fiveAlso = five; // == 5
  var threeAlso = three; // error
}
```

In ES5 functions were essentially containers that could be "seen" out of, but not into.

In ES6 `var` still works that way, using functions as containers, but there are two new ways to declare variables: `const` and `let`.

`const` and `let` use `{` and `}` blocks as containers, hence "block scope". Block scoping is most useful during loops. Consider the following:

```
var i;
for (i = 0; i < 10; i += 1) {
  var j = i;
  let k = i;
}
console.log(j); // 9
console.log(k); // undefined
```

Despite the introduction of block scoping, functions are still the preferred mechanism for dealing with most loops.

`let` works like `var` in the sense that its data is read/write. `let` is also useful when used in a for loop. For example, without `let`, the following example would output `5,5,5,5,5`:

```
for(var x=0; x<5; x++) {
    setTimeout(()=>console.log(x), 0)
}
```

However, when using `let` instead of `var`, the value would be scoped in a way that people would expect.

```
for(let x=0; x<5; x++) {
    setTimeout(()=>console.log(x), 0)
}
```

Alternatively, `const` is read-only. Once `const` has been assigned, the identifier cannot be reassigned.

For example:

```
const myName = 'pat';
let yourName = 'jo';

yourName = 'sam'; // assigns
myName = 'jan'; // error
```

The read-only nature can be demonstrated with any object:

```
const literal = {};

literal.attribute = 'test'; // fine
literal = []; // error;
```

However there are two cases where `const` does not work as you think it should.

1. A const object literal.
2. A const reference to an object.

Const Object Literal

```
const person = {
    name: 'Tammy'
};

person.name = 'Pushpa'; // OK, name property changed.

person = null; // "TypeError: Assignment to constant variable."
```

The example above demonstrates that we are able to change the **name** property of object **person**, but we are unable to reset the reference **person** since it has been marked as `const`.

Const Reference To An Object

Something similar to the above code is using a `const` reference, below we've switch to using **let** for the literal object.

```
let person = {  
    name: 'Tammy'  
};  
  
const p = person;  
  
p.name = 'Pushpa'; // OK, name property changed.  
  
p = null;          // "TypeError: Assignment to constant variable."
```

Take away, marking an object reference **const** does not make properties inside the object **const**.

[Ref:](#)

...spread and ...rest

A Spread operator allows in-place expansion of an expression for the following cases:

1. Array
2. Function call
3. Multiple variable destructuring

The Rest operator works in the opposite direction of the spread operator, it collects an indefinite number of comma separated expressions into an array.

Operator Spread

Spread example:

```
const add = (a, b) => a + b;
let args = [3, 5];
add(...args); // same as `add(args[0], args[1])`, or `add.apply(null, args)`
```

Functions aren't the only place in JavaScript that makes use of comma separated lists - arrays can now be concatenated with ease:

```
let cde = ['c', 'd', 'e'];
let scale = ['a', 'b', ...cde, 'f', 'g']; // ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

Similarly, object literals can do the same thing:

```
let mapABC = { a: 5, b: 6, c: 3};
let mapABCD = { ...mapABC, d: 7}; // { a: 5, b: 6, c: 3, d: 7 }
```

Operator Rest

Rest arguments share the ellipsis like syntax of rest operators but are used for a different purpose. Rest arguments are used to access a variable number of arguments passed to a function. For example:

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

function add(...numbers) {
    return numbers[0] + numbers[1];
}

addSimple(3, 2); // 5
add(3, 2); // 5

// or in es6 style:
const addEs6 = (...numbers) => numbers.reduce((p, c) => p + c, 0);

addEs6(1, 2, 3); // 6
```

Technically JavaScript already had an `arguments` variable set on each function (except for arrow functions), however `arguments` has a lot of issues, one of which is the fact that it is not technically an array.

Rest arguments are in fact arrays. The other important difference is that rest arguments only include arguments not specifically named in a function like so:

```
function print(a, b, c, ...more) {
    console.log(more[0]);
    console.log(arguments[0]);
}

print(1, 2, 3, 4, 5);
// 4
// 1
```

Destructuring

Destructuring is a way to quickly extract data out of an `{}` or `[]` without having to write much code.

Destructuring can be used to turn the following:

```
let foo = ['one', 'two', 'three'];

let one  = foo[0];
let two  = foo[1];
let three = foo[2];
```

into

```
let foo = ['one', 'two', 'three'];
let [one, two, three] = foo;
console.log(one); // 'one'
```

This is pretty interesting, but at first it might be hard to see the use case. ES6 also supports object destructuring, which might make uses more obvious:

```
let myModule = {
  drawSquare: function drawSquare(length) { /* implementation */ },
  drawCircle: function drawCircle(radius) { /* implementation */ },
  drawText: function drawText(text) { /* implementation */ },
};

let {drawSquare, drawText} = myModule;

drawSquare(5);
drawText('hello');
```

Destructuring can also be used for passing objects into a function, allowing you to pull specific properties out of an object in a concise manner. It is also possible to assign default values to destructured arguments, which can be a useful pattern if passing in a configuration object.

```
let jane = { firstName: 'Jane', lastName: 'Doe' };
let john = { firstName: 'John', lastName: 'Doe', middleName: 'Smith' }
function sayName({firstName, lastName, middleName = 'N/A'}) {
  console.log(`Hello ${firstName} ${middleName} ${lastName}`)
}

sayName(jane) // -> Hello Jane N/A Doe
sayName(john) // -> Hello John Smith Doe
```

There are many more sophisticated things that can be done with destructuring, and the [MDN](#) has some great examples, including nested object destructuring and dynamic destructuring with `for ... in` operators".

ES6 Modules

ES6 introduced *module* support. A module in ES6 is single file that allows code and data to be isolated, it helps in organizing and grouping code logically. In other languages it's called a package or library.

All code and data inside the module has file scope, what this means is they are not accessible from code outside the module. To share code or data outside a module, it needs to be exported using the **export** keyword.

```
// File: circle.js

export const pi = 3.141592;

export const circumference = diameter => diameter * pi;
```

The code above uses the *Arrow* function for `circumference`, which was introduced in ES6, and is a shortform for the following.

```
export function circumference(diameter) {
    return diameter * pi;
}
```

Module Systems

Using a module on the backend(server side) is relatively straightforward, you simply make use of the **import** keyword. However Web Browsers have no concept of modules or import, they just know how to load javascript code. We need a way to bring in a javascript module to start using it from other javascript code. This is where a module loader comes in.

We won't get into the various module systems out there, but it's worth understanding there are various module loaders available. The popular choices out there are:

- RequireJS
- SystemJS
- Webpack

Loading a Module From a Browser

Below we make use of SystemJS to load a module. The script first loads the code for the SystemJS library, then the function call **System.import** is used to import(load) the *app* module.

Loading ES6 modules is a little trickier. In an ES6-compliant browser you use the `System` keyword to load modules asynchronously. To make our code work with current browsers, however, we will use the SystemJS library as a polyfill:

```
<script src="/node_module/systemjs/dist/system.js"></script>
<script>
  var promise = System.import('app')
    .then(function() {
      console.log('Loaded!');
    })
    .then(null, function(error) {
      console.error('Failed to load:', error);
    });
</script>
```

TypeScript

Table of Content

- [Getting Started](#)
- [Working With tsc](#)
- [Typings](#)
- [Linting](#)
- [TypeScript Features](#)
- [TypeScript Classes](#)
- [Interfaces](#)
- [Shapes](#)
- [Type Inference](#)
- [Decorators](#)
 - [Property Decorators](#)
 - [Class Decorators](#)
 - [Parameter Decorators](#)

ES6 is the current version of JavaScript. TypeScript is a superset of ES6, which means all ES6 features are part of TypeScript, but not all TypeScript features are part of ES6. Consequently, TypeScript must be transpiled into ES5 to run in most browsers.

One of TypeScript's primary features is the addition of type information, hence the name. This type information can help make JavaScript programs more predictable and easier to reason about.

Types let developers write more explicit "contracts". In other words, things like function signatures are more explicit.

Without TS:

```
function add(a, b) {  
    return a + b;  
}  
  
add(1, 3);    // 4  
add(1, '3'); // '13'
```

With TS:

```
function add(a: number, b: number) {  
    return a + b;  
}  
  
add(1, 3); // 4  
// compiler error before JS is even produced  
add(1, '3'); // '13'
```

Getting Started With TypeScript

Install the TypeScript transpiler using npm:

```
$ npm install -g typescript
```

Then use `tsc` to manually compile a TypeScript source file into ES5:

```
$ tsc test.ts
$ node test.js
```

Note About ES6 Examples

Our earlier ES6 class won't compile now. TypeScript is more demanding than ES6 and it expects instance properties to be declared:

```
class Pizza {
  toppings: string[];
  constructor(toppings: string[]) {
    this.toppings = toppings;
  }
}
```

Note that now that we've declared `toppings` to be an array of strings, TypeScript will enforce this. If we try to assign a number to it, we will get an error at compilation time.

If you want to have a property that can be set to a value of any type, however, you can still do this: just declare its type to be "any":

```
class Pizza {
  toppings: any;
  //...
}
```

Working With tsc

So far `tsc` has been used to compile a single file. Typically programmers have a lot more than one file to compile. Thankfully `tsc` can handle multiple files as arguments.

Imagine two ultra simple files/modules:

`a.ts`

```
export const A = (a) => console.log(a);
```

`b.ts`

```
export const B = (b) => console.log(b);
```

Before TypeScript@1.8.2:

```
$ tsc ./a.ts ./b.ts
a.ts(1,1): error TS1148: Cannot compile modules unless the '--module' flag is provided
.
```

Hmmm. What's the deal with this module flag? TypeScript has a help menu, let's take a look:

```
$ tsc --help | grep module
-m KIND, --module KIND           Specify module code generation: 'commonjs', 'amd',
'system', 'umd' or 'es2015'
--moduleResolution               Specifies module resolution strategy: 'node' (Node
.js) or 'classic' (TypeScript pre-1.6).
```

(TypeScript has more help than what we've shown; we filtered by `grep` for brevity.) There are two help entries that reference "module", and `--module` is the one TypeScript was complaining about. The description explains that TypeScript supports a number of different module schemes. For the moment `commonjs` is desirable. This will produce modules that are compatible with node.js's module system.

```
$ tsc -m commonjs ./a.ts ./b.ts
```

Since TypeScript@1.8.2, `tsc` has a default rule for `--module` option: `target === 'ES6' ? 'ES6' : 'commonjs'` (more details can be found [here](#)), so we can simply run:

```
$ tsc ./a.ts ./b.ts
```

`tsc` should produce no output. In many command line traditions, no output is actually a mark of success. Listing the directory contents will confirm that our TypeScript files did in fact compile.

```
$ ls  
a.js    a.ts    b.js    b.ts
```

Excellent - there are now two JavaScript modules ready for consumption.

Telling the `tsc` command what to compile becomes tedious and labor intensive even on small projects. Fortunately TypeScript has a means of simplifying this. `tsconfig.json` files let programmers write down all the compiler settings they want. When `tsc` is run, it looks for `tsconfig.json` files and uses their rules to compile JavaScript.

For Angular 2 projects there are a number of specific settings that need to be configured in a project's `tsconfig.json`

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "noImplicitAny": false,
    "removeComments": false,
    "sourceMap": true
  },
  "exclude": [
    "node_modules",
    "dist/"
  ]
}
```

Target

The compilation target. TypeScript supports targeting different platforms depending on your needs. In our case, we're targeting modern browsers which support ES5.

Module

The target module resolution interface. We're integrating TypeScript through webpack which supports different interfaces. We've decided to use node's module resolution interface, `commonjs`.

Decorators

Decorator support in TypeScript [hasn't been finalized yet](#) but since Angular 2 uses decorators extensively, these need to be set to true. Decorators have not been introduced yet, and will be covered later in this section.

TypeScript with Webpack

We won't be running `tsc` manually, however. Instead, webpack's `ts-loader` will do the transpilation during the build:

```
// webpack.config.js
//...
rules: [
  { test: /\.ts$/, loader: 'ts', exclude: /node_modules/ },
  //...
]
```

This loader calls `tsc` for us, and it will use our `tsconfig.json`.

Typings

Astute readers might be wondering what happens when TypeScript programmers need to interface with JavaScript modules that have no type information. TypeScript recognizes files labelled `*.d.ts` as *definition* files. These files are meant to use TypeScript to describe interfaces presented by JavaScript libraries.

There are communities of people dedicated to creating typings for JavaScript projects. There is also a utility called `typings` (`npm install --save-dev typings`) that can be used to manage third party typings from a variety of sources. (*Deprecated in TypeScript 2.0*)

In TypeScript 2.0, users can get type files directly from `@types` through `npm` (for example, `npm install --save @types/lodash` will install `lodash` type file).

Linting

Many editors support the concept of "linting" - a grammar check for computer programs.

Linting can be done in a programmer's editor and/or through automation.

For TypeScript there is a package called `tslint`, (`npm install --save-dev tslint`) which can be plugged into many editors. `tslint` can also be configured with a `tslint.json` file.

Webpack can run `tslint` before it attempts to run `tsc`. This is done by installing `tslint-loader` (`npm install --save-dev tslint-loader`) which plugs into webpack like so:

```
// ...
module: {
  preLoaders: [
    { test: /\.ts$/, loader: 'tslint' }
  ],
  loaders: [
    { test: /\.ts$/, loader: 'ts', exclude: /node_modules/ },
    // ...
  ]
}
// ...
```

TypeScript Features

Now that producing JavaScript from TypeScript code has been de-mystified, some of its features can be described and experimented with.

- Types
- Interfaces
- Shapes
- Decorators

Types

Many people do not realize it, but JavaScript *does* in fact have types, they're just "duck typed", which roughly means that the programmer does not have to think about them.

JavaScript's types also exist in TypeScript:

- `boolean` (true/false)
- `number` integers, floats, `Infinity` and `Nan`
- `string` characters and strings of characters
- `[]` Arrays of other types, like `number[]` or `boolean[]`
- `{}` Object literal
- `undefined` not set

TypeScript also adds

- `enum` enumerations like `{ Red, Blue, Green }`
- `any` use any type
- `void` nothing

Primitive type example:

```
let isDone: boolean = false;
let height: number = 6;
let name: string = "bob";
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
enum Color {Red, Green, Blue};
let c: Color = Color.Green;
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean

function showMessage(data: string): void {
    alert(data);
}
showMessage('hello');
```

This illustrates the primitive types in TypeScript, and ends by illustrating a `showMessage` function. In this function the parameters have specific types that are checked when `tsc` is run.

In many JavaScript functions it's quite common for functions to take optional parameters. TypeScript provides support for this, like so:

```
function logMessage(message: string, isDebug?: boolean) {
    if (isDebug) {
        console.log('Debug: ' + message);
    } else {
        console.log(message);
    }
}
logMessage('hi');           // 'hi'
logMessage('test', true); // 'Debug: test'
```

Using a `?` lets `tsc` know that `isDebug` is an optional parameter. `tsc` will not complain if `isDebug` is omitted.

TypeScript Classes

TypeScript also treats `class` es as their own type:

```
class Foo { foo: number; }
class Bar { bar: string; }

class Baz {
    constructor(foo: Foo, bar: Bar) { }
}

let baz = new Baz(new Foo(), new Bar()); // valid
baz = new Baz(new Bar(), new Foo());      // tsc errors
```

Like function parameters, `class` es sometimes have optional members. The same `?:` syntax can be used on a `class` definition:

```
class Person {
    name: string;
    nickName?: string;
}
```

In the above example, an instance of `Person` is guaranteed to have a `name`, and might optionally have a `nickName`.

Interfaces

An *interface* is a TypeScript artifact, it is not part of ECMAScript. A *interface* is a way to define a *contract* on a function with respect to the arguments and their type. Along with functions, an *interface* can also be used with a Class as well to define custom types.

An interface is an abstract type, it does not contain any code as a *class* does. It only defines the 'signature' or shape of an API. During transpilation, an `interface` will not generate any code, it is only used by Typescript for type checking during development.

Here is an example of an interface describing a function API:

```
interface Callback {
  (error: Error, data: any): void;
}

function callServer(callback: Callback) {
  callback(null, 'hi');
}

callServer((error, data) => console.log(data)); // 'hi'
callServer('hi'); // tsc error
```

Sometimes JavaScript functions can accept multiple types as well as varying arguments, that is, they can have different call signatures. Interfaces can be used to specify this.

```
interface PrintOutput {
  (message: string): void; // common case
  (message: string[]): void; // less common case
}

let printOut: PrintOutput = (message) => {
  if (Array.isArray(message)) {
    console.log(message.join(', '));
  } else {
    console.log(message);
  }
}

printOut('hello'); // 'hello'
printOut(['hi', 'bye']); // 'hi, bye'
```

Here is an example of an interface describing an object literal:

```
interface Action {  
    type: string;  
}  
  
let a: Action = {  
    type: 'literal'  
}
```

Shapes

Underneath TypeScript is JavaScript, and underneath JavaScript is typically a JIT (Just-In-Time compiler). Given JavaScript's underlying semantics, types are typically reasoned about by "shapes". These underlying shapes work like TypeScript's interfaces, and are in fact how TypeScript compares custom types like `class` es and `interface` s.

Consider an expansion of the previous example:

```
interface Action {
  type: string;
}

let a: Action = {
  type: 'literal'
}

class NotAnAction {
  type: string;
  constructor() {
    this.type = 'Constructor function (class)';
  }
}

a = new NotAnAction(); // valid TypeScript!
```

Despite the fact that `Action` and `NotAnAction` have different identifiers, `tsc` lets us assign an instance of `NotAnAction` to `a` which has a type of `Action`. This is because TypeScript only really cares that objects have the same shape. In other words if two objects have the same attributes, with the same typings, those two objects are considered to be of the same type.

Type Inference

One common misconception about TypeScript's types is that code needs to explicitly describe types at every possible opportunity. Fortunately this is not the case. TypeScript has a rich type inference system that will "fill in the blanks" for the programmer. Consider the following:

type-inference-finds-error.ts

```
let numbers = [2, 3, 5, 7, 11];
numbers = ['this will generate a type error'];
```

```
tsc ./type-inference-finds-error.ts
type-inference-finds-error.ts(2,1): error TS2322: Type 'string[]' is not assignable to
type 'number[]'.
  Type 'string' is not assignable to type 'number'.
```

The code contains no extra type information. In fact, it's valid ES6. If `var` had been used, it would be valid ES5. Yet TypeScript is still able to determine type information.

Type inference can also work through context, which is handy with callbacks. Consider the following:

type-inference-finds-error-2.ts

```
interface FakeEvent {
  type: string;
}

interface FakeEventHandler {
  (e: FakeEvent): void;
}

class FakeWindow {
  onMouseDown: FakeEventHandler
}
const fakeWindow = new FakeWindow();

fakeWindow.onMouseDown = (a: number) => {
  // this will fail
};
```

```
tsc ./type-inference-finds-error-2.ts
type-inference-finds-error-2.ts(14,1): error TS2322: Type '(a: number) => void' is not
assignable to type 'FakeEventHandler'.
  Types of parameters 'a' and 'e' are incompatible.
    Type 'number' is not assignable to type 'FakeEvent'.
      Property 'type' is missing in type 'Number'.
```

In this example the context is not obvious since the interfaces have been defined explicitly. In a browser environment with a real `window` object, this would be a handy feature, especially the type completion of the `Event` object.

Decorators

Decorators are proposed for a future version of JavaScript, but the Angular 2 team *really* wanted to use them, and they have been included in TypeScript.

Decorators are functions that are invoked with a prefixed `@` symbol, and *immediately* followed by a `class`, parameter, method or property. The decorator function is supplied information about the `class`, parameter or method, and the decorator function returns something in its place, or manipulates its target in some way. Typically the "something" a decorator returns is the same thing that was passed in, but it has been augmented in some way.

Decorators are quite new in TypeScript, and most use cases demonstrate the use of existing decorators. However, decorators are just functions, and are easier to reason about after walking through a few examples. Decorators are functions, and there are four things (`class`, parameter, method and property) that can be decorated; consequently there are four different function signatures for decorators:

- `class`: `declare type ClassDecorator = <TFunction extends Function>(target: TFunction) => TFunction | void;`
- `property`: `declare type PropertyDecorator = (target: Object, propertyKey: string | symbol) => void;`
- `method`: `declare type MethodDecorator = <T>(target: Object, propertyKey: string | symbol, descriptor: TypedPropertyDescriptor<T>) => TypedPropertyDescriptor<T> | void;`
- `parameter`: `declare type ParameterDecorator = (target: Object, propertyKey: string | symbol, parameterIndex: number) => void;`

Readers who have played with Angular 2 will notice that these signatures do not look like the signatures used by Angular 2 specific decorators like `@Component()`.

Notice the `()` on `@Component`. This means that the `@Component` is called once JavaScript encounters `@Component()`. In turn, this means that there must be a `Component` function somewhere that returns a function matching one of the decorator signatures outlined above. This is an example of the decorator factory pattern.

If decorators still look confusing, perhaps some examples will clear things up.

Property Decorators

Property decorators work with properties of classes.

```
function Override(label: string) {
  return function (target: any, key: string) {
    Object.defineProperty(target, key, {
      configurable: false,
      get: () => label
    });
  }
}

class Test {
  @Override('test') // invokes Override, which returns the decorator
  name: string = 'pat';
}

let t = new Test();
console.log(t.name); // 'test'
```

The above example must be compiled with both the `--experimentalDecorators` and `--emitDecoratorMetadata` flags.

In this case the decorated property is replaced by the `label` passed to the decorator. It's important to note that property values cannot be directly manipulated by the decorator; instead an accessor is used.

Here's a classic property example that uses a *plain decorator*

```
function ReadOnly(target: any, key: string) {
  Object.defineProperty(target, key, { writable: false });
}

class Test {
  @ReadOnly // notice there are no `()`` 
  name: string;
}

const t = new Test();
t.name = 'jan';
console.log(t.name); // 'undefined'
```

In this case the `name` property is not `writable`, and remains undefined.

Class Decorators

```

function log(prefix?: string) {
  return (target) => {
    // save a reference to the original constructor
    var original = target;

    // a utility function to generate instances of a class
    function construct(constructor, args) {
      var c: any = function () {
        return constructor.apply(this, args);
      }
      c.prototype = constructor.prototype;
      return new c();
    }

    // the new constructor behavior
    var f: any = function (...args) {
      console.log(prefix + original.name);
      return construct(original, args);
    }

    // copy prototype so instanceof operator still works
    f.prototype = original.prototype;

    // return new constructor (will override original)
    return f;
  };
}

@log('hello')
class World {}

const w = new World(); // outputs "helloWorld"

```

In the example `log` is invoked using `@`, and passed a string as a parameter, `@log()` returns an anonymous function that is the actual decorator.

The decorator function takes a `class`, or constructor function (ES5) as an argument. The decorator function then returns a new class construction function that is used whenever `World` is instantiated.

This decorator does nothing other than log out its given parameter, and its `target`'s class name to the console.

Parameter Decorators

```
function logPosition(target: any, propertyKey: string, parameterIndex: number) {
  console.log(parameterIndex);
}

class Cow {
  say(b: string, @logPosition c: boolean) {
    console.log(b);
  }
}

new Cow().say('hello', false); // outputs 1 (newline) hello
```

The above demonstrates decorating method parameters. Readers familiar with Angular 2 can now imagine how Angular 2 implemented their `@Inject()` system.

The JavaScript Toolchain

Table of Content

- Source Control: git
 - The Command Line
 - Command Line JavaScript: NodeJS
 - Back-End Code Sharing and Distribution: npm
 - Module Loading, Bundling and Build Tasks
 - Chrome

The JavaScript Toolchain

In this section, we'll describe the tools that you'll be using for the rest of the course.



Source Control: [Git](#)

A source control, sometimes called a version control brings change management to saving files at different points in the development process. A **Version control system (VCS)** that will we make use of is *Git*.

Git is a decentralized distributed versioning system, it allows programmers to collaborate on the same codebase without stepping on each other's toes. It has become the de-facto source control system for open source development because of its decentralized model and cheap branching features.

For more information on how to use Git, head over to [Pro Git](#)

The Command Line

JavaScript development tools are very command line oriented. If you come from a Windows background you may find this unfamiliar. However the command line provides better support for automating development tasks, so it's worth getting comfortable with it.

We will provide examples for all command line activities required by this course.

Command Line JavaScript: NodeJS

Node.js is a JavaScript runtime environment that allows JavaScript code to run outside of a browser using Google V8 JavaScript engine. Node.js is used for writing fast executing code on the server to handle events and non-blocking I/O efficiently.

- REPL (Read-Eval-Print-Loop) to quickly write and test JavaScript code.
- The V8 JavaScript interpreter.
- Modules for doing OS tasks like file I/O, HTTP, etc.

While Node.js was initially intended for writing server code in JavaScript, today it is widely used by JavaScript tools, which makes it relevant to front-end programmers too. A lot of the tools you'll be using in this course leverage Node.js.

Back-End Code Sharing and Distribution: **npm**

`npm` is the "node package manager". It installs with NodeJS, and gives you access to a wide variety of 3rd-party JavaScript modules.

It also performs dependency management for your back-end application. You specify module dependencies in a file called `package.json`; running `npm install` will resolve, download and install your back-end application's dependencies.

Module Loading, Bundling and Build Tasks: [Webpack](#)

Webpack is a JavaScript module bundler. It takes modules with their dependencies and generates static assets representing those modules. Webpack known only how to bundle JavaScript. To bundle other assets like CSS, HTML, images or just about anything it uses additional loaders. Webpack can also be extended via plugins, for example minification and mangling can be done using the UglifyJS plugin for webpack.

Bootstrapping an Angular 2 Application

Bootstrapping is an essential process in Angular - it is where the application is loaded when Angular comes to life.

Bootstrapping Angular 2 applications is certainly different from Angular 1.x, but is still a straightforward procedure. Let's take a look at how this is done.

Understanding the File Structure

To get started let's create a bare-bones Angular 2 application with a single component. To do this we need the following files:

- *app/app.component.ts* - this is where we define our root component
- *app/app.module.ts* - the entry Angular Module to be bootstrapped
- *index.html* - this is the page the component will be rendered in
- *app/index.ts* - is the glue that combines the component and page together

app/app.component.ts

```
import {Component} from '@angular/core'

@Component({
  selector: 'app',
  template: '<b>Bootstrapping an Angular 2 Application</b>'
})

export class MyApp {}
```

index.html

```
...
<body>
  <app>Loading...</app>
</body>
...
```

app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } '@angular/core';
import { MyApp } from './app.component'

@NgModule({
  imports: [BrowserModule],
  declarations: [MyApp],
  bootstrap: [MyApp]
})
export class AppModule { }
```

app/main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(MyAppModule);
```

If you're making use of Ahead-of-Time (AoT) compilation, you would code `main.ts` as follows.

```
import { platformBrowser } from '@angular/platform-browser';
import { AppModuleNgFactory } from '../aot/app/app.module.ngfactory';

platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);
```

[View Example](#)

The bootstrap process loads `main.ts` which is the main entry point of the application. The `MyAppModule` operates as the root module of our application. The module is configured to use `MyApp` as the component to bootstrap, and will be rendered on any `app` HTML element encountered.

There is an `app` HTML element in the `index.html` file, and we use `app/index.ts` to import the `MyAppModule` component and the `platformBrowserDynamic().bootstrapModule` function and kickstart the process. As shown above, you may optionally use **AoT** in which case you will be working with Factories, in the example, `MyAppModuleNgFactory` and `bootstrapModuleFactory`.

Why does Angular 2 bootstrap itself in this way? Well there is actually a very good reason. Since Angular 2 is not a web-only based framework, we can write components that will run in NativeScript, or Cordova, or any other environment that can host Angular 2 applications.

The magic is then in our bootstrapping process - we can import which platform we would like to use, depending on the environment we're operating under. In our example, since we were running our Angular 2 application in the browser, we used the bootstrapping process found in `@angular/platform-browser-dynamic`.

It's also a good idea to leave the bootstrapping process in its own separate `index.ts` file. This makes it easier to test (since the components are isolated from the `bootstrap` call), easier to reuse and gives better organization and structure to our application.

There is more to understanding Angular Modules and `@NgModule` which will be covered later, but for now this is enough to get started.

Bootstrapping Providers

The bootstrap process also starts the dependency injection system in Angular 2. We won't go over Angular 2's dependency injection system here - that is covered later. Instead let's take a look at an example of how to bootstrap your application with application-wide providers.

For this, we will register a service called `Greeter` with the `providers` property of the module we are using to bootstrap the application.

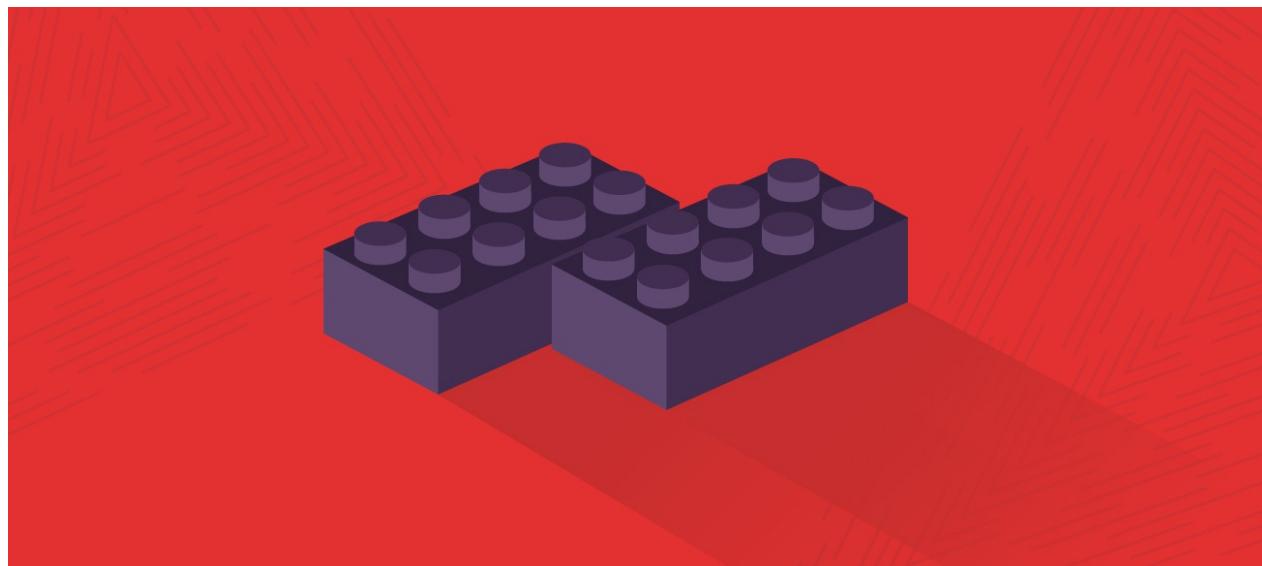
app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } '@angular/core';
import { MyApp } from './app.component'
import { Greeter } from './greeter.service';

@NgModule({
  imports: [BrowserModule],
  providers: [Greeter],
  declarations: [MyApp],
  bootstrap: [MyApp]
})
export class AppModule {
```

[View Example](#)

Components in Angular 2



The core concept of any Angular 2 application is the *component*. In effect, the whole application can be modeled as a tree of these components.

This is how the Angular 2 team defines a component:

A component controls a patch of screen real estate that we could call a view, and declares reusable UI building blocks for an application.

Basically, a component is anything that is visible to the end user and which can be reused many times within an application.

In Angular 1.x we had router views and directives which worked sort of like components. The idea of directive components became quite popular. They were created by using `directive` with a controller while relying on the `controllerAs` and `bindToController` properties. For example:

```
angular.module('ngcourse')
.directive('ngcHelloComponent', () => ({
  restrict: 'E',
  scope: { name: '=' },
  template: '<span>Hello, .</span>',
  controller: MyComponentCtrl,
  controllerAs: 'ctrl',
  bindToController: true
}));
```

In fact, this concept became so popular that in Angular 1.5 the `.component` method was introduced as syntactic sugar.

```
angular.module('ngcourse')
.component('ngcHelloComponent', {
  bindings: { name: '=' },
  template: '<span>Hello, .</span>',
  controller: MyComponentCtrl
});
```

Creating Components

Components in Angular 2 build upon the lessons learned from Angular 1.5. We define a component's application logic inside a class. To this we attach `@Component`, a TypeScript decorator, which allows you to modify a class or function definition and adds metadata to properties and function arguments.

- `selector` is the element property that we use to tell Angular to create and insert an instance of this component.
- `template` is a form of HTML that tells Angular what needs to be rendered in the DOM.

The Component below will interpolate the value of `name` variable into the template between the double braces ``, what get rendered in the view is`

Hello World

.

```
import {Component} from '@angular/core';

@Component({
  selector: 'hello',
  template: '<p>Hello, </p>'
})
export class Hello {
  name: string;

  constructor() {
    this.name = 'World';
  }
}
```

We need to import the `Component` decarator from `@angular/core` before we can make use of it. To use this component we simply add `<hello></hello>` to the HTML file or another template, and Angular will insert an instance of the `Hello` view between those tags.

[View Example](#)

Application Structure with Components

Table of Content

- [Passing Data into a Component](#)
- [Responding to Component Events](#)
- [Using Two-Way Data Binding](#)
- [Accessing Child Components from Template](#)

A useful way of conceptualizing Angular application design is to look at it as a tree of nested components, each having an isolated scope.

For example consider the following:

```
<TodoApp>
  <TodoList>
    <TodoItem></TodoItem>
    <TodoItem></TodoItem>
    <TodoItem></TodoItem>
  </TodoList>
  <TodoForm></TodoForm>
</TodoApp>
```

At the root we have `TodoApp` which consists of a `TodoList` and a `TodoForm`. Within the list we have several `TodoItem`s. Each of these components is visible to the user, who can interact with these components and perform actions.

Passing Data into a Component

There are two ways to pass data into a component, with 'property binding' and 'event binding'. In Angular 2, data and event change detection happens top-down from parent to children. However for Angular 2 events we can use the DOM event mental model where events flow bottom-up from child to parent. So, Angular 2 events can be treated like regular HTML DOM based events when it comes to cancellable event propagation.

The `@Input()` decorator defines a set of parameters that can be passed down from the component's parent. For example, we can modify the `Hello` component so that `name` can be provided by the parent.

```
import {Component, Input} from '@angular/core';

@Component({
  selector: 'hello',
  template: '<p>Hello, {{name}}</p>'
})
export class Hello {
  @Input() name: string;
}
```

The point of making components is not only encapsulation, but also reusability. Inputs allow us to configure a particular instance of a component.

We can now use our component like so:

```
<!-- To bind to a raw string -->
<hello name="World"></hello>
<!-- To bind to a variable in the parent scope -->
<hello [name]="name"></hello>
```

[View Example](#)

Unlike Angular 1.x, this is one-way binding.

Responding to Component Events

An event handler is specified inside the template using round brackets to denote event binding. This event handler is then coded in the class to process the event.

```
import {Component} from '@angular/core';

@Component({
  selector: 'counter',
  template: `
    <div>
      <p>Count: {{ num }}</p>
      <button (click)="increment()">Increment</button>
    </div>
  `
})
export class Counter {
  num: number = 0;

  increment() {
    this.num++;
  }
}
```

[View Example](#)

To send data out of components via outputs, start by defining the outputs attribute. It accepts a list of output parameters that a component exposes to its parent.

```
import {Component, EventEmitter, Input, Output} from '@angular/core';

@Component({
  selector: 'counter',
  template: `
    <div>
      <p>Count: {{ count }}</p>
      <button (click)="increment()">Increment</button>
    </div>
  `
})
export class Counter {
  @Input() count: number = 0;
  @Output() result: EventEmitter = new EventEmitter();

  increment() {
    this.count++;
    this.result.emit(this.count);
  }
}
```

```
import {Component} from '@angular/core';

@Component({
  selector: 'app',
  template: `
    <div>
      Parent Num: {{ num }}<br />
      Parent Count: {{ parentCount }}
      <counter [count]="num" (result)="onChange($event)">
      </counter>
    </div>
  `
})
export class App {
  num: number;
  parentCount: number;

  constructor() {
    this.num = 0;
    this.parentCount = 0;
  }

  onChange(val: number) {
    this.parentCount = val;
  }
}
```

[View Example](#)

Together a set of input + output bindings define the public API of your component. In our templates we use the [squareBrackets] to pass inputs and the (parenthesis) to handle outputs.

Using Two-Way Data Binding

Two-way data binding combines the input and output binding into a single notation using the `ngModel` directive.

```
<input [(ngModel)]="name" >
```

What this is doing behind the scenes is equivalent to:

```
<input [ngModel]="name" (ngModelChange)="name=$event">
```

To create your own component that supports two-way binding, you must define an `@Output` property to match an `@Input`, but suffix it with the `Change`. The code example below, inside class Counter shows how to make property count support two-way binding.

```
import {Component, Input, Output, EventEmitter} from '@angular/core';

@Component({
  selector: 'counter',
  template: `
    <div>
      <p>
        <ng-content></ng-content>
        Count: {{ count }} -
        <button (click)="increment()">Increment</button>
      </p>
    </div>
  `
})
export class Counter {
  @Input() count: number = 0;
  @Output() countChange: EventEmitter<number> = new EventEmitter<number>();

  increment() {
    this.count++;
    this.countChange.emit(this.count);
  }
}
```

[View Example](#)

Access Child Components From the Template

In our templates, we may find ourselves needing to access values provided by the child components which we use to build our own component.

The most straightforward examples of this may be seen dealing with forms or inputs:

app/my-example.component.html

```
<section>
  <form #myForm="ngForm" (ngSubmit)="submitForm(myForm)">
    <label for="name">Name</label>
    <input type="text" name="name" id="name" ngModel>
    <button type="submit">Submit</button>
  </form>
  Form value: {{formValue}}
</section>
```

app/my-example.component.ts

```
import {Component} from '@angular/core';

@Component({
  selector: 'my-example',
  templateUrl: 'app/my-example.component.html'
})
export class MyExampleComponent {
  formValue: string = JSON.stringify({});
  submitForm (form: NgForm) {
    this.formValue = JSON.stringify(form.value);
  }
}
```

[View Example](#)

This isn't a magic feature which only forms or inputs have, but rather a way of referencing the instance of a child component in your template. With that reference, you can then access public properties and methods on that component.

```
<my-profile #profile></my-profile>
My name is {{ profile.name }}
```

```
@Component({
  selector: 'my-profile',
  templateUrl: 'app/my-profile.component.html'
})
export class MyProfile {
  name: string = 'John Doe';
}
```

[View Example](#)

There are other means of accessing and interfacing with child components, but if you simply need to reference properties or methods of a child, this can be a simple and straightforward method of doing so.

Projection

Projection is a very important concept in Angular 2. It enables developers to build reusable components and make applications more scalable and flexible. To illustrate that, suppose we have a `ChildComponent` like:

```
@Component({
  selector: 'child',
  template: `
    <div>
      <h4>Child Component</h4>
      {{ childContent }}
    </div>
  `
})
export class ChildComponent {
  childContent: string = "Default content";
}
```

What should we do if we want to replace `{{ childContent }}` to any HTML that provided to `ChildComponent`? One tempting idea is to define an `@Input` containing the text, but what if you wanted to provide styled HTML, or other components? Trying to handle this with an `@Input` can get messy quickly, and this is where content projection comes in. Components by default support projection, and you can use the `ngContent` directive to place the projected content in your template.

So, change `ChildComponent` to use projection:

```
import {Component, Input} from '@angular/core';

@Component({
  selector: 'child',
  template: `
    <h4>Child Component</h4>
    <ng-content></ng-content>
  `
})
class ChildComponent {}
```

Then, when we use `ChildComponent` in the template:

```
<child>
  <p>My projected content.</p>
</child>
```

This is telling Angular, that for any markup that appears between the opening and closing tag of `<child>`, to place inside of `<ng-content></ng-content>`.

When doing this, we can have other components, markup, etc projected here and the `ChildComponent` does not need to know about or care what is being provided.

[View Example](#)

But what if we have multiple `<ng-content></ng-content>` and want to specify the position of the projected content to certain `ng-content`? For example, for the previous `ChildComponent`, if we want to format the projected content into an extra `header` and `footer` section:

```
import {Component, Input} from '@angular/core';

@Component({
  selector: 'child',
  template: `
    <h4>Child Component</h4>
    <ng-content select="header"></ng-content>
    <ng-content></ng-content>
    <ng-content select="footer"></ng-content>
  `
})
class ChildComponent {}
```

Then in the template, we can use directives, say, `<header>` to specify the position of projected content to the `ng-content` with `select="header"`:

```
<div>
  <child>
    <header>Header Content</header>
    Main Content
    <footer>Footer Content</footer>
  </child>
</div>
```

Besides using directives, developers can also select a `ng-content` through css class:

```
<ng-content select=".class-select"></ng-content>
```

```
<div class="class-select">  
  div with .class-select  
</div>
```

[View Example](#)

Structuring Applications with Components

As the complexity and size of our application grows, we want to divide responsibilities among our components further.

- *Smart / Container components* are application-specific, higher-level, container components, with access to the application's domain model.
- *Dumb / Presentational components* are components responsible for UI rendering and/or behavior of specific entities passed in via components API (i.e component properties and events). Those components are more in-line with the upcoming Web Component standards.

Using Other Components

Components depend on other components, directives and pipes. For example, `TodoList` relies on `TodoItem`. To let a component know about these dependencies we group them into a module.

```
import {NgModule} from '@angular/core';
import {TodoInput} from './components/todo-input';
import {TodoItem} from './components/todo-item';
import {TodoList} from './components/todo-list';

@NgModule({
  imports: [ ... ],
  declarations: [ TodoList, TodoItem, TodoInput ],
  bootstrap: [ ... ]
})
export class ToDoAppModule { }
```

The property `declarations` expects an array of components, directives and pipes that are part of the module.

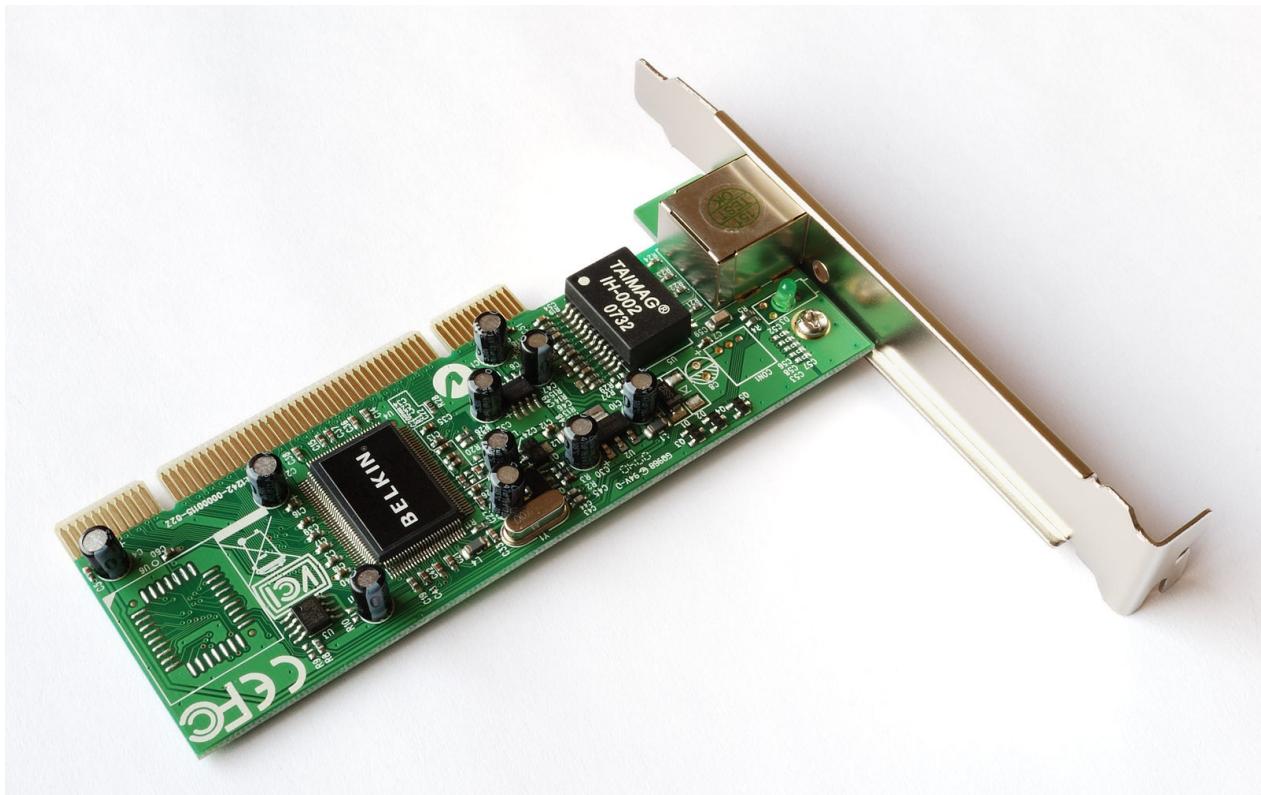
Please see the [Modules section](#) for more info about `NgModule`.

Advanced

Table of Content

- Component Lifecycle
- Accessing Other Components
- View Encapsulation
- ElementRef

Advanced Components



Now that we are familiar with component basics, we can look at some of the more interesting things we can do with them.

Component Lifecycle

A component has a lifecycle managed by Angular itself. Angular manages creation, rendering, data-bound properties etc. It also offers hooks that allow us to respond to key lifecycle events.

Here is the complete lifecycle hook interface inventory:

- `ngOnChanges` - called when an input binding value changes
- `ngOnInit` - after the first `ngOnChanges`
- `ngDoCheck` - after every run of change detection
- `ngAfterContentInit` - after component content initialized
- `ngAfterContentChecked` - after every check of component content
- `ngAfterViewInit` - after component's view(s) are initialized
- `ngAfterViewChecked` - after every check of a component's view(s)
- `ngOnDestroy` - just before the component is destroyed

from [Component Lifecycle](#)

[View Example](#)

Accessing Child Component Classes

@ViewChild and @ViewChildren

The `@ViewChild` and `@ViewChildren` decorators provide access to the class of child component from the containing component.

The `@ViewChild` is a decorator function that takes the name of a component class as its input and finds its selector in the template of the containing component to bind to.

`@ViewChild` can also be passed a template reference variable.

For example, we bind the component class `Alert` to its selector `<my-alert>` and assign it to the property `alert`. This allows us to gain access to class methods, like `show()`.

```
import {Component, ViewChild} from '@angular/core';
import {Alert} from './alert.component';

@Component({
  selector: 'app',
  template: `
    <my-alert>My alert</my-alert>
    <button (click)="showAlert()">Show Alert</button>
  `})
export class App {
  @ViewChild(Alert) alert: Alert;

  showAlert() {
    this.alert.show();
  }
}
```

[View Example](#)

In the interest of separation of concern, we'd normally want to have child elements take care of their own behaviors and pass in an `@Input()`. However, it might be a useful construct in keeping things generic.

When there are multiple embedded components in the template, we can also use `@ViewChildren`. It collects a list of instances of the `Alert` component, stored in a `QueryList` object that behaves similar to an array.

```
import {Component, QueryList, ViewChildren} from '@angular/core';
import {Alert} from './alert.component';

@Component({
  selector: 'app',
  template: `
    <my-alert ok="Next" (close)="showAlert(2)">Step 1: Learn angular</my-alert>
    <my-alert ok="Next" (close)="showAlert(3)">Step 2: Love angular</my-alert>
    <my-alert ok="Close">Step 3: Build app</my-alert>
    <button (click)="showAlert(1)">Show steps</button>
  `
})
export class App {
  @ViewChildren(Alert) alerts: QueryList<Alert>;
  alertsArr = [];

  ngAfterViewInit() {
    this.alertsArr = this.alerts.toArray();
  }

  showAlert(step) {
    this.alertsArr[step - 1].show(); // step 1 is alert index 0
  }
}
```

[View Example](#)

As shown above, given a class type to `@ViewChild` and `@ViewChildren` a child component or a list of children component are selected respectively using their selector from the template. In addition both `@ViewChild` and `@ViewChildren` can be passed a selector string:

```

import {Component, QueryList, ViewChild, ViewChildren} from '@angular/core';
import {Alert} from './alert.component';

@Component({
  selector: 'app',
  template: `
    <my-alert #first ok="Next" (close)="showAlert(2)">Step 1: Learn angular</my-alert>
    <my-alert ok="Next" (close)="showAlert(3)">Step 2: Love angular</my-alert>
    <my-alert ok="Close">Step 3: Build app</my-alert>
    <button (click)="showAlert(1)">Show steps</button>
  `
})
export class App {
  @ViewChild('first') alerts: Alert;
  @ViewChildren(Alert) alerts: QueryList<Alert>;
  alertsArr = [];

  ngAfterViewInit() {
    this.alertsArr = this.alerts.toArray();
  }

  showAlert(step) {
    this.first.show();
  }
}

```

Note that View children will not be set until the 'ngAfterViewInit' lifecycle hook is called.

@ContentChild and @ContentChildren

`@ContentChild` and `@ContentChildren` work the same way as the equivalent `@ViewChild` and `@ViewChildren`, however, the key difference is that `@ContentChild` and `@ContentChildren` select from the **projected content** within the component.

Again, note that content children will not be set until `ngAfterContentInit` component lifecycle hook.

[View Example](#)

View Encapsulation

View encapsulation defines whether the template and styles defined within the component can affect the whole application or vice versa. Angular provides three encapsulation strategies:

- `Emulated` (default) - styles from main HTML propagate to the component. Styles defined in this component's `@Component` decorator are scoped to this component only.
- `Native` - styles from main HTML do not propagate to the component. Styles defined in this component's `@Component` decorator are scoped to this component only.
- `None` - styles from the component propagate back to the main HTML and therefore are visible to all components on the page. Be careful with apps that have `None` and `Native` components in the application. All components with `None` encapsulation will have their styles duplicated in all components with `Native` encapsulation.

```
@Component({  
  ...  
  encapsulation: ViewEncapsulation.None,  
  styles: [ ... ]  
})  
export class Hello { ... }
```

[View Example](#)

ElementRef

Provides access to the underlying native element (DOM element).

```
import {AfterContentInit, Component, ElementRef} from '@angular/core';

@Component({
  selector: 'app',
  template: `
    <h1>My App</h1>
    <pre style="background: #eee; padding: 1rem; border-radius: 3px; overflow: auto;">
      <code>{{ node }}</code>
    </pre>
  `
})
export class App implements AfterContentInit {
  node: string;

  constructor(private elementRef: ElementRef) {}

  ngAfterContentInit() {
    const tmp = document.createElement('div');
    const el = this.elementRef.nativeElement.cloneNode(true);

    tmp.appendChild(el);
    this.node = tmp.innerHTML;
  }
}
```

[View Example](#)

Directives

A Directive modifies the DOM to change appearance, behavior or layout of DOM elements. Directives are one of the core building blocks Angular 2 uses to build applications. In fact, Angular 2 components are in large part directives with templates.

From an Angular 1 perspective, Angular 2 components have assumed a lot of the roles directives used to. The majority of issues that involve templates and dependency injection rules will be done through components, and issues that involve modifying generic behaviour is done through directives.

There are three main types of directives in Angular 2:

- Component - directive with a template.
- *Attribute directives* - directives that change the behavior of a component or element but don't affect the template
- *Structural directives* - directives that change the behavior of a component or element by affecting how the template is rendered

Attribute Directives

Table of Content

- [NgStyle Directive](#)
- [NgClass Directive](#)

Attribute directives are a way of changing the appearance or behavior of a component or a native DOM element. Ideally, a directive should work in a way that is component agnostic and not bound to implementation details.

For example, Angular 2 has built-in attribute directives such as `ngclass` and `ngstyle` that work on any component or element.

NgStyle Directive

Angular 2 provides a built-in directive, `ngStyle`, to modify a component or element's `style` attribute. Here's an example:

```
@Component({
  selector: 'style-example',
  template: `
    <p style="padding: 1rem"
      [ngStyle]="{
        color: 'red',
        'font-weight': 'bold',
        borderBottom: borderStyle
      }">
      <ng-content></ng-content>
    </p>
  `
})
export class StyleExampleComponent {
  borderStyle: string = '1px solid black';
}
```

[View Example](#)

Notice that binding a directive works the exact same way as component attribute bindings. Here, we're binding an expression, an object literal, to the `ngStyle` directive so the directive name must be enclosed in square brackets. `ngStyle` accepts an object whose properties and values define that element's style. In this case, we can see that both kebab case and lower camel case can be used when specifying a style property. Also notice that both the html `style` attribute and Angular 2 `ngStyle` directive are combined when styling the element.

NgClass Directive

The `ngClass` directive changes the `class` attribute that is bound to the component or element it's attached to. There are a few different ways of using the directive.

Binding a string

We can bind a string directly to the attribute. This works just like adding an html `class` attribute.

```
@Component({
  selector: 'class-as-string',
  template: `
    <p ngClass="centered-text underlined" class="orange">
      <ng-content></ng-content>
    </p>
  `,
  styles: [
    .centered-text {
      text-align: center;
    }

    .underlined {
      border-bottom: 1px solid #ccc;
    }

    .orange {
      color: orange;
    }
  ]
})
export class ClassAsStringComponent {
```

[View Example](#)

In this case, we're binding a string directly so we avoid wrapping the directive in square brackets. Also notice that the `ngClass` works with the `class` attribute to combine the final classes.

Binding an array

```
@Component({
  selector: 'class-as-array',
  template: `
    <p [ngClass]=["'warning', 'big']">
      <ng-content></ng-content>
    </p>
  `,
  styles: [
    .warning {
      color: red;
      font-weight: bold;
    }

    .big {
      font-size: 1.2rem;
    }
  ]
})
export class ClassAsArrayComponent {
```

[View Example](#)

Here, since we are binding to the `ngClass` directive by using an expression, we need to wrap the directive name in square brackets. Passing in an array is useful when you want to have a function put together the list of applicable class names.

Binding an object

Lastly, an object can be bound to the directive. Angular 2 applies each property name of that object to the component if that property is true.

```
@Component({
  selector: 'class-as-object',
  template: `
    <p [ngClass]="{ card: true, dark: false, flat: flat }">
      <ng-content></ng-content>
      <br/>
      <button type="button" (click)="flat=!flat">Toggle Flat</button>
    </p>
  `,
  styles: [
    .card {
      border: 1px solid #eee;
      padding: 1rem;
      margin: 0.4rem;
      font-family: sans-serif;
      box-shadow: 2px 2px 2px #888888;
    }

    .dark {
      background-color: #444;
      border-color: #000;
      color: #fff;
    }

    .flat {
      box-shadow: none;
    }
  ]
})
export class ClassAsObjectComponent {
  flat: boolean = true;
}
```

[View Example](#)

Here we can see that since the object's `card` and `flat` properties are true, those classes are applied but since `dark` is false, it's not applied.

Structural Directives

Table of Content

- [NgIf Directive](#)
- [NgFor Directive](#)
- [NgSwitch Directives](#)
- [Combining Directives](#)

Structural Directives are a way of handling how a component or element renders through the use of the `template` tag. This allows us to run some code that decides what the final rendered output will be. Angular 2 has a few built-in structural directives such as `ngIf` , `ngFor` , and `ngSwitch` .

Note: For those who are unfamiliar with the `template` tag, it is an HTML element with a few special properties. Content nested in a template tag is not rendered on page load and is something that is meant to be loaded through code at runtime. For more information on the `template` tag, visit the [MDN documentation](#).

Structural directives have their own special syntax in the template that works as syntactic sugar.

```
@Component({
  selector: 'directive-example',
  template: `
    <p *structuralDirective="expression">
      Under a structural directive.
    </p>
  `
})
```

Instead of being enclosed by square brackets, our dummy structural directive is prefixed with an asterisk. Notice that the binding is still an expression binding even though there are no square brackets. That's due to the fact that it's syntactic sugar that allows using the directive in a more intuitive way and similar to how directives were used in Angular 1. The component template above is equivalent to the following:

```
@Component({
  selector: 'directive-example',
  template: `
    <template [structuralDirective]="expression">
      <p>
        Under a structural directive.
      </p>
    </template>
  `
})
```

Here, we see what was mentioned earlier when we said that structural directives use the `template` tag. Angular 2 also has a built-in `template` directive that does the same thing:

```
@Component({
  selector: 'directive-example',
  template: `
    <p template="structuralDirective expression">
      Under a structural directive.
    </p>
  `
})
```

NgIf Directive

The `ngIf` directive conditionally adds or removes content from the DOM based on whether or not an expression is true or false.

Here's our app component, where we bind the `ngIf` directive to an example component.

```
@Component({
  selector: 'app',
  template: `
    <button type="button" (click)="toggleExists()">Toggle Component</button>
    <hr/>
    <if-example *ngIf="exists">
      Hello
    </if-example>
  `
})
export class AppComponent {
  exists: boolean = true;

  toggleExists() {
    this.exists = !this.exists;
  }
}
```

[View Example](#)

Clicking the button will toggle whether or not `IfExampleComponent` is a part of the DOM and not just whether it is visible or not. This means that every time the button is clicked,

`IfExampleComponent` will be created or destroyed. This can be an issue with components that have expensive create/destroy actions. For example, a component could have a large child subtree or make several HTTP calls when constructed. In these cases it may be better to avoid using `ngIf` if possible.

NgFor Directive

The `ngFor` directive is a way of repeating a template by using each item of an iterable as that template's context.

```
@Component({
  selector: 'app',
  template: `
    <for-example *ngFor="let episode of episodes" [episode]="episode">
      {{episode.title}}
    </for-example>
  `
})
export class AppComponent {
  episodes: any[] = [
    { title: 'Winter Is Coming', director: 'Tim Van Patten' },
    { title: 'The Kingsroad', director: 'Tim Van Patten' },
    { title: 'Lord Snow', director: 'Brian Kirk' },
    { title: 'Cripples, Bastards, and Broken Things', director: 'Brian Kirk' },
    { title: 'The Wolf and the Lion', director: 'Brian Kirk' },
    { title: 'A Golden Crown', director: 'Daniel Minahan' },
    { title: 'You Win or You Die', director: 'Daniel Minahan' },
    { title: 'The Pointy End', director: 'Daniel Minahan' }
  ];
}
```

[View Example](#)

The `ngFor` directive has a different syntax from other directives we've seen. If you're familiar with the [for...of statement](#), you'll notice that they're almost identical. `ngFor` lets you specify an iterable object to iterate over and the name to refer to each item by inside the scope. In our example, you can see that `episode` is available for interpolation as well as property binding. The directive does some extra parsing so that when this is expanded to template form, it looks a bit different:

```
@Component({
  selector: 'app',
  template: `
    <template ngFor [ngForOf]="episodes" let-episode>
      <for-example [episode]="episode">
        {{episode.title}}
      </for-example>
    </template>
  `
})
```

[View Example](#)

Notice that there is an odd `let-episode` property on the template element. The `ngFor` directive provides some variables as context within its scope. `let-episode` is a context binding and here it takes on the value of each item of the iterable. `ngFor` also provides some other values that can be bound to:

- `index` - position of the current item in the iterable starting at `0`
- `first` - `true` if the current item is the first item in the iterable
- `last` - `true` if the current item is the last item in the iterable
- `even` - `true` if the current index is an even number
- `odd` - `true` if the current index is an odd number

```
@Component({
  selector: 'app',
  template: `
    <for-example
      *ngFor="let episode of episodes; let i = index; let isOdd = odd"
      [episode]="episode"
      [ngClass]={`${ isOdd }`}>
      NaN.
    </for-example>

    <hr/>

    <h2>Desugared</h2>

    <template ngFor [ngForOf]="episodes" let-episode let-i="index" let-isOdd="odd">
      <for-example [episode]="episode" [ngClass]={`${ isOdd }`}>
        NaN.
      </for-example>
    </template>
  `

})
```

[View Example](#)

trackBy

Often `ngFor` is used to iterate through a list of objects with a unique ID field. In this case, we can provide a `trackBy` function which helps Angular keep track of items in the list so that it can detect which items have been added or removed and improve performance.

Angular 2 will try and track objects by reference to determine which items should be created and destroyed. However, if you replace the list with a new source of objects, perhaps as a result of an API request - we can get some extra performance by telling Angular 2 how we

want to keep track of things.

For example, if the `Add Episode` button was to make a request and return a new list of episodes, we might not want to destroy and re-create every item in the list. If the episodes have a unique ID, we could add a `trackBy` function:

```
@Component({
  selector: 'app',
  template: `
    <button (click)="addOtherEpisode()" [disabled]="otherEpisodes.length === 0">Add Episode</button>
    <for-example
      *ngFor="let episode of episodes;
      let i = index; let isOdd = odd;
      trackBy: trackById" [episode]="episode"
      [ngClass]={`${isOdd ? 'odd' : ''}`">

    </for-example>
  `
})
export class AppComponent {

  otherEpisodes: any[] = [
    { title: 'Two Swords', director: 'D. B. Weiss', id: 8 },
    { title: 'The Lion and the Rose', director: 'Alex Graves', id: 9 },
    { title: 'Breaker of Chains', director: 'Michelle MacLaren', id: 10 },
    { title: 'Oathkeeper', director: 'Michelle MacLaren', id: 11 }
  ]

  episodes: any[] = [
    { title: 'Winter Is Coming', director: 'Tim Van Patten', id: 0 },
    { title: 'The Kingsroad', director: 'Tim Van Patten', id: 1 },
    { title: 'Lord Snow', director: 'Brian Kirk', id: 2 },
    { title: 'Cripples, Bastards, and Broken Things', director: 'Brian Kirk', id: 3 },
    { title: 'The Wolf and the Lion', director: 'Brian Kirk', id: 4 },
    { title: 'A Golden Crown', director: 'Daniel Minahan', id: 5 },
    { title: 'You Win or You Die', director: 'Daniel Minahan', id: 6 },
    { title: 'The Pointy End', director: 'Daniel Minahan', id: 7 }
  ];

  addOtherEpisode() {
    // We want to create a new object reference for sake of example
    let episodesCopy = JSON.parse(JSON.stringify(this.episodes))
    this.episodes=[...episodesCopy, this.otherEpisodes.pop()];
  }
  trackById(index: number, episode: any): number {
    return episode.id;
  }
}
```

To see how this can affect the `ForExample` component, let's add some logging to it.

```
export class ForExampleComponent {  
  @Input() episode;  
  
  ngOnInit() {  
    console.log('component created', this.episode)  
  }  
  ngOnDestroy() {  
    console.log('destroying component', this.episode)  
  }  
}
```

[View Example](#)

When we view the example, as we click on `Add Episode`, we can see console output indicating that only one component was created - for the newly added item to the list.

However, if we were to remove the `trackBy` from the `*ngFor` - every time we click the button, we would see the items in the component getting destroyed and recreated.

[View Example Without trackBy](#)

NgSwitch Directives

`ngSwitch` is actually comprised of two directives, an attribute directive and a structural directive. It's very similar to a `switch statement` in JavaScript and other programming languages, but in the template.

```

@Component({
  selector: 'app',
  template: `
    <div class="tabs-selection">
      <tab [active]="isSelected(1)" (click)="setTab(1)">Tab 1</tab>
      <tab [active]="isSelected(2)" (click)="setTab(2)">Tab 2</tab>
      <tab [active]="isSelected(3)" (click)="setTab(3)">Tab 3</tab>
    </div>

    <div [ngSwitch]="tab">
      <tab-content *ngSwitchCase="1">Tab content 1</tab-content>
      <tab-content *ngSwitchCase="2">Tab content 2</tab-content>
      <tab-content *ngSwitchCase="3"><tab-3></tab-3></tab-content>
      <tab-content *ngSwitchDefault>Select a tab</tab-content>
    </div>
  `
})

export class AppComponent {
  tab: number = 0;

  setTab(num: number) {
    this.tab = num;
  }

  isSelected(num: number) {
    return this.tab === num;
  }
}

```

[View Example](#)

Here we see the `ngSwitch` attribute directive being attached to an element. This expression bound to the directive defines what will be compared against in the switch structural directives. If an expression bound to `ngSwitchCase` matches the one given to `ngSwitch`, those components are created and the others destroyed. If none of the cases match, then components that have `ngSwitchDefault` bound to them will be created and the others destroyed. Note that multiple components can be matched using `ngSwitchCase` and in those cases all matching components will be created. Since components are created or destroyed be aware of the costs in doing so.

Using Multiple Structural Directives

Sometimes we'll want to combine multiple structural directives together, like iterating using `ngFor` but wanting to do an `ngIf` to make sure that the value matches some or multiple conditions. Combining structural directives can lead to unexpected results however, so Angular 2 requires that a template can only be bound to one directive at a time. To apply multiple directives we'll have to expand the sugared syntax or nest template tags.

```
@Component({
  selector: 'app',
  template: `
    <template ngFor [ngForOf]="[1,2,3,4,5,6]" let-item>
      <div *ngIf="item > 3">
        {{item}}
      </div>
    </template>
  `
})
```

[View Example](#)

The previous tabs example can use `ngFor` and `ngSwitch` if the tab title and content is abstracted away into the component class.

```
import {Component} from '@angular/core';

@Component({
  selector: 'app',
  template: `
    <div class="tabs-selection">
      <tab
        *ngFor="let tab of tabs; let i = index"
        [active]="isSelected(i)"
        (click)="setTab(i)">

        {{ tab.title }}
      </tab>
    </div>

    <div [ngSwitch]="tabNumber">
      <template ngFor [ngForOf]="tabs" let-tab let-i="index">
        <tab-content *ngSwitchCase="i">
          {{tab.content}}
        </tab-content>
      </template>
      <tab-content *ngSwitchDefault>Select a tab</tab-content>
    </div>
  `

})

export class AppComponent {
  tabNumber: number = -1;

  tabs = [
    { title: 'Tab 1', content: 'Tab content 1' },
    { title: 'Tab 2', content: 'Tab content 2' },
    { title: 'Tab 3', content: 'Tab content 3' },
  ];

  setTab(num: number) {
    this.tabNumber = num;
  }

  isSelected(num: number) {
    return this.tabNumber === i;
  }
}
```

[View Example](#)

Advanced

Table of Content

- [Creating an Attribute Directive](#)
 - [Listening to an Element Host](#)
 - [Setting Properties in a Directive](#)
- [Creating a Structural Directive](#)
 - [View Containers and Embedded Views](#)
 - [Providing Context Variables to Directives](#)

Creating a Structural Directive

We'll create a `delay` structural directive that delays instantiation of a component or element. This can potentially be used for cosmetic effect or for manually handling timing of when components are loaded, either for performance or UX.

```
@Directive({
  selector: '[delay]'
})
export class DelayDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainerRef: ViewContainerRef
  ) { }

  @Input('delay')
  set delayTime(time: number): void { }
}
```

[View Example](#)

We use the same `@Directive` class decorator as attribute directives and define a selector in the same way. One big difference here is that due to the nature of structural directives being bound to a template, we have access to `TemplateRef`, an object representing the `template` tag the directive is attached to. We also add an input property in a similar way, but this time with a `set` handler so we can execute some code when Angular 2 performs the binding. We bind `delay` in exactly the same way as the Angular 2 built-in structural directives.

```
@Component({
  selector: 'app',
  template: `
    <div *ngFor="let item of [1,2,3,4,5,6]">
      <card *delay="500 * item">
        {{item}}
      </card>
    </div>
  `
})

export class App { }
```

[View Example](#)

Notice that no content is being rendered however. This is due to Angular 2 simulating the `html template tag` and not rendering any child elements by default. To be able to get this content to render, we'll have to attach the template given by `TemplateRef` as an *embedded view to a view container*.

Listening to an Element Host

Listening to the *host* - that is, the DOM element the directive is attached to - is among the primary ways directives extend the component or element's behavior. Previously, we saw its common use case.

```
@Directive({
  selector: '[myDirective]'
})
class MyDirective {
  @HostListener('click', ['$event'])
  onClick() {}
}
```

We can also respond to external events, such as from `window` or `document`, by adding the target in the listener.

```
@Directive({
  selector: `[highlight]`
})
export class HighlightDirective {
  constructor(private el: ElementRef, private renderer: Renderer) {}

  @HostListener('document:click', ['$event'])
  handleClick(event: Event) {
    if (this.el.nativeElement.contains(event.target)) {
      this.highlight('yellow');
    } else {
      this.highlight(null);
    }
  }

  highlight(color) {
    this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', color);
  }
}
```

[View Example](#)

Although less common, we can also use `@HostListener` if we'd like to register listeners on the host element of a Component.

Host Elements

The concept of a host element applies to both directives and components.

For a directive, the concept is fairly straight forward. Whichever template tag you place your directive attribute on is considered the host element. If we were implementing the `HighlightDirective` above like so:

```
<div>
  <p highlight>
    <span>Text to be highlighted</span>
  </p>
</div>
```

The `<p>` tag would be considered the host element. If we were using a custom `TextBoxComponent` as the host, the code would look like this:

```
<div>
  <my-text-box highlight>
    <span>Text to be highlighted</span>
  </my-text-box>
</div>
```

In the context of a Component, the host element is the tag that you create through the `selector` string in the component configuration. For the `TextBoxComponent` in the example above, the host element in the context of the component class would be the `<my-text-box>` tag.

Setting Properties with a Directive

We can use attribute directives to affect the value of properties on the host node by using the `@HostBinding` decorator.

The `@HostBinding` decorator allows us to programmatically set a property value on the directive's host element. It works similarly to a property binding defined in a template, except it specifically targets the host element. The binding is checked for every change detection cycle, so it can change dynamically if desired.

For example, lets say that we want to create a directive for buttons that dynamically adds a class when we press on it. That could look something like:

```
import { Directive, HostBinding, HostListener } from '@angular/core';

@Directive({
  selector: '[buttonPress]'
})
export class ButtonPressDirective {
  @HostBinding('attr.role') role = 'button';
  @HostBinding('class.pressed') isPressed: boolean;

  @HostListener('mousedown') hasPressed() {
    this.isPressed = true;
  }
  @HostListener('mouseup') hasReleased() {
    this.isPressed = false;
  }
}
```

Notice that for both use cases of `@HostBinding` we are passing in a string value for which property we want to affect. If we don't supply a string to the decorator, then the name of the class member will be used instead.

In the first `@HostBinding`, we are statically setting the `role` attribute to `button`. For the second example, the `pressed` class will be applied when `isPressed` is true.

Tip: Though less common, `@HostBinding` can also be applied to Components if required.

Creating an Attribute Directive

Let's start with a simple button that moves a user to a different page.

```
@Component({
  selector: 'visit-rangle',
  template: `
    <button type="button" (click)="visitRangle()">Visit Rangle</button>
  `
})
export class VisitRangleComponent {
  visitRangle() {
    location.href = 'https://rangle.io';
  }
}
```

[View Example](#)

We're polite, so rather than just sending the user to a new page, we're going to ask if they're ok with that first by creating an attribute directive and attaching that to the button.

```
@Directive({
  selector: '[confirm]'
})
export class ConfirmDirective {

  @HostListener('click', ['$event'])
  confirmFirst(event: Event) {
    return window.confirm('Are you sure you want to do this?');
  }
}
```

[View Example](#)

Directives are created by using the `@Directive` decorator on a class and specifying a selector. For directives, the selector name must be camelCase and wrapped in square brackets to specify that it is an attribute binding. We're using the `@HostListener` decorator to listen in on events on the component or element it's attached to. In this case we're watching the `click` event and passing in the event details which are given by the special `$event` keyword. Next, we want to attach this directive to the button we created earlier.

```
template: `<button type="button" (click)="visitRangle()" confirm>Visit Rangle</button>`
```

[View Example](#)

Notice, however, that the button doesn't work quite as expected. That's because while we're listening to the click event and showing a confirm dialog, the component's click handler runs before the directive's click handler and there's no communication between the two. To do this we'll need to rewrite our directive to work with the component's click handler.

```
@Directive({
  selector: `[confirm]`
})
export class ConfirmDirective {
  @Input('confirm') onConfirmed: Function = () => {};

  @HostListener('click', ['$event'])
  confirmFirst() {
    const confirmed = window.confirm('Are you sure you want to do this?');

    if(confirmed) {
      this.onConfirmed();
    }
  }
}
```

[View Example](#)

Here, we want to specify what action needs to happen after a confirm dialog's been sent out and to do this we create an input binding just like we would on a component. We'll use our directive name for this binding and our component code changes like this:

```
<button type="button" [confirm]="visitRangle">Visit Rangle</button>
```

[View Example](#)

Now our button works just as we expected. We might want to be able to customize the message of the confirm dialog however. To do this we'll use another binding.

```
@Directive({
  selector: `[confirm]`
})
export class ConfirmDirective {
  @Input('confirm') onConfirmed: Function = () => {};
  @Input() confirmMessage: string = 'Are you sure you want to do this?';

  @HostListener('click', ['$event'])
  confirmFirst() {
    const confirmed = window.confirm(this.confirmMessage);

    if(confirmed) {
      this.onConfirmed();
    }
  }
}
```

[View Example](#)

Our directive gets a new input property that represents the confirm dialog message, which we pass in to `window.confirm` call. To take advantage of this new input property, we add another binding to our button.

```
<button
  type="button"
  [confirm]="visitRangle"
  confirmMessage="Click ok to visit Rangle.io!">
  Visit Rangle
</button>
```

[View Example](#)

Now we have a button with a customizable confirm message before it moves you to a new url.

View Containers and Embedded Views

View Containers are containers where one or more *Views* can be attached. Views represent some sort of layout to be rendered and the context under which to render it. View containers are anchored to components and are responsible for generating its output so this means that changing which views are attached to the view container affect the final rendered output of the component.

Two types of views can be attached to a view container: *Host Views* which are linked to a *Component*, and *Embedded Views* which are linked to a *template*. Since structural directives interact with templates, we are interested in using *Embedded Views* in this case.

```
import {Directive, Input, TemplateRef, ViewContainerRef} from '@angular/core';

@Directive({
  selector: '[delay]'
})
export class DelayDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainerRef: ViewContainerRef
  ) { }

  @Input('delay')
  set delayTime(time: number): void {
    setTimeout(
      () => {
        this.viewContainerRef.createEmbeddedView(this.templateRef);
      },
      time);
  }
}
```

[View Example](#)

Directives get access to the view container by injecting a `viewContainerRef`. Embedded views are created and attached to a view container by calling the `viewContainerRef`'s `createEmbeddedView` method and passing in the template. We want to use the template our directive is attached to so we pass in the injected `TemplateRef`.

Providing Context Variables to Directives

Suppose we want to record some metadata on how our directive affected components and make this data available to them. For example, in our `delay` directive, we're making a `setTimeout` call, which in JavaScript's single-threaded asynchronous model means that it may not run after the exact time we provided. We'll capture the exact time it loads and make that variable available in the template.

```
export class DelayContext {
  constructor(private loadTime: number) { }

}

@Directive({
  selector: '[delay]'
})
export class DelayDirective {
  constructor(
    private templateRef: TemplateRef<DelayContext>,
    private viewContainerRef: ViewContainerRef
  ) { }

  @Input('delay')
  set delayTime(time: number): void {
    setTimeout(
      () => {
        this.viewContainerRef.createEmbeddedView(
          this.templateRef,
          new DelayContext(performance.now())
        );
      },
      time);
  }
}
```

[View Example](#)

We've made a few changes to our `delay` directive. We've created a new `DelayContext` class that contains the context that we want to provide to our directive. In this case, we want to capture the actual time the `createEmbeddedView` call occurs and make that available as `loadTime` in our directive. We've also provided our new class as the generic argument to the `TemplateRef` function. This enables static analysis and lets us make sure our calls to `createEmbeddedView` pass in a variable of type `DelayContext`. In our `createEmbeddedView` call we pass in our variable which has captured the time of the method call.

In the component using `delay`, we access the `loadTime` context variable in the same way we access variables in `ngFor`.

```
@Component({
  selector: 'app',
  template: `
    <div *ngFor="let item of [1,2,3,4,5,6]">
      <card *delay="500 * item; let loaded = loadTime">
        <div class="main">{{item}}</div>
        <div class="sub">{{loaded | number:'1.4-4'}}</div>
      </card>
    </div>
  `
})
```

[View Example](#)

Pipes



Angular 2 provides a new way of filtering data: `pipes`. Pipes are a replacement for Angular 1.x's `filters`. Most of the built-in filters from Angular 1.x have been converted to Angular 2 pipes; a few other handy ones have been included as well.

Using Pipes

Like a filter, a pipe also takes data as input and transforms it to the desired output. A basic example of using pipes is shown below:

```
import {Component} from '@angular/core';

@Component({
  selector: 'product-price',
  template: `<p>Total price of product is {{ price | currency }}</p>`
})
export class ProductPrice {
  price: number = 100.1234;
}
```

[View Example](#)

Passing Parameters

A pipe can accept optional parameters to modify the output. To pass parameters to a pipe, simply add a colon and the parameter value to the end of the pipe expression:

```
pipeName: parameterValue
```

You can also pass multiple parameters this way:

```
pipeName: parameter1: parameter2
```

```
import {Component} from '@angular/core';

@Component({
  selector: 'product-price',
  template: '<p>Total price of product is {{ price | currency: "CAD": true: "1.2-4" }}</p>'
})
export class ProductPrice {
  price: number = 100.123456;
}
```

[View Example](#)

Chaining Pipes

We can chain pipes together to make use of multiple pipes in one expression.

```
import {Component} from '@angular/core';

@Component({
  selector: 'product-price',
  template: '<p>Total price of product is {{ price | currency: "CAD": true: "1.2-4" | lowercase }}</p>'
})
export class ProductPrice {
  price: number = 100.123456;
}
```

[View Example](#)

Custom Pipes

Angular 2 allows you to create your own custom pipes:

```
import {Pipe, PipeTransform} from '@angular/core';

const FILE_SIZE_UNITS = ['B', 'KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'];
const FILE_SIZE_UNITS_LONG = ['Bytes', 'Kilobytes', 'Megabytes', 'Gigabytes', 'Petabytes',
  'Exabytes', 'Zettabytes', 'Yottabytes'];

@Pipe({
  name: 'formatFileSize'
})
export class FormatFileSizePipe implements PipeTransform {
  transform(sizeInBytes: number, longForm: boolean): string {
    const units = longForm
      ? FILE_SIZE_UNITS_LONG
      : FILE_SIZE_UNITS;
    let power = Math.round(Math.log(sizeInBytes)/Math.log(1024));
    power = Math.min(power, units.length - 1);
    const size = sizeInBytes / Math.pow(1024, power); // size in new units
    const formattedSize = Math.round(size * 100) / 100; // keep up to 2 decimals
    const unit = units[power];
    return `${formattedSize} ${unit}`;
  }
}
```

Each custom pipe implementation must:

- have the `@Pipe` decorator with pipe metadata that has a `name` property. This value will be used to call this pipe in template expressions. It must be a valid JavaScript identifier.
- implement the `PipeTransform` interface's `transform` method. This method takes the value being piped and a variable number of arguments of any type and return a transformed ("piped") value.

Each colon-delimited parameter in the template maps to one method argument in the same order.

```
import {Component} from '@angular/core';

@Component({
  selector: 'hello',
  template: `
    <div>
      <p *ngFor="let f of fileSizes">{{ f | formatFileSize }}</p>
      <p>{{ largeFileSize | formatFileSize:true }}</p>
    </div>`
})
export class Hello {
  fileSizes = [10, 100, 1000, 10000, 100000, 10000000, 1000000000];
  largeFileSize = Math.pow(10, 15)
}
```

[View Example](#)

Stateful Pipes

There are two categories of pipes:

- *Stateless* pipes are pure functions that flow input data through without remembering anything or causing detectable side-effects. Most pipes are stateless. The `CurrencyPipe` we used and the length pipe we created are examples of a stateless pipe.
- *Stateful* pipes are those which can manage the state of the data they transform. A pipe that creates an HTTP request, stores the response and displays the output, is a stateful pipe. Stateful Pipes should be used cautiously.

Angular 2 provides `AsyncPipe`, which is stateful.

AsyncPipe

`AsyncPipe` can receive a `Promise` or `Observable` as input and subscribe to the input automatically, eventually returning the emitted value(s). It is stateful because the pipe maintains a subscription to the input and its returned values depend on that subscription.

```
import {Component} from '@angular/core';
import {Observable} from 'rxjs/Observable';

@Component({
  selector: 'product-price',
  template: `
    <p>Total price of product is {{fetchPrice | async | currency: "CAD": true: "1.2-2"
}}</p>
    <p>Seconds: {{seconds | async}}</p>
  `
})
export class ProductPrice {
  count: number = 0;
  fetchPrice: Promise<number> = new Promise((resolve, reject) => {
    setTimeout(() => resolve(10), 500);
  });

  seconds: Observable<number> = new Observable(observer => {
    setInterval(() => { observer.next(this.count++); }, 1000);
  });
}
```

[View Example](#)

Implementing Stateful Pipes

Pipes are stateless by default. We must declare a pipe to be stateful by setting the `pure` property of the `@Pipe` decorator to false. This setting tells Angular's change detection system to check the output of this pipe each cycle, whether its input has changed or not.

```

import {Pipe, PipeTransform, ChangeDetectorRef} from '@angular/core';
import {Observable} from 'rxjs/Observable';
import 'rxjs/add/observable/timer';
import 'rxjs/add/operator/take';

/**
 * On number change, animates from `oldNumber` to `newNumber`
 */
// naive implementation assumes small number increments
@Pipe({
  name: 'animateNumber',
  pure: false
})
export class AnimateNumberPipe implements PipeTransform {
  private currentNumber: number = null; // intermediary number
  private newNumber: number = null;
  private subscription;

  constructor(private cdRef: ChangeDetectorRef) {}

  transform(newNumber: number): string {
    if (this.newNumber === null) { // set initial value
      this.currentNumber = this.newNumber = newNumber;
    }
    if (newNumber !== this.newNumber) {
      if (this.subscription) {
        this.currentNumber = this.newNumber;
        this.subscription.unsubscribe();
      }
      this.newNumber = newNumber;
      const oldNumber = this.currentNumber;
      const direction = ((newNumber - oldNumber) > 0) ? 1 : -1;
      const numbersToCount = Math.abs(newNumber - oldNumber) + 1;
      this.subscription = Observable.timer(0, 100) // every 100 ms
        .take(numbersToCount)
        .subscribe(
          () => {
            this.currentNumber += direction;
            this.cdRef.markForCheck();
          },
          null,
          () => this.subscription = null
        );
    }
    return this.currentNumber;
  }
}

```

[View Example](#)

Services

Multiple components will need access to data and we don't want to copy and paste the same code over and over. Instead, we'll create a single reusable data service and learn to inject it in the components that need it.

Refactoring data access to a separate service keeps the component lean and focused on supporting the view. It also makes it easier to unit test the component with a mock service.

- A service is the mechanism used to share functionalities over Components (or with one Component if our app contains only one Component)
- As you may already know, we can create Components in angular 2 and nest multiple Components together with selectors. Once our Components are nested we need to manipulate some Data and maybe this Data is not just a variable within our Component. Or we want to do some calculations.
- In real life Data comes from Server. (e.g. JSON file)
- Service is the best way to handle this.
- Service is the best place from where we can bring our external Data to our app. Or do some repetitive task or calculations.
- Service can be shared between as many as Components we want.
- Or maybe you want to share other things (functionalities) over multiple Components or Portions of your app? The answer is : Service

Basic

Observable

Table of Content

- Using Observables
- Error Handling
- Disposing Subscriptions and Releasing Resources
- Observables vs Promises
- Using Observables From Other Sources
- Observables Array Operations
- Combining Streams with flatMap
- Cold vs Hot Observables
- Summary

An exciting new feature used with Angular 2 is the `Observable`. This isn't an Angular 2 specific feature, but rather a proposed standard for managing async data that will be included in the release of ES7. Observables open up a continuous channel of communication in which multiple values of data can be emitted over time. From this we get a pattern of dealing with data by using array-like operations to parse, modify and maintain data. Angular 2 uses observables extensively - you'll see them in the HTTP service and the event system.

Forms

An application without user input is just a page. Capturing input from the user is the cornerstone of any application. In many cases, this means dealing with forms and all of their complexities.

Angular 2 is much more flexible than Angular 1 for handling forms — we are no longer restricted to relying solely on `ngModel`. Instead, we are given degrees of simplicity and power, depending on the form's purpose.

- [Template-Driven Forms](#) use built-in directives to create straightforward form components with minimal code.
- [Model-Driven Forms](#) uses the provided APIs to handle more complex validation and subforms.

Getting Started

Opt-In APIs

Before we dive into any of the form features, we need to do a little bit of housekeeping. We need to bootstrap our application using the `FormsModule` and/or `ReactiveFormsModule`.

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic'
import { ReactiveFormsModule, FormsModule } from '@angular/forms';
import { MyApp } from './components'

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule
  ],
  declarations: [MyApp],
  bootstrap: [MyApp]
})
export class AppModule {

}

platformBrowserDynamic().bootstrapModule(MyAppModule)
```

Input Labeling

Most of the form examples use the following HTML5 style for labeling inputs:

```
<label for="name">Name</label>
<input type="text" name="username" id="name">
```

Angular 2 also supports the alternate HTML5 style, which precludes the necessity of `id`s on `<input>`s:

```
<label>
  Name
  <input type="text" name="username">
</label>
```


Template-Driven Forms

Table of Content

- Nesting Form Data
- Using Template Model Binding
- Validating Template-Driven Forms

The most straightforward approach to building forms in Angular 2 is to take advantage of the directives provided for you.

First, consider a typical form:

```
<form method="POST" action="/register" id="SignupForm">
  <label for="email">Email</label>
  <input type="text" name="email" id="email">

  <label for="password">Password</label>
  <input type="password" name="password" id="password">

  <button type="submit">Sign Up</button>
</form>
```

Angular 2 has already provided you a `form` directive, and form related directives such as `input`, etc which operates under the covers. For a basic implementation, we just have to add a few attributes and make sure our component knows what to do with the data.

index.html

```
<signup-form>Loading...</signup-form>
```

signup-form.component.html

```
<form #signupForm="ngForm" (ngSubmit)="registerUser(signupForm)">
  <label for="email">Email</label>
  <input type="text" name="email" id="email" ngModel>

  <label for="password">Password</label>
  <input type="password" name="password" id="password" ngModel>

  <button type="submit">Sign Up</button>
</form>
```

signup-form.component.ts

```
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'signup-form',
  templateUrl: 'app/signup-form.component.html',
})
export class SignupForm {
  registerUser(form: NgForm) {
    console.log(form.value);
    // {email: '...', password: '...'}
    // ...
  }
}
```

Nesting Form Data

If you find yourself wrestling to fit nested trees of data inside of a flat form, Angular has you covered for both simple and complex cases.

Let's assume you had a payment endpoint which required data, similar to the following:

```
{  
  "contact": {  
    "firstname": "Bob",  
    "lastname": "McKenzie",  
    "email": "BobAndDoug@GreatWhiteNorth.com",  
    "phone": "555-TAKE-OFF"  
  },  
  "address": {  
    "street": "123 Some St",  
    "city": "Toronto",  
    "region": "ON",  
    "country": "CA",  
    "code": "H0H 0H0"  
  },  
  "paymentCard": {  
    "provider": "Credit Lending Company Inc",  
    "cardholder": "Doug McKenzie",  
    "number": "123 456 789 012",  
    "verification": "321",  
    "expiry": "2020-02"  
  }  
}
```

While forms are flat and one-dimensional, the data built from them is not. This leads to complex transforms to convert the data you've been given into the shape you need.

Worse, in cases where it is possible to run into naming collisions in form inputs, you might find yourself using long and awkward names for semantic purposes.

```
<form>
  <fieldset>
    <legend>Contact</legend>

    <label for="contact_first-name">First Name</label>
    <input type="text" name="contact_first-name" id="contact_first-name">

    <label for="contact_last-name">Last Name</label>
    <input type="text" name="contact_last-name" id="contact_last-name">

    <label for="contact_email">Email</label>
    <input type="email" name="contact_email" id="contact_email">

    <label for="contact_phone">Phone</label>
    <input type="text" name="contact_phone" id="contact_phone">
  </fieldset>

  <!-- ... -->

</form>
```

A form handler would have to convert that data into a form that your API expects. Thankfully, this is something Angular 2 has a solution for.

ngModelGroup

When building a template-driven form in Angular 2, we can lean on the `ngModelGroup` directive to arrive at a cleaner implementation, while Angular does the heavy lifting of converting form-fields into nested data.

```
<form #paymentForm="ngForm" (ngSubmit)="purchase(paymentForm)">
  <fieldset ngModelGroup="contact">
    <legend>Contact</legend>

    <label>
      First Name <input type="text" name="firstname" ngModel>
    </label>
    <label>
      Last Name <input type="text" name="lastname" ngModel>
    </label>
    <label>
      Email <input type="email" name="email" ngModel>
    </label>
    <label>
      Phone <input type="text" name="phone" ngModel>
    </label>
  </fieldset>

  <fieldset ngModelGroup="address">
    <!-- ... -->
  </fieldset>

  <fieldset ngModelGroup="paymentCard">
    <!-- ... -->
  </fieldset>
</form>
```

- Using the alternative HTML5 labeling format; IDs have no bearing on the `ngForm` / `ngModel` paradigm
- Aside from semantic purposes, `ngModelGroup` does not have to be used on `<fieldset>` — it would work just as well on a `<div>`.

If we were to fill out the form, it would end up in the shape we need for our API, and we can still rely on the HTML field validation if we know it's available.

Using Template Model Binding

One-Way Binding

If you need a form with default values, you can start using the value-binding syntax for `ngModel`.

app/signup-form.component.html

```
<form #signupForm="ngForm" (ngSubmit)=register(signupForm)>

  <label for="username">Username</label>
  <input type="text" name="username" id="username" [ngModel]="generatedUser">

  <label for="email">Email</label>
  <input type="email" name="email" id="email" ngModel>

  <button type="submit">Sign Up</button>
</form>
```

app/signup-form.component.ts

```
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';
// ...

@Component({ /* ... */ })
export class SignupForm {
  generatedUser: string = generateUniqueUserID();

  register(form: NgForm) {
    console.log(form.value);
    // ...
  }
}
```

Two-Way Binding

While Angular 2 assumes one-way binding by default, two-way binding is still available if you need it.

In order to have access to two-way binding in template-driven forms, use the “Banana-Box” syntax (`[(ngModel)]="propertyName"`).

Be sure to declare all of the properties you will need on the component.

```
<form #signupForm="ngForm" (ngSubmit)=register(signupForm)>

  <label for="username">Username</label>
  <input type="text" name="username" id="username" [(ngModel)]="username">

  <label for="email">Email</label>
  <input type="email" name="email" id="email" [(ngModel)]="email">

  <button type="submit">Sign Up</button>
</form>
```

```
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';
// ...

@Component({ /* ... */ })
export class SignUpForm {
  username: string = generateUniqueUserID();
  email: string = '';

  register(form: NgForm) {
    console.log(form.value.username);
    console.log(this.username);
    // ...
  }
}
```

Validating Template-Driven Forms

Validation

Using the template-driven approach, form validation is a matter of following HTML5 practices:

```
<!-- a required field -->
<input type="text" required>

<!-- an optional field of a specific length -->
<input type="text" pattern=".{3,8}">

<!-- a non-optional field of specific length -->
<input type="text" pattern=".{3,8}" required>

<!-- alphanumeric field of specific length -->
<input type="text" pattern="[A-Za-z0-9]{0,5}">
```

Note that the `pattern` attribute is a less-powerful version of JavaScript's RegEx syntax.

There are other HTML5 attributes which can be learned and applied to various types of input; however in most cases they act as upper and lower limits, preventing extra information from being added or removed.

```
<!-- a field which will accept no more than 5 characters -->
<input type="text" maxlength="5">
```

You can use one or both of these methods when writing a template-driven form. Focus on the user experience: in some cases, it makes sense to prevent accidental entry, and in others it makes sense to allow unrestricted entry but provide something like a counter to show limitations.

Model-Driven Forms

Table of Content

- [FormBuilder](#)
- [Validating Model-Driven Forms](#)
- [FormBuilder Custom Validation](#)

While template-driven forms in Angular 2 give us a way to create sophisticated forms while writing as little JavaScript as possible, model-driven forms enable us to test our forms without being required to rely on end-to-end tests. That's why model-driven forms are created imperatively, so they end up as actually properties on our components, which makes them easier to test.

In addition, it's important to understand that when building reactive/model-driven forms, Angular 2 doesn't magically create the templates for us. So it's not that we just create the form model and then all of a sudden we have DOM generated in our app that renders the form. Reactive forms are more like an addition to template-driven forms, although, depending on what we're doing some things can be left out here and there (e.g. validators on DOM elements etc.).

Visual Cues for Users

HTML5 provides `:invalid` and `:valid` pseudo-selectors for its input fields.

```
input[type="text"]::valid {  
    border: 2px solid green;  
}  
  
input[type="text"]::invalid {  
    border: 2px solid red;  
}
```

Unfortunately, this system is rather unsophisticated and would require more manual effort in order to work with complex forms or user behavior.

Rather than writing extra code, and creating and enforcing your own CSS classes, to manage these behaviors, Angular 2 provides you with several classes, already accessible on your inputs.

```
/* field value is valid */  
.ng-valid {}  
  
/* field value is invalid */  
.ng-invalid {}  
  
/* field has not been clicked in, tapped on, or tabbed over */  
.ng-untouched {}  
  
/* field has been previously entered */  
.ng-touched {}  
  
/* field value is unchanged from the default value */  
.ng-pristine {}  
  
/* field value has been modified from the default */  
.ng-dirty {}
```

Note the three pairs:

- valid / invalid
- untouched / touched
- pristine / dirty

These pairs can be used in many combinations in your CSS to change style based on the three separate flags they represent. Angular will toggle between the pairs on each input as the state of the input changes.

```
/* field has been unvisited and unchanged */
input.ng-untouched.ng-pristine {}

/* field has been previously visited, and is invalid */
input.ng-touched.ng-invalid {}
```

`.ng-untouched` will not be replaced by `.ng-touched` until the user *leaves* the input for the first time

For templating purposes, Angular also gives you access to the unprefixed properties on the input, in both code and template:

```
<input name="myInput" [FormControl]="myCustomInput">
<div [hidden]="myCustomInput.pristine">I've been changed</div>
```

Routing

In this section we will discuss the role of routing in Single Page Applications and Angular 2's new component router.

Why Routing?

Routing allows us to express some aspects of the application's state in the URL. Unlike with server-side front-end solutions, this is optional - we can build the full application without ever changing the URL. Adding routing, however, allows the user to go straight into certain aspects of the application. This is very convenient as it can keep your application linkable and bookmarkable and allow users to share links with others.

Routing allows you to:

- Maintain the state of the application
- Implement modular applications
- Implement the application based on the roles (certain roles have access to certain URLs)

Configuring Routes

Base URL Tag

The Base URL tag must be set within the `<head>` tag of index.html:

```
<base href="/">
```

In the demos we use a script tag to set the base tag. In a real application it must be set as above.

Route Definition Object

The `Routes` type is an array of routes that defines the routing for the application. This is where we can set up the expected paths, the components we want to use and what we want our application to understand them as.

Each route can have different attributes; some of the common attributes are:

- `path` - URL to be shown in the browser when application is on the specific route
- `component` - component to be rendered when the application is on the specific route
- `redirectTo` - redirect route if needed; each route can have either component or redirect attribute defined in the route (covered later in this chapter)
- `pathMatch` - optional property that defaults to 'prefix'; determines whether to match full URLs or just the beginning. When defining a route with empty path string set pathMatch to 'full', otherwise it will match all paths.
- `children` - array of route definitions objects representing the child routes of this route (covered later in this chapter).

To use `Routes`, create an array of [route configurations](#).

Below is the sample `Routes` array definition:

```
const routes: Routes = [
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two', component: ComponentTwo }
];
```

[See Routes definition](#)

RouterModule

`RouterModule.forRoot` takes the `Routes` array as an argument and returns a *configured* router module. This router module must be specified in the list of imports of the app module.

```
...
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two', component: ComponentTwo }
];

export const routing = RouterModule.forRoot(routes);

@NgModule({
  imports: [
    BrowserModule,
    routing
  ],
  declarations: [
    AppComponent,
    ComponentOne,
    ComponentTwo
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Redirecting the Router to Another Route

When your application starts, it navigates to the empty route by default. We can configure the router to redirect to a named route by default:

```
export const routes: Routes = [
  { path: '', redirectTo: 'component-one', pathMatch: 'full' },
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two', component: ComponentTwo }
];
```

This tells the router to redirect to `component-one` when matching the empty path ("").

When starting the application, it will automatically navigate to the route for `component-one`.

Defining Links Between Routes

RouterLink

Add links to routes using the `RouterLink` directive.

For example the following code defines a link to the route at path `component-one`.

```
<a [routerLink]="/component-one">Component One</a>
```

Navigating Programmatically

Alternatively, you can navigate to a route by calling the `navigate` function on the router:

```
this.router.navigate(['/component-one']);
```

Dynamically Adding Route Components

Rather than define each route's component separately, use `<RouterOutlet>` which serves as a component placeholder; Angular 2 dynamically adds the component for the route being activated into the `<router-outlet></router-outlet>` element.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app',
  template: `
    <nav>
      <a [routerLink]="/component-one">Component One</a>
      <a [routerLink]="/component-two">Component Two</a>
    </nav>

    <router-outlet></router-outlet>
    <!-- Route components are added by router here -->
  `
})
export class AppComponent {}
```

In the above example, the component corresponding to the route specified will be placed after the `<router-outlet></router-outlet>` element when the link is clicked.

[View Example](#)

View examples running in full screen mode to see route changes in the URL.

Using Route Parameters

Say we are creating an application that displays a product list. When the user clicks on a product in the list, we want to display a page showing the detailed information about that product. To do this you must:

- add a route parameter ID
- link the route to the parameter
- add the service that reads the parameter.

Declaring Route Parameters

The route for the component that displays the details for a specific product would need a route parameter for the ID of that product. We could implement this using the following

Routes :

```
export const routes: Routes = [
  { path: '', redirectTo: 'product-list', pathMatch: 'full' },
  { path: 'product-list', component: ProductList },
  { path: 'product-details/:id', component: ProductDetails }
];
```

Note `:id` in the path of the `product-details` route, which places the parameter in the path. For example, to see the product details page for product with ID 5, you must use the following URL: `localhost:3000/product-details/5`

Linking to Routes with Parameters

In the `ProductList` component you could display a list of products. Each product would have a link to the `product-details` route, passing the ID of the product:

```
<a *ngFor="let product of products"
  [routerLink]=["'/product-details', product.id]>
</a>
```

Note that the `routerLink` directive passes an array which specifies the path and the route parameter. Alternatively we could navigate to the route programmatically:

```
goToProductDetails(id) {
  this.router.navigate(['/product-details', id]);
}
```

Reading Route Parameters

The `ProductDetails` component must read the parameter, then load the product based on the ID given in the parameter.

The `ActivatedRoute` service provides a `params` Observable which we can subscribe to to get the route parameters (see [Observables](#)).

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'product-details',
  template: `
    <div>
      Showing product details for product:
    </div>
  `,
})
export class LoanDetailsPage implements OnInit, OnDestroy {
  id: number;
  private sub: any;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.sub = this.route.params.subscribe(params => {
      this.id = +params['id']; // (+) converts string 'id' to a number

      // In a real app: dispatch action to load the details here.
    });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }
}
```

The reason that the `params` property on `ActivatedRoute` is an Observable is that the router may not recreate the component when navigating to the same component. In this case the parameter may change without the component being recreated.

[View Basic Example](#)

[View Example with Programmatic Route Navigation](#)

View examples running in full screen mode to see route changes in the URL.

Defining Child Routes

When some routes may only be accessible and viewed within other routes it may be appropriate to create them as child routes.

For example: The product details page may have a tabbed navigation section that shows the product overview by default. When the user clicks the "Technical Specs" tab the section shows the specs instead.

If the user clicks on the product with ID 3, we want to show the product details page with the overview:

```
localhost:3000/product-details/3/overview
```

When the user clicks "Technical Specs":

```
localhost:3000/product-details/3/specs
```

`overview` and `specs` are child routes of `product-details/:id`. They are only reachable within product details.

Our `Routes` with children would look like:

```
export const routes: Routes = [
  { path: '', redirectTo: 'product-list', pathMatch: 'full' },
  { path: 'product-list', component: ProductList },
  { path: 'product-details/:id', component: ProductDetails,
    children: [
      { path: '', redirectTo: 'overview', pathMatch: 'full' },
      { path: 'overview', component: Overview },
      { path: 'specs', component: Specs }
    ]
  }
];
```

Where would the components for these child routes be displayed? Just like we had a `<router-outlet></router-outlet>` for the root application component, we would have a router outlet inside the `ProductDetails` component. The components corresponding to the child routes of `product-details` would be placed in the router outlet in `ProductDetails`.

```

import { Component, OnInit, OnDestroy } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'product-details',
  template: `
    <p>Product Details: {{id}}</p>
    <!-- Product information -->
    <nav>
      <a [routerLink]=["overview"]>Overview</a>
      <a [routerLink]=["specs"]>Technical Specs</a>
    </nav>
    <router-outlet></router-outlet>
    <!-- Overview & Specs components get added here by the router -->
  `
})
export default class ProductDetails implements OnInit, OnDestroy {
  id: number;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.sub = this.route.params.subscribe(params => {
      this.id = +params['id']; // (+) converts string 'id' to a number
    });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }
}

```

Alternatively, we could specify `overview` route URL simply as:

```
localhost:3000/product-details/3
```

```

export const routes: Routes = [
  { path: '', redirectTo: 'product-list', pathMatch: 'full' },
  { path: 'product-list', component: ProductList },
  { path: 'product-details/:id', component: ProductDetails,
    children: [
      { path: '', component: Overview },
      { path: 'specs', component: Specs }
    ]
  }
];

```

Since the `overview` child route of `product-details` has an empty path, it will be loaded by default. The `specs` child route remains the same.

[View Example with child routes](#)[View Example with route params & child routes](#)

View examples running in full screen mode to see route changes in the URL.

Accessing a Parent's Route Parameters

In the above example, say that the child routes of `product-details` needed the ID of the product to fetch the spec or overview information. The child route component can access the parent route's parameters as follows:

```
export default class Overview {
  parentRouteId: number;
  private sub: any;

  constructor(private router: Router,
    private route: ActivatedRoute) {}

  ngOnInit() {
    // Get parent ActivatedRoute of this route.
    this.sub = this.router.routerState.parent(this.route)
      .params.subscribe(params => {
        this.parentRouteId = +params["id"];
      });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }
}
```

[View Example child routes accessing parent's route parameters](#)

View examples running in full screen mode to see route changes in the URL.

Links

Routes can be prepended with `/`, or `../`; this tells Angular 2 where in the route tree to link to.

Prefix	Looks in
/	Root of the application
none	Current component children routes
.../	Current component parent routes

Example:

```
<a [routerLink]=["route-one"]>Route One</a>
<a [routerLink]=["../route-two"]>Route Two</a>
<a [routerLink]=["/route-three"]>Route Three</a>
```

In the above example, the link for route one links to a child of the current route. The link for route two links to a sibling of the current route. The link for route three links to a child of the root component (same as route one link if current route is root component).

[View Example with linking throughout route tree](#)

View examples running in full screen mode to see route changes in the URL.

Controlling Access to or from a Route

To control whether the user can navigate to or away from a given route, use route guards.

For example, we may want some routes to only be accessible once the user has logged in or accepted Terms & Conditions. We can use route guards to check these conditions and control access to routes.

Route guards can also control whether a user can leave a certain route. For example, say the user has typed information into a form on the page, but has not submitted the form. If they were to leave the page, they would lose the information. We may want to prompt the user if the user attempts to leave the route without submitting or saving the information.

Registering the Route Guards with Routes

In order to use route guards, we must register them with the specific routes we want them to run for.

For example, say we have an `accounts` route that only users that are logged in can navigate to. This page also has forms and we want to make sure the user has submitted unsaved changes before leaving the accounts page.

In our route config we can add our guards to that route:

```
import { Routes, RouterModule } from '@angular/router';
import { AccountPage } from './account-page';
import { LoginRouteGuard } from './login-route-guard';
import { SaveFormsGuard } from './save-forms-guard';

const routes: Routes = [
  { path: 'home', component: HomePage },
  {
    path: 'accounts',
    component: AccountPage,
    canActivate: [LoginRouteGuard],
    canDeactivate: [SaveFormsGuard]
  }
];

export const appRoutingProviders: any[] = [];

export const routing = RouterModule.forRoot(routes);
```

Now `LoginRouteGuard` will be checked by the router when activating the `accounts` route, and `SaveFormsGuard` will be checked when leaving that route.

Implementing CanActivate

Let's look at an example activate guard that checks whether the user is logged in:

```
import { CanActivate } from '@angular/router';
import { Injectable } from '@angular/core';
import { LoginService } from './login-service';

@Injectable()
export class LoginRouteGuard implements CanActivate {

  constructor(private loginService: LoginService) {}

  canActivate() {
    return this.loginService.isLoggedIn();
  }
}
```

This class implements the `canActivate` interface by implementing the `canActivate` function.

When `canActivate` returns true, the user can activate the route. When `canActivate` returns false, the user cannot access the route. In the above example, we allow access when the user is logged in.

`canActivate` can also be used to notify the user that they can't access that part of the application, or redirect them to the login page.

[See Official Definition for CanActivate](#)

Implementing CanDeactivate

`CanDeactivate` works in a similar way to `CanActivate` but there are some important differences. The `canDeactivate` function passes the component being deactivated as an argument to the function:

```
export interface CanDeactivate<T> {
  canDeactivate(component: T, route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean;
}
```

We can use that component to determine whether the user can deactivate.

```
import { CanDeactivate } from '@angular/router';
import { Injectable } from '@angular/core';
import { AccountPage } from './account-page';

@Injectable()
export class SaveFormsGuard implements CanDeactivate<AccountPage> {

  canDeactivate(component: AccountPage) {
    return component.areFormsSaved();
  }

}
```

[See Official Definition for CanDeactivate](#)

Async Route Guards

The `canActivate` and `canDeactivate` functions can either return values of type `boolean`, or `Observable<boolean>` (an Observable that resolves to `boolean`). If you need to do an asynchronous request (like a server request) to determine whether the user can navigate to or away from the route, you can simply return an `Observable<boolean>`. The router will wait until it is resolved and use that value to determine access.

For example, when the user navigates away you could have a dialog service ask the user to confirm the navigation. The dialog service returns an `Observable<boolean>` which resolves to true if the user clicks 'OK', or false if user clicks 'Cancel'.

```
canDeactivate() {
  return dialogService.confirm('Discard unsaved changes?');
}
```

[View Example](#)

[See Official Documentation for Route Guards](#)

Passing Optional Parameters

Query parameters allow you to pass optional parameters to a route such as pagination information.

For example, on a route with a paginated list, the URL might look like the following to indicate that we've loaded the second page:

```
localhost:3000/product-list?page=2
```

The key difference between query parameters and [route parameters](#) is that route parameters are essential to determining route, whereas query parameters are optional.

Passing Query Parameters

Use the `[queryParams]` directive along with `[routerLink]` to pass query parameters. For example:

```
<a [routerLink]=["/product-list"] [queryParams]="{ page: 99 }">Go to Page 99</a>
```

Alternatively, we can navigate programmatically using the `Router` service:

```
goToPage(pageNum) {
  this.router.navigate(['/product-list'], { queryParams: { page: pageNum } });
}
```

Reading Query Parameters

Similar to reading [route parameters](#), the `Router` service returns an [Observable](#) we can subscribe to to read the query parameters:

```
import { Component } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';

@Component({
  selector: 'product-list',
  template: `<!-- Show product list --&gt;`
})
export default class ProductList {
  constructor(
    private route: ActivatedRoute,
    private router: Router) {}

  ngOnInit() {
    this.sub = this.route
      .queryParams
      .subscribe(params =&gt; {
        // Defaults to 0 if no query param provided.
        this.page = +params['page'] || 0;
      });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }

  nextPage() {
    this.router.navigate(['product-list'], { queryParams: { page: this.page + 1 } });
  }
}</pre>
```

[View Example](#)

[See Official Documentation on Query Parameters](#)

Using Auxiliary Routes

Angular 2 supports the concept of auxiliary routes, which allow you to set up and navigate multiple independent routes in a single app. Each component has one primary route and zero or more auxiliary outlets. Auxiliary outlets must have unique name within a component.

To define the auxiliary route we must first add a named router outlet where contents for the auxiliary route are to be rendered.

Here's an example:

```
import {Component} from '@angular/core';

@Component({
  selector: 'app',
  template: `
    <nav>
      <a [routerLink]="/component-one">Component One</a>
      <a [routerLink]="/component-two">Component Two</a>
      <a [routerLink]="[{ outlets: { 'sidebar': [ 'component-aux' ] } }]">Component Aux</a>
    </nav>

    <div style="color: green; margin-top: 1rem;">Outlet:</div>
    <div style="border: 2px solid green; padding: 1rem;">
      <router-outlet></router-outlet>
    </div>

    <div style="color: green; margin-top: 1rem;">Sidebar Outlet:</div>
    <div style="border: 2px solid blue; padding: 1rem;">
      <router-outlet name="sidebar"></router-outlet>
    </div>
  `
})
export class AppComponent {

}
```

Next we must define the link to the auxiliary route for the application to navigate and render the contents.

```
<a [routerLink]="[{ outlets: { 'sidebar': [ 'component-aux' ] } }]">
  Component Aux
</a>
```

[View Example](#)

Each auxiliary route is an independent route which can have:

- its own child routes
- its own auxiliary routes
- its own route-params
- its own history stack

Change Detection

Change detection is the process that allows Angular to keep our views in sync with our models.

Change detection has changed in a big way between the old version of Angular and the new one. In Angular 1, the framework kept a long list of watchers (one for every property bound to our templates) that needed to be checked every-time a digest cycle was started. This was called *dirty checking* and it was the only change detection mechanism available.

Because by default Angular 1 implemented two way data binding, the flow of changes was pretty much chaotic, models were able to change directives, directives were able to change models, directives were able to change other directives and models were able to change other models.

In Angular 2, **the flow of information is unidirectional**, even when using `ngModel` to implement two way data binding, which is only syntactic sugar on top of the unidirectional flow. In this new version of the framework, our code is responsible for updating the models. Angular is only responsible for reflecting those changes in the components and the DOM by means of the selected change detection strategy.

How Change Detection Works

Let's see how change detection works with a simple example.

We are going to create a simple `MovieApp` to show information about one movie. This app is going to consist of only two components: the `MovieComponent` that shows information about the movie and the `MainComponent` which holds a reference to the movie with buttons to perform some actions.

Our `MainComponent` will have three properties: the `slogan` of the app, the `title` of the movie and the lead `actor`. The last two properties will be passed to the `MovieComponent` element referenced in the template.

app/main.component.ts

```
import {Component} from '@angular/core';
import {MovieComponent} from './movie.component';
import {Actor} from './actor.model';

@Component({
  selector: 'main',
  template: `
    <h1>MovieApp</h1>
    <p>{{ slogan }}</p>
    <button type="button" (click)="changeActorProperties()">Change Actor Properties</button>
    <button type="button" (click)="changeActorObject()">Change Actor Object</button>
    <movie [title]="title" [actor]="actor"></movie>`
})
export class MainComponent {
  slogan: string = 'Just movie information';
  title: string = 'Terminator 1';
  actor: Actor = new Actor('Arnold', 'Schwarzenegger');

  changeActorProperties() {
    this.actor.firstName = 'Nicholas';
    this.actor.lastName = 'Cage';
  }

  changeActorObject() {
    this.actor = new Actor('Bruce', 'Willis');
  }
}
```

In the above code snippet, we can see that our component defines two buttons that trigger different methods. The `changeActorProperties` will update the lead actor of the movie by directly changing the properties of the `actor` object. In contrast, the method `changeActorObject` will change the information of the actor by creating a completely new instance of the `Actor` class.

The `Actor` model is pretty straightforward, it is just a class that defines the `firstName` and the `lastName` of an actor.

app/actor.model.ts

```
export class Actor {
  constructor(
    public firstName: string,
    public lastName: string) {}
}
```

Finally, the `MovieComponent` shows the information provided by the `MainComponent` in its template.

app/movie.component.ts

```
import {Component, Input} from '@angular/core';
import {Actor} from './actor.model';

@Component({
  selector: 'movie',
  styles: ['div {border: 1px solid black}'],
  template: `
    <div>
      <h3>{{ title }}</h3>
      <p>
        <label>Actor:</label>
        <span>{{actor.firstName}} {{actor.lastName}}</span>
      </p>
    </div>`
})
export class MovieComponent {
  @Input() title: string;
  @Input() actor: Actor;
}
```

The final result of the app is shown in the screenshot below:



Change Detector Classes

Table of Content

- Change Detection Strategy: OnPush

At runtime, Angular 2 will create special classes that are called *change detectors*, one for every component that we have defined. In this case, Angular will create two classes:

`MainComponent_ChangeDetector` and `MovieComponent_ChangeDetector`.

The goal of the change detectors is to know which model properties used in the template of a component have changed since the last time the change detection process ran.

In order to know that, Angular creates an instance of the appropriate change detector class and a link to the component that it's supposed to check.

In our example, because we only have one instance of the `MainComponent` and the `MovieComponent`, we will have only one instance of the `MainComponent_ChangeDetector` and the `MovieComponent_ChangeDetector`.

The code snippet below is a conceptual model of how the `MainComponent_ChangeDetector` class might look.

```

class MainComponent_ChangeDetector {

  constructor(
    public previousSlogan: string,
    public previousTitle: string,
    public previousActor: Actor,
    public movieComponent: MovieComponent
  ) {}

  detectChanges(slogan: string, title: string, actor: Actor) {
    if (slogan !== this.previousSlogan) {
      this.previousSlogan = slogan;
      this.movieComponent.slogan = slogan;
    }
    if (title !== this.previousTitle) {
      this.previousTitle = title;
      this.movieComponent.title = title;
    }
    if (actor !== this.previousActor) {
      this.previousActor = actor;
      this.movieComponent.actor = actor;
    }
  }
}

```

Because in the template of our `MainComponent` we reference three variables (`slogan`, `title` and `actor`), our change detector will have three properties to store the "old" values of these three properties, plus a reference to the `MainComponent` instance that it's supposed to "watch". When the change detection process wants to know if our `MainComponent` instance has changed, it will run the method `detectChanges` passing the current model values to compare with the old ones. If a change was detected, the component gets updated.

Disclaimer: This is just a conceptual overview of how change detector classes work; the actual implementation may be different.

Change Detection Strategy: Default

By default, Angular defines a certain change detection strategy for every component in our application. To make this definition explicit, we can use the property `changeDetection` of the `@Component` decorator.

`app/movie.component.ts`

```
// ...
import {ChangeDetectionStrategy} from '@angular/core';

@Component({
  // ...
  changeDetection: ChangeDetectionStrategy.Default
})
export class MovieComponent {
  // ...
}
```

[View Example](#)

Let's see what happens when a user clicks the button "Change Actor Properties" when using the `Default` strategy.

As noted previously, changes are triggered by events and the propagation of changes is done in two phases: the application phase and the change detection phase.

Phase 1 (Application):

In the first phase, the application (our code) is responsible for updating the models in response to some event. In this scenario, the properties `actor.firstName` and `actor.lastName` are updated.

Phase 2 (Change Detection):

Now that our models are updated, Angular must update the templates using change detection.

Change detection always starts at the root component, in this case the `MainComponent`, and checks if any of the model properties bound to its template have changed, comparing the old value of each property (before the event was triggered) to the new one (after the models were updated). The `MainComponent` template has a reference to three properties, `slogan`, `title` and `actor`, so the comparison made by its corresponding change detector will look like:

- Is `slogan !== previousSlogan`? No, it's the same.
- Is `title !== previousTitle`? No, it's the same.
- Is `actor !== previousActor`? No, it's the same.

Notice that even if we change the properties of the `actor` object, we are always working with the same instance. Because we are doing a shallow comparison, the result of asking if `actor !== previousActor` will always be `false` even when its internal property values have

indeed changed. Even though the change detector was unable to find any change, the **default strategy** for the change detection is **to traverse all the components of the tree** even if they do not seem to have been modified.

Next, change detection moves down in the component hierarchy and check the properties bound to the `MovieComponent`'s template doing a similar comparison:

- Is `title` `!== previousTitle` ? No, it's the same.
- Is `actorFirstName` `!== previousActorFirstName` ? Yes, it has changed.
- Is `actorLastName` `!== previousActorLastName` ? Yes, it has changed.

Finally, Angular has detected that some of the properties bound to the template have changed so it will update the DOM to get the view in sync with the model.

Performance Impact

Traversing all the tree components to check for changes could be costly. Imagine that instead of just having one reference to `<movie>` inside our `MainComponent`'s template, we have multiple references?

```
<movie *ngFor="let movie of movies" [title]="movie.title" [actor]="movie.actor"></movie>
```

If our movie list grows too big, the performance of our system will start degrading. We can narrow the problem to one particular comparison:

- Is `actor` `!== previousActor` ?

As we have learned, this result is not very useful because we could have changed the properties of the object without changing the instance, and the result of the comparison will always be `false`. Because of this, change detection is going to have to check every child component to see if any of the properties of that object (`firstName` or `lastName`) have changed.

What if we can find a way to indicate to the change detection that our `MovieComponent` depends only on its inputs and that these inputs are immutable? In short, we are trying to guarantee that when we change any of the properties of the `actor` object, we end up with a different `Actor` instance so the comparison `actor` `!== previousActor` will always return `true`. On the other hand, if we did not change any property, we are not going to create a new instance, so the same comparison is going to return `false`.

If the above condition can be guaranteed (create a new object every time any of its properties changes, otherwise we keep the same object) and when checking the inputs of the `MovieComponent` has this result:

- Is `title !== previousTitle` ? No, it's the same.
- Is `actor !== previousActor` ? No, it's the same.

then we can skip the internal check of the component's template because we are now certain that nothing has changed internally and there's no need to update the DOM. This will improve the performance of the change detection system because fewer comparisons have to be made to propagate changes through the app.

Change Detection Strategy: OnPush

To inform Angular that we are going to comply with the conditions mentioned before to improve performance, we will use the `OnPush` change detection strategy on the `MovieComponent`.

`app/movie.component.ts`

```
// ...  
  
@Component({  
    // ...  
    changeDetection: ChangeDetectionStrategy.OnPush  
})  
export class MovieComponent {  
    // ...  
}
```

[View Example](#)

This will inform Angular that our component only depends on its inputs and that any object that is passed to it should be considered immutable. This time when we click the "Change Actor Properties" button nothing changes in the view.

Let's follow the logic behind it again. When the user clicks the button, the method `changeActorProperties` is called and the properties of the `actor` object get updated.

When the change detection analyzes the properties bound to the `MainComponent`'s template, it will see the same picture as before:

- Is `slogan !== previousSlogan` No, it's the same.
- Is `title !== previousTitle` ? No, it's the same.
- Is `actor !== previousActor` ? No, it's the same.

But this time, we explicitly told Angular that our component only depends on its inputs and all of them are immutable. Angular then assumes that the `MovieComponent` hasn't changed and will skip the check for that component. Because we didn't force the `actor` object to be immutable, we end up with our model out of sync with the view.

Let's rerun the app but this time we will click the "Change Actor Object" button. This time, we are creating a new instance of the `Actor` class and assigning it to the `this.actor` object. When change detection analyzes the properties bound to the `MainComponent`'s template it will find:

- Is `slogan !== previousSlogan` No, it's the same.
- Is `title !== previousTitle` ? No, it's the same.
- Is `actor !== previousActor` ? Yes, it has changed.

Because change detection now knows that the `actor` object changed (it's a new instance) it will go ahead and continue checking the template for `MovieComponent` to update its view. At the end, our templates and models are in sync.

Enforcing Immutability

We cheated a little in the previous example. We told Angular that all of our inputs, including the `actor` object, were immutable objects, but we went ahead and updated its properties, violating the immutability principle. As a result we ended up with a sync problem between our models and our views. One way to enforce immutability is using the library [Immutable.js](#).

Because in JavaScript primitive types like `string` and `number` are immutable by definition, we should only take care of the objects we are using. In this case, the `actor` object.

Here's an example comparing a mutable type like an `array` to an immutable type like a `string`:

```
var b = ['C', 'a', 'r'];
b[0] = 'B';
console.log(b) // ['B', 'a', 'r'] => The first letter changed, arrays are mutable

var a = 'Car';
a[0] = 'B';
console.log(a); // 'Car' => The first letter didn't change, strings are immutable
```

First we need to install the `immutable.js` library using the command:

```
npm install --save immutable
```

Then in our `MainComponent` we import the library and use it to create an actor object as an immutable.

app/main.component.ts

```

import {Component} from '@angular/core';
import {MovieComponent} from './movie.component';
import * as Immutable from 'immutable';

@Component({
  selector: 'main',
  template: `
    <h1>MovieApp</h1>
    <p>{{ slogan }}</p>
    <button type="button" (click)="changeActor()">Change Actor</button>
    <movie [title]="title" [actor]="actor"></movie>`
})
export class MainComponent {
  slogan: string = 'Just movie information';
  title: string = 'Terminator 1';
  actor: Immutable.Map<string, string> = Immutable.Map({firstName: 'Arnold', lastName: 'Schwarzenegger'});

  changeActor() {
    this.actor = this.actor.merge({firstName: 'Nicholas', lastName: 'Cage'});
  }
}

```

Now, instead of creating an instance of an `Actor` class, we are defining an immutable object using `Immutable.Map`. Because `this.actor` is now an immutable object, we cannot change its internal properties (`firstName` and `lastName`) directly. What we can do however is create another object based on `actor` that has different values for both fields - that is exactly what the `merge` method does.

Because we are always getting a new object when we try to change the `actor`, there's no point in having two different methods in our component. We removed the methods

`changeActorProperties` and `changeActorObject` and created a new one called `changeActor`.

Additional changes have to be made to the `MovieComponent` as well. First we need to declare the `actor` object as an immutable type, and in the template, instead of trying to access the object properties directly using a syntax like `actor.firstName`, we need to use the `get` method of the immutable.

`app/movie.component.ts`

```
import {Component, Input} from '@angular/core';
import {ChangeDetectionStrategy} from '@angular/core';
import * as Immutable from 'immutable';

@Component({
  selector: 'movie',
  styles: ['div {border: 1px solid black}'],
  template: `
    <div>
      <h3>{{ title }}</h3>
      <p>
        <label>Actor:</label>
        <span>{{actor.get('firstName')}} {{actor.get('lastName')}}</span>
      </p>
    </div>`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class MovieComponent {
  @Input() title: string;
  @Input() actor: Immutable.Map<string, string>;
}
```

[View Example](#)

Using this pattern we are taking full advantage of the "OnPush" change detection strategy and thus reducing the amount of work done by Angular to propagate changes and to get models and views in sync. This improves the performance of the application.

Additional Resources

To learn more about change detection, visit the following links (in order of relevance):

- [NgConf 2014: Change Detection \(Video\)](#)
- [Angular API Docs: ChangeDetectionStrategy](#)
- [Victor Savkin Blog: Change Detection in Angular 2](#)
- [Victor Savkin Blog: Two Phases of Angular 2 Applications](#)
- [Victor Savkin Blog: Angular, Immutability and Encapsulation](#)

Angular 2 Dependency Injection

Dependency Injection (DI) was a core feature in Angular 1.x, and that has not changed in Angular 2. DI is a programming concept that predates Angular. The purpose of DI is to simplify dependency management in software components. By reducing the amount of information a component needs to know about its dependencies, unit testing can be made easier and code is more likely to be flexible.

Angular 2 improves on Angular 1.x's DI model by unifying Angular 1.x's two injection systems. Tooling issues with respect to static analysis, minification and namespace collisions have also been fixed in Angular 2.

What is DI?

So dependency injection makes programmers' lives easier, but what does it *really* do?

Consider the following code:

```
class Hamburger {  
    private bun: Bun;  
    private patty: Patty;  
    private toppings: Toppings;  
    constructor() {  
        this.bun = new Bun('withSesameSeeds');  
        this.patty = new Patty('beef');  
        this.toppings = new Toppings(['lettuce', 'pickle', 'tomato']);  
    }  
}
```

The above code is a contrived class that represents a hamburger. The class assumes a `Hamburger` consists of a `Bun`, `Patty` and `Toppings`. The class is also responsible for *making* the `Bun`, `Patty` and `Toppings`. This is a bad thing. What if a vegetarian burger were needed? One naive approach might be:

```
class VeggieHamburger {  
    private bun: Bun;  
    private patty: Patty;  
    private toppings: Toppings;  
    constructor() {  
        this.bun = new Bun('withSesameSeeds');  
        this.patty = new Patty('tofu');  
        this.toppings = new Toppings(['lettuce', 'pickle', 'tomato']);  
    }  
}
```

There, problem solved right? But what if we need a gluten free hamburger? What if we want different toppings... maybe something more generic like:

```
class Hamburger {
    private bun: Bun;
    private patty: Patty;
    private toppings: Toppings;
    constructor(bunType: string, pattyType: string, toppings: string[]) {
        this.bun = new Bun(bunType);
        this.patty = new Patty(pattyType);
        this.toppings = new Toppings(toppings);
    }
}
```

Okay this is a little different, and it's more flexible in some ways, but it is still quite brittle.

What would happen if the `Patty` constructor changed to allow for new features? The whole `Hamburger` class would have to be updated. In fact, any time any of these constructors used in `Hamburger`'s constructor are changed, `Hamburger` would also have to be changed.

Also, what happens during testing? How can `Bun`, `Patty` and `Toppings` be effectively mocked?

Taking those concerns into consideration, the class could be rewritten as:

```
class Hamburger {
    private bun: Bun;
    private patty: Patty;
    private toppings: Toppings;
    constructor(bun: Bun, patty: Patty, toppings: Toppings) {
        this.bun = bun;
        this.patty = patty;
        this.toppings = toppings;
    }
}
```

Now when `Hamburger` is instantiated it does not need to know anything about its `Bun`, `Patty`, or `Toppings`. The construction of these elements has been moved out of the class. This pattern is so common that TypeScript allows it to be written in shorthand like so:

```
class Hamburger {
    constructor(private bun: Bun, private patty: Patty,
        private toppings: Toppings) {}
}
```

The `Hamburger` class is now simpler and easier to test. This model of having the dependencies provided to `Hamburger` is basic dependency injection.

However there is still a problem. How can the instantiation of `Bun`, `Patty` and `Toppings` best be managed?

This is where dependency injection as a *framework* can benefit programmers, and it is what Angular 2 provides with its dependency injection system.

DI Framework

So there's a fancy new `Hamburger` class that is easy to test, but it's currently awkward to work with. Instantiating a `Hamburger` requires:

```
const hamburger = new Hamburger(new Bun(), new Patty('beef'), new Toppings());
```

That's a lot of work to create a `Hamburger`, and now all the different pieces of code that make `Hamburger`s have to understand how `Bun`, `Patty` and `Toppings` get instantiated.

One approach to dealing with this new problem might be to make a factory function like so:

```
function makeHamburger() {
  const bun = new Bun();
  const patty = new Patty('beef');
  const toppings = new Toppings(['lettuce', 'tomato', 'pickles']);
  return new Hamburger(bun, patty, toppings);
}
```

This is an improvement, but when more complex `Hamburger`s need to be created this factory will become confusing. The factory is also responsible for knowing how to create four different components. This is a lot for one function.

This is where a dependency injection framework can help. DI Frameworks have the concept of an `Injector` object. An Injector is a lot like the factory function above, but more general, and powerful. Instead of one giant factory function, an Injector has a factory, or *recipe* (pun intended) for a collection of objects. With an `Injector`, creating a `Hamburger` could be as easy as:

```
const injector = new Injector([Hamburger, Bun, Patty, Toppings]);
const burger = injector.get(Hamburger);
```

Angular 2's DI

Table of Content

- [@Inject\(\) and @Injectable](#)
- [Injection Beyond Classes](#)
- [The Injector Tree](#)

The last example introduced a hypothetical `Injector` object. Angular 2 simplifies DI even further. With Angular 2, programmers almost never have to get bogged down with injection details.

Angular 2's DI system is (mostly) controlled through `@NgModule`'s. Specifically the `providers` array.

For example:

```
import { Injectable, NgModule } from '@angular/core';

@Injectable()
class Hamburger {
  constructor(private bun: Bun, private patty: Patty,
    private toppings: Toppings) {}
}

@NgModule({
  providers: [ Hamburger ],
})
export class DiExample {};
```

In the above example the `DiExample` module is told about the `Hamburger` class.

Another way of saying this is that Angular 2 has been *provided* a `Hamburger`.

That seems pretty straightforward, but astute readers will be wondering how Angular 2 knows how to build `Hamburger`. What if `Hamburger` was a string, or a plain function?

Angular 2 *assumes* that it's being given a class.

What about `Bun`, `Patty` and `Toppings`? How is `Hamburger` getting those?

It's not, at least not yet. Angular 2 does not know about them yet. That can be changed easily enough:

```
import { Injectable, NgModule } from '@angular/core';

@Injectable()
class Hamburger {
  constructor(private bun: Bun, private patty: Patty,
    private toppings: Toppings) {}
}

@Injectable()
class Patty {}

@Injectable()
class Bun {}

@Injectable()
class Toppings {}

@NgModule({
  providers: [ Hamburger, Patty, Bun, Toppings ],
})
```

Okay, this is starting to look a little bit more complete. Although it's still unclear how `Hamburger` is being told about its dependencies. Perhaps that is related to those odd `@Injectable` statements.

@Inject and @Injectable

Statements that look like `@SomeName` are decorators. [Decorators](#) are a proposed extension to JavaScript. In short, decorators let programmers modify and/or tag methods, classes, properties and parameters. There is a lot to decorators. In this section the focus will be on decorators relevant to DI: `@Inject` and `@Injectable`. For more information on Decorators please see [the EcmaScript 6 and TypeScript Features section](#).

@Inject()

`@Inject()` is a *manual* mechanism for letting Angular 2 know that a *parameter* must be injected. It can be used like so:

```
import { Component, Inject } from '@angular/core';
import { Hamburger } from '../services/hamburger';

@Component({
  selector: 'app',
  template: `Bun Type: {{ bunType }}`
})
export class App {
  bunType: string;
  constructor(@Inject(Hamburger) h) {
    this.bunType = h.bun.type;
  }
}
```

In the above we've asked for `h` to be the singleton Angular associates with the `class symbol` `Hamburger` by calling `@Inject(Hamburger)`. It's important to note that we're using `Hamburger` for its typings *and* as a *reference* to its singleton. We are *not* using `Hamburger` to instantiate anything, Angular does that for us behind the scenes.

When using TypeScript, `@Inject` is only needed for injecting *primitives*. TypeScript's types let Angular 2 know what to do in most cases. The above example would be simplified in TypeScript to:

```

import { Component } from '@angular/core';
import { Hamburger } from '../services/hamburger';

@Component({
  selector: 'app',
  template: `Bun Type: {{ bunType }}`
})
export class App {
  bunType: string;
  constructor(h: Hamburger) {
    this.bunType = h.bun.type;
  }
}

```

[View Example](#)

@Injectable()

`@Injectable()` lets Angular 2 know that a *class* can be used with the dependency injector.

`@Injectable()` is not *strictly* required if the class has *other* Angular 2 decorators on it or does not have any dependencies.

What is important is that any class that is going to be injected with Angular 2 *is decorated*. However, best practice is to decorate injectables with `@Injectable()`, as it makes more sense to the reader.

Here's an example of `Hamburger` marked up with `@Injectable`:

```

import { Injectable } from '@angular/core';
import { Bun } from './bun';
import { Patty } from './patty';
import { Toppings } from './toppings';

@Injectable()
export class Hamburger {
  constructor(public bun: Bun, public patty: Patty, public toppings: Toppings) {
  }
}

```

In the above example Angular 2's injector determines what to inject into `Hamburger`'s constructor by using type information. This is possible because these particular dependencies are typed, and are *not primitive* types. In some cases Angular 2's DI needs more information than just types.

Injection Beyond Classes

So far the only types that injection has been used for have been classes, but Angular 2 is not limited to injecting classes. The concept of `providers` was also briefly touched upon.

So far `providers` have been used with Angular 2's `@NgModule` meta in an array. `providers` have also all been class identifiers. Angular 2 lets programmers specify providers with a more verbose "recipe". This is done with by providing Angular 2 an Object literal (`{}`):

```
import { NgModule } from '@angular/core';
import { App } from './containers/app'; // hypothetical app component
import { Hamburger } from './services/hamburger';

@NgModule({
  providers: [ { provide: Hamburger, useClass: Hamburger } ],
})
export class DiExample {};
```

This example is yet another example that `provide` is a class, but it does so with Angular 2's longer format.

This long format is really handy. If the programmer wanted to switch out `Hamburger` implementations, for example to allow for a `DoubleHamburger`, they could do this easily:

```
import { NgModule } from '@angular/core';
import { App } from './containers/app'; // hypothetical app component
import { Hamburger } from './services/hamburger';
import { DoubleHamburger } from './services/double-hamburger';

@NgModule({
  providers: [ { provide: Hamburger, useClass: DoubleHamburger } ],
})
export class DiExample {};
```

The best part of this implementation swap is that the injection system knows how to build `DoubleHamburgers`, and will sort all of that out.

The injector can use more than classes though. `useValue` and `useFactory` are two other examples of `provider` "recipes" that Angular 2 can use. For example:

```
import { NgModule } from '@angular/core';
import { App } from './containers/app'; // hypothetical app component

const randomFactory = () => { return Math.random(); };

@NgModule({
  providers: [ { provide: 'Random', useFactory: randomFactory } ],
})
export class DiExample {};
```

In the hypothetical app component, 'Random' could be injected like:

```
import { Component, Inject, provide } from '@angular/core';
import { Hamburger } from '../services/hamburger';

@Component({
  selector: 'app',
  template: `Random: {{ value }}`
})
export class App {
  value: number;
  constructor(@Inject('Random') r) {
    this.value = r;
  }
}
```

[View Example](#)

One important note is that 'Random' is in quotes, both in the `provide` function and in the consumer. This is because as a factory we have no `Random` identifier anywhere to access.

The above example uses Angular 2's `useFactory` recipe. When Angular 2 is told to `provide` things using `useFactory`, Angular 2 expects the provided value to be a function. Sometimes functions and classes are even more than what's needed. Angular 2 has a "recipe" called `useValue` for these cases that works almost exactly the same:

```
import { NgModule } from '@angular/core';
import { App } from './containers/app'; // hypothetical app component

@NgModule({
  providers: [ { provide: 'Random', useValue: Math.random() } ],
})
export class DiExample {};
```

[View Example](#)

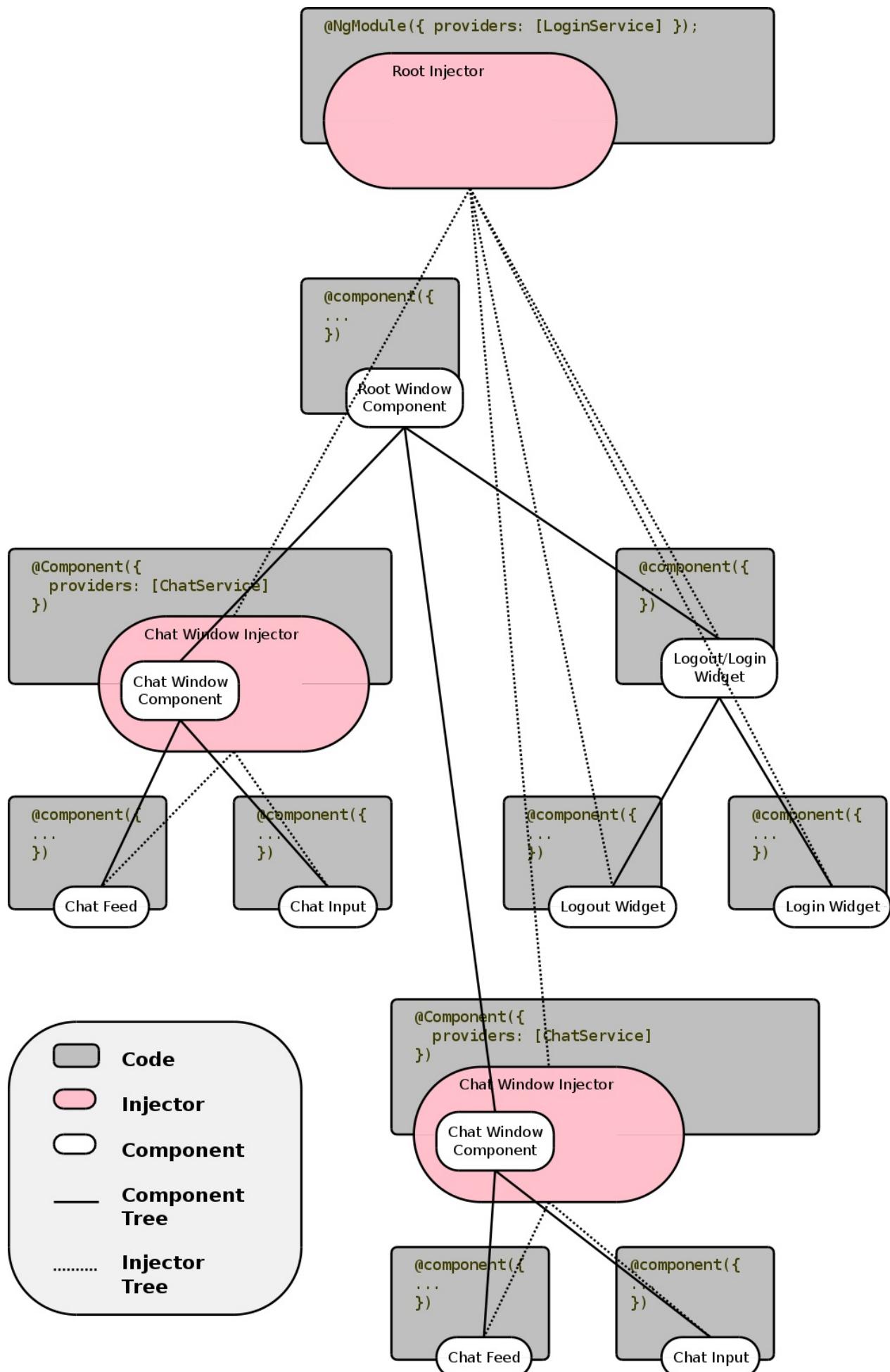
In this case, the product of `Math.random` is assigned to the `useValue` property passed to the `provider`.

The Injector Tree

Angular 2 injectors (generally) return singletons. That is, in the previous example, all components in the application will receive the same random number. In Angular 1.x there was only one injector, and all services were singletons. Angular 2 overcomes this limitation by using a tree of injectors.

In Angular 2 there is not just one injector per application, there is *at least* one injector per application. Injectors are organized in a tree that parallels Angular 2's component tree.

Consider the following tree, which models a chat application consisting of two open chat windows, and a login/logout widget.



In the image above, there is one root injector, which is established through `@NgModule`'s `providers` array. There's a `LoginService` registered with the root injector.

Below the root injector is the root `@Component`. This particular component has no `providers` array and will use the root injector for all of its dependencies.

There are also two child injectors, one for each `ChatWindow` component. Each of these components has their own instantiation of a `chatService`.

There is a third child component, `Logout/Login`, but it has no injector.

There are several grandchild components that have no injectors. There are `chatFeed` and `chatInput` components for each `ChatWindow`. There are also `LoginWidget` and `LogoutWidget` components with `Logout/Login` as their parent.

The injector tree does not make a new injector for every component, but does make a new injector for every component with a `providers` array in its decorator.

Components that have no `providers` array look to their parent component for an injector. If the parent does not have an injector, it looks up until it reaches the root injector.

Warning: Be careful with `provider` arrays. If a child component is decorated with a `providers` array that contains dependencies that were *also* requested in the parent component(s), the dependencies the child receives will shadow the parent dependencies. This can have all sorts of unintended consequences.

Consider the following example:

`app/module.ts`

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { App } from './app.component';
import { ChildInheritor, ChildOwnInjector } from './components/index';
import { Unique } from './services/unique';

const randomFactory = () => { return Math.random(); };

@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    App,
    ChildInheritor,
    ChildOwnInjector,
  ],
  /** Provide dependencies here */
  providers: [
    Unique,
  ],
  bootstrap: [ App ],
})
export class AppModule {}
```

In the example above, `Unique` is bootstrapped into the root injector.

app/services/unique.ts

```
import { Injectable } from '@angular/core';

@Injectable()
export class Unique {
  value: string;
  constructor() {
    this.value = (+Date.now()).toString(16) + '.' +
      Math.floor(Math.random() * 500);
  }
}
```

The `Unique` service generates a value unique to *its* instance upon instantiation.

app/components/child-inheritor.component.ts

```

import { Component, Inject } from '@angular/core';
import { Unique } from '../services/unique';

@Component({
  selector: 'child-inheritor',
  template: `<span>{{ value }}</span>`
})
export class ChildInheritor {
  value: number;
  constructor(u: Unique) {
    this.value = u.value;
  }
}

```

The child inheritor has no injector. It will traverse the component tree upwards looking for an injector.

app/components/child-own-injector.component.ts

```

import { Component, Inject } from '@angular/core';
import { Unique } from '../services/unique';

@Component({
  selector: 'child-own-injector',
  template: `<span>{{ value }}</span>`,
  providers: [Unique]
})
export class ChildOwnInjector {
  value: number;
  constructor(u: Unique) {
    this.value = u.value;
  }
}

```

The child own injector component has an injector that is populated with its own instance of `Unique`. This component will not share the same value as the root injector's `Unique` instance.

app/containers/app.ts

```
import { Component, Inject } from '@angular/core';
import { Unique } from '../services/unique';

@Component({
  selector: 'app',
  template: `
    <p>
      App's Unique dependency has a value of {{ value }}
    </p>
    <p>
      which should match
    </p>
    <p>
      ChildInheritor's value: <child-inheritor></child-inheritor>
    </p>
    <p>
      However,
    </p>
    <p>
      ChildOwnInjector should have its own value: <child-own-injector></child-own-injector>
    </p>
    <p>
      ChildOwnInjector's other instance should also have its own value <child-own-injector></child-own-injector>
    </p>
    ,
  `,
})
export class App {
  value: number;
  constructor(u: Unique) {
    this.value = u.value;
  }
}
```

[View Example](#)

Modules

Angular Modules provides a mechanism for creating blocks of functionality that can be combined to build an application.

What is an Angular 2 Module?

In Angular 2, a module is a mechanism to group components, directives, pipes and services that are related, in such a way that can be combined with other modules to create an application. An Angular 2 application can be thought of as a puzzle where each piece (or each module) is needed to be able to see the full picture.

Another analogy to understand Angular 2 modules is classes. In a class, we can define public or private methods. The public methods are the API that other parts of our code can use to interact with it while the private methods are implementation details that are hidden. In the same way, a module can export or hide components, directives, pipes and services. The exported elements are meant to be used by other modules, while the ones that are not exported (hidden) are just used inside the module itself and cannot be directly accessed by other modules of our application.

A Basic Use of Modules

To be able to define modules we have to use the decorator `NgModule`.

```
import { NgModule } from '@angular/core';

@NgModule({
  imports: [ ... ],
  declarations: [ ... ],
  bootstrap: [ ... ]
})
export class AppModule { }
```

In the example above, we have turned the class `AppModule` into an Angular 2 module just by using the `NgModule` decorator. The `NgModule` decorator requires at least three properties:

`imports` , `declarations` and `bootstrap` .

The property `imports` expects an array of modules. Here's where we define the pieces of our puzzle (our application). The property `declarations` expects an array of components, directives and pipes that are part of the module. The `bootstrap` property is where we define the root component of our module. Even though this property is also an array, 99% of the time we are going to define only one component.

There are very special circumstances where more than one component may be required to bootstrap a module but we are not going to cover those edge cases here.

Here's how a basic module made up of just one component would look like:

app/app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'rio-app',
  template: '<h1>My Angular 2 App</h1>'
})
export class AppComponent {}
```

app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

The file `app.component.ts` is just a "hello world" component, nothing interesting there. In the other hand, the file `app.module.ts` is following the structure that we've seen before for defining a module but in this case, we are defining the modules and components that we are going to be using.

The first thing that we notice is that our module is importing the `BrowserModule` as an explicit dependency. The `BrowserModule` is a built-in module that exports basic directives, pipes and services. Unlike previous versions of Angular 2, we have to explicitly import those dependencies to be able to use directives like `*ngFor` or `*ngIf` in our templates.

Given that the root (and only) component of our module is the `AppComponent` we have to list it in the `bootstrap` array. Because in the `declarations` property we are supposed to define **all** the components or pipes that make up our application, we have to define the `AppComponent` again there too.

Before moving on, there's an important clarification to make. **There are two types of modules, root modules and feature modules.**

In the same way that in a module we have one root component and many possible secondary components, **in an application we only have one root module and zero or many feature modules**. To be able to bootstrap our application, Angular needs to know which one is the root module. An easy way to identify a root module is by looking at the `imports` property of its `NgModule` decorator. If the module is importing the `BrowserModule` then it's a root module, if instead is importing the `CommonModule` then it is a feature module.

As developers, we need to take care of importing the `BrowserModule` in the root module and instead, import the `CommonModule` in any other module we create for the same application. Failing to do so might result in problems when working with lazy loaded modules as we are going to see in following sections.

By convention, the root module should always be named `AppModule`.

Bootstrapping an Application

To bootstrap our module based application, we need to inform Angular which one is our root module to perform the compilation in the browser. This compilation in the browser is also known as "Just in Time" (JIT) compilation.

`main.ts`

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

It is also possible to perform the compilation as a build step of our workflow. This method is called "Ahead of Time" (AOT) compilation and will require a slightly different bootstrap process that we are going to discuss in another section.

[View Example](#)

In the next section we are going to see how to create a module with multiple components, services and pipes.

Adding Components, Pipes and Services to a Module

In the previous section, we learned how to create a module with just one component but we know that is hardly the case. Our modules are usually made up of multiple components, services, directives and pipes. In this chapter we are going to extend the example we had before with a custom component, pipe and service.

Let's start by defining a new component that we are going to use to show credit card information.

credit-card.component.ts

```
import { Component, OnInit } from '@angular/core';
import { CreditCardService } from './credit-card.service';

@Component({
  selector: 'rio-credit-card',
  template: `
    <p>Your credit card is: {{ creditCardNumber | creditCardMask }}</p>
  `
})
export class CreditCardComponent implements OnInit {
  creditCardNumber: string;

  constructor(private creditCardService: CreditCardService) {}

  ngOnInit() {
    this.creditCardNumber = this.creditCardService.getCreditCard();
  }
}
```

This component is relying on the `CreditCardService` to get the credit card number, and on the pipe `creditCardMask` to mask the number except the last 4 digits that are going to be visible.

credit-card.service.ts

```
import { Injectable } from '@angular/core';

@Injectable()
export class CreditCardService {
  getCreditCard(): string {
    return '2131313133123174098';
  }
}
```

credit-card-mask.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'creditCardMask'
})
export class CreditCardMaskPipe implements PipeTransform {
  transform(plainCreditCard: string): string {
    const visibleDigits = 4;
    let maskedSection = plainCreditCard.slice(0, -visibleDigits);
    let visibleSection = plainCreditCard.slice(-visibleDigits);
    return maskedSection.replace(/\./g, '*') + visibleSection;
  }
}
```

With everything in place, we can now use the `creditCardComponent` in our root component.

app.component.ts

```
import { Component } from "@angular/core";

@Component({
  selector: 'rio-app',
  template: `
    <h1>My Angular 2 App</h1>
    <rio-credit-card></rio-credit-card>
  `
})
export class AppComponent {}
```

Of course, to be able to use this new component, pipe and service, we need to update our module, otherwise Angular is not going to be able to compile our application.

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { CreditCardMaskPipe } from './credit-card-mask.pipe';
import { CreditCardService } from './credit-card.service';
import { CreditCardComponent } from './credit-card.component';

@NgModule({
  imports: [BrowserModule],
  providers: [CreditCardService],
  declarations: [
    AppComponent,
    CreditCardMaskPipe,
    CreditCardComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Notice that we have added the component `CreditCardComponent` and the pipe `CreditCardMaskPipe` to the `declarations` property, along with the root component of the module `AppComponent`. In the other hand, our custom service is configured with the dependency injection system with the `providers` property.

[View Example](#)

Be aware that this method of defining a service in the `providers` property **should only be used in the root module**. Doing this in a feature module is going to cause unintended consequences when working with lazy loaded modules.

In the next section, we are going to see how to safely define services in feature modules.

Creating a Feature Module

When our root module start growing, it starts to be evident that some elements (components, directives, etc.) are related in a way that almost feel like they belong to a library that can be "plugged in".

In our previous example, we started to see that. Our root module has a component, a pipe and a service that its only purpose is to deal with credit cards. What if we extract these three elements to their own **feature module** and then we import it into our **root module**?

We are going to do just that. The first step is to create two folders to differentiate the elements that belong to the root module from the elements that belong to the feature module.

```
.  
├── app  
│   ├── app.component.ts  
│   └── app.module.ts  
├── credit-card  
│   ├── credit-card-mask.pipe.ts  
│   ├── credit-card.component.ts  
│   ├── credit-card.module.ts  
│   └── credit-card.service.ts  
└── index.html  
└── main.ts
```

Notice how each folder has its own module file: *app.module.ts* and *credit-card.module.ts*.

Let's focus on the latter first.

credit-card/credit-card.module.ts

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { CreditCardMaskPipe } from './credit-card-mask.pipe';
import { CreditCardService } from './credit-card.service';
import { CreditCardComponent } from './credit-card.component';

@NgModule({
  imports: [CommonModule],
  declarations: [
    CreditCardMaskPipe,
    CreditCardComponent
  ],
  providers: [CreditCardService],
  exports: [CreditCardComponent]
})
export class CreditCardModule {}

```

Our feature `CreditCardModule` it's pretty similar to the root `AppModule` with a few important differences:

- We are not importing the `BrowserModule` but the `CommonModule`. If we see the documentation of the `BrowserModule` [here](#), we can see that it's re-exporting the `CommonModule` with a lot of other services that helps with rendering an Angular 2 application in the browser. These services are coupling our root module with a particular platform (the browser), but we want our feature modules to be platform independent. That's why we only import the `commonModule` there, which only exports common directives and pipes.

When it comes to components, pipes and directives, every module should import its own dependencies disregarding if the same dependencies were imported in the root module or in any other feature module. In short, even when having multiple feature modules, each one of them needs to import the `CommonModule`.

- We are using a new property called `exports`. Every element defined in the `declarations` array is **private by default**. We should only export whatever the other modules in our application need to perform its job. In our case, we only need to make the `CreditCardComponent` visible because it's being used in the template of the `AppComponent`.

app/app.component.ts

```

...
@Component({
  ...
  template: `
    ...
    <rio-credit-card></rio-credit-card>
  `
})
export class AppComponent {}

```

We are keeping the `CreditCardMaskPipe` private because it's only being used inside the `CreditCardModule` and no other module should use it directly.

We can now import this feature module into our simplified root module.

`app/app.module.ts`

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { CreditCardModule } from '../credit-card/credit-card.module';
import { AppComponent } from './app.component';

@NgModule({
  imports: [
    BrowserModule,
    CreditCardModule
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule {} 

```

At this point we are done and our application behaves as expected.

[View Example](#)

Services and Lazy Loaded Modules

Here's the tricky part of Angular modules. While components, pipes and directives are scoped to its modules unless explicitly exported, services are globally available unless the module is lazy loaded.

It's hard to understand that at first so let's try to see what's happening with the `CreditCardService` in our example. Notice first that the service is not in the `exports` array but in the `providers` array. With this configuration, our service is going to be available

everywhere, even in the `AppComponent` which lives in another module. So, even when using modules, there's no way to have a "private" service unless... the module is being lazy loaded.

When a module is lazy loaded, Angular is going to create a child injector (which is a child of the root injector from the root module) and will create an instance of our service there.

Imagine for a moment that our `CreditCardModule` is configured to be lazy loaded. With our current configuration, when the application is bootstrapped and our root module is loaded in memory, an instance of the `CreditCardService` (a singleton) is going to be added to the root injector. But, when the `CreditCardModule` is lazy loaded sometime in the future, a child injector will be created for that module **with a new instance** of the `CreditCardService`. At this point we have a hierarchical injector with **two instances** of the same service, which is not usually what we want.

Think for example of a service that does the authentication. We want to have only one singleton in the entire application, disregarding if our modules are being loaded at bootstrap or lazy loaded. So, in order to have our feature module's service **only** added to the root injector, we need to use a different approach.

`credit-card/credit-card.module.ts`

```
import { NgModule, ModuleWithProviders } from '@angular/core';
/* ...other imports... */

@NgModule({
  imports: [CommonModule],
  declarations: [
    CreditCardMaskPipe,
    CreditCardComponent
  ],
  exports: [CreditCardComponent]
})
export class CreditCardModule {
  static forRoot(): ModuleWithProviders {
    return {
      ngModule: CreditCardModule,
      providers: [CreditCardService]
    }
  }
}
```

Different than before, we are not putting our service directly in the property `providers` of the `NgModule` decorator. This time we are defining a static method called `forRoot` where we define the module **and** the service we want to export.

With this new syntax, our root module is slightly different.

app/app.module.ts

```
/* ...imports... */\n\n@NgModule({\n  imports: [\n    BrowserModule,\n    CreditCardModule.forRoot()\n  ],\n  declarations: [AppComponent],\n  bootstrap: [AppComponent]\n})\nexport class AppModule { }
```

Can you spot the difference? We are not importing the `CreditCardModule` directly, instead what we are importing is the object returned from the `forRoot` method, which includes the `CreditCardService`. Although this syntax is a little more convoluted than the original, it will guarantee us that only one instance of the `CreditCardService` is added to the root module. When the `CreditCardModule` is loaded (even lazy loaded), no new instance of that service is going to be added to the child injector.

[View Example](#)

As a rule of thumb, **always use the `forRoot` syntax when exporting services from feature modules**, unless you have a very special need that requires multiple instances at different levels of the dependency injection tree.

Directive Duplications

Because we no longer define every component and directive directly in every component that needs it, we need to be aware of how Angular modules handle directives and components that target the same element (have the same selector).

Let's assume for a moment that by mistake, we have created two directives that target the same property:

This example is a variation of the code found in the [official documentation](#).

blue-highlight.directive.ts

```
import { Directive, ElementRef, Renderer } from '@angular/core';

@Directive({
  selector: '[highlight]'
})
export class BlueHighlightDirective {
  constructor(renderer: Renderer, el: ElementRef) {
    renderer.setStyle(el.nativeElement, 'backgroundColor', 'blue');
    renderer.setStyle(el.nativeElement, 'color', 'gray');
  }
}
```

yellow-highlight.directive.ts

```
import { Directive, ElementRef, Renderer } from '@angular/core';

@Directive({
  selector: '[highlight]'
})
export class YellowHighlightDirective {
  constructor(renderer: Renderer, el: ElementRef) {
    renderer.setStyle(el.nativeElement, 'backgroundColor', 'yellow');
  }
}
```

These two directives are similar, they are trying to style an element. The `BlueHighlightDirective` will try to set the background color of the element to blue while changing the color of the text to gray, while the `YellowHighlightDirective` will try only to change the background color to yellow. Notice that both are targeting any HTML element that has the property `highlight`. What would happen if we add both directives to the same module?

app.module.ts

```
// Imports

@NgModule({
  imports: [BrowserModule],
  declarations: [
    AppComponent,
    BlueHighlightDirective,
    YellowHighlightDirective
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Let's see how we would use it in the only component of the module.

app.component.ts

```
import { Component } from "@angular/core";

@Component({
  selector: 'rio-app',
  template: '<h1 highlight>My Angular 2 App</h1>'
})
export class AppComponent {}
```

We can see that in the template of our component, we are using the directive `highlight` in our `h1` element but, which styles are going to end up being applied?

The answer is: the text is going to be gray and the background yellow.

[View Example](#)

We are allowed to define multiple directives that target the same elements in the same module. What's going to happen is that Angular is going to do every transformation **in order**.

```
declarations: [
  ...,
  BlueHighlightDirective,
  YellowHighlightDirective
]
```

Because we have defined both directives in an array, and **arrays are ordered collection of items**, when the compiler finds an element with the property `highlight`, it will first apply the transformations of `BlueHighlightDirective`, setting the text gray and the background blue,

and then will apply the transformations of `YellowHighlightDirective`, changing again the background color to yellow.

In summary, **when two or more directives target the same element, they are going to be applied in the order they were defined.**

Lazy Loading a Module

Another advantage of using modules to group related pieces of functionality of our application is the ability to load those pieces on demand. Lazy loading modules helps us decrease the startup time. With lazy loading our application does not need to load everything at once, it only needs to load what the user expects to see when the app first loads. Modules that are lazily loaded will only be loaded when the user navigates to their routes.

To show this relationship, let's start by defining a simple module that will act as the root module of our example application.

app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { EagerComponent } from './eager.component';
import { routing } from './app.routing';

@NgModule({
  imports: [
    BrowserModule,
    routing
  ],
  declarations: [
    AppComponent,
    EagerComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

So far this is a very common module that relies on the `BrowserModule`, has a `routing` mechanism and two components: `AppComponent` and `EagerComponent`. For now, let's focus on the root component of our application (`AppComponent`) where the navigation is defined.

app/app.component.ts

```

import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <h1>My App</h1>
    <nav>
      <a routerLink="eager">Eager</a>
      <a routerLink="lazy">Lazy</a>
    </nav>
    <router-outlet></router-outlet>
  `
})
export class AppComponent {}

```

Our navigation system has only two paths: `eager` and `lazy`. To know what those paths are loading when clicking on them we need to take a look at the `routing` object that we passed to the root module.

`app/app.routing.ts`

```

import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { EagerComponent } from './eager.component';

const routes: Routes = [
  { path: '', redirectTo: 'eager', pathMatch: 'full' },
  { path: 'eager', component: EagerComponent },
  { path: 'lazy', loadChildren: 'lazy/lazy.module#LazyModule' }
];

export const routing: ModuleWithProviders = RouterModule.forRoot(routes);

```

Here we can see that the default path in our application is called `eager` which will load `EagerComponent`.

`app/eager.component.ts`

```

import { Component } from '@angular/core';

@Component({
  template: '<p>Eager Component</p>'
})
export class EagerComponent {}

```

But more importantly, we can see that whenever we try to go to the path `lazy`, we are going to lazy load a module conveniently called `LazyModule`. Look closely at the definition of that route:

```
{ path: 'lazy', loadChildren: 'lazy/lazy.module#LazyModule' }
```

There's a few important things to notice here:

1. We use the property `loadChildren` instead of `component`.
2. We pass a string instead of a symbol to avoid loading the module eagerly.
3. We define not only the path to the module but the name of the class as well.

There's nothing special about `LazyModule` other than it has its own `routing` and a component called `LazyComponent`.

app/lazy/lazy.module.ts

```
import { NgModule } from '@angular/core';

import { LazyComponent } from './lazy.component';
import { routing } from './lazy.routing';

@NgModule({
  imports: [routing],
  declarations: [LazyComponent]
})
export class LazyModule {}
```

If we define the class `LazyModule` as the `default` export of the file, we don't need to define the class name in the `loadChildren` property as shown above.

The `routing` object is very simple and only defines the default component to load when navigating to the `lazy` path.

app/lazy/lazy.routing.ts

```
import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { LazyComponent } from './lazy.component';

const routes: Routes = [
  { path: '', component: LazyComponent }
];

export const routing: ModuleWithProviders = RouterModule.forChild(routes);
```

Notice that we use the method call `forChild` instead of `forRoot` to create the routing object. We should always do that when creating a routing object for a feature module, no matter if the module is supposed to be eagerly or lazily loaded.

Finally, our `LazyComponent` is very similar to `EagerComponent` and is just a placeholder for some text.

`app/lazy/lazy.component.ts`

```
import { Component } from '@angular/core';

@Component({
  template: '<p>Lazy Component</p>'
})
export class LazyComponent {}
```

[View Example](#)

When we load our application for the first time, the `AppModule` along the `AppComponent` will be loaded in the browser and we should see the navigation system and the text "Eager Component". Until this point, the `LazyModule` has not been downloaded, only when we click the link "Lazy" the needed code will be downloaded and we will see the message "Lazy Component" in the browser.

We have effectively lazily loaded a module.

Lazy Loading and the Dependency Injection Tree

Lazy loaded modules create their own branch on the Dependency Injection (DI) tree. This means that it's possible to have services that belong to a lazy loaded module, that are not accessible by the root module or any other eagerly loaded module of our application.

To show this behaviour, let's continue with the example of the previous section and add a

`CounterService` to our `LazyModule`.

`app/lazy/lazy.module.ts`

```
...
import { CounterService } from './counter.service';

@NgModule({
  ...
  providers: [CounterService]
})
export class LazyModule {}
```

Here we added the `CounterService` to the `providers` array. Our `CounterService` is a simple class that holds a reference to a `counter` property.

`app/lazy/counter.service.ts`

```
import { Injectable } from '@angular/core';

@Injectable()
export class CounterService {
  counter = 0;
}
```

We can modify the `LazyComponent` to use this service with a button to increment the `counter` property.

`app/lazy/lazy.component.ts`

```
import { Component } from '@angular/core';

import { CounterService } from './counter.service';

@Component({
  template: `
    <p>Lazy Component</p>
    <button (click)="increaseCounter()">Increase Counter</button>
    <p>Counter: {{ counterService.counter }}</p>
  `
})
export class LazyComponent {

  constructor(public counterService: CounterService) {}

  increaseCounter() {
    this.counterService.counter += 1;
  }
}
```

[View Example](#)

The service is working. If we increment the counter and then navigate back and forth between the `eager` and the `lazy` routes, the `counter` value will persist in the lazy loaded module.

But the question is, how can we verify that the service is isolated and cannot be used in a component that belongs to a different module? Let's try to use the same service in the `EagerComponent`.

app/eager.component.ts

```
import { Component } from '@angular/core';
import { CounterService } from './lazy/counter.service';

@Component({
  template: `
    <p>Eager Component</p>
    <button (click)="increaseCounter()">Increase Counter</button>
    <p>Counter: {{ counterService.counter }}</p>
  `
})
export class EagerComponent {
  constructor(public counterService: CounterService) {}

  increaseCounter() {
    this.counterService.counter += 1;
  }
}
```

If we try to run this new version of our code, we are going to get an error message in the browser console:

```
No provider for CounterService!
```

What this error tells us is that the `AppModule`, where the `EagerComponent` is defined, has no knowledge of a service called `CounterService`. `CounterService` lives in a different branch of the DI tree created for `LazyModule` when it was lazy loaded in the browser.

Shared Modules and Dependency Injection

Now that we have proven that lazy loaded modules create their own branch on the Dependency Injection tree, we need to learn how to deal with services that are imported by means of a shared module in both an eager and lazy loaded module.

Let's create a new module called `SharedModule` and define the `CounterService` there.

app/shared/shared.module.ts

```
import { NgModule } from '@angular/core';
import { CounterService } from './counter.service';

@NgModule({
  providers: [CounterService]
})
export class SharedModule {}
```

Now we are going to import that `SharedModule` in the `AppModule` and the `LazyModule` .

app/app.module.ts

```
...
import { SharedModule } from './shared/shared.module';

@NgModule({
  imports: [
    SharedModule,
    ...
  ],
  declarations: [
    EagerComponent,
    ...
  ]
})
export class AppModule {}
```

app/lazy/lazy.module.ts

```

...
import { SharedModule } from '../shared/shared.module';

@NgModule({
  imports: [
    SharedModule,
    ...
  ],
  declarations: [LazyComponent]
})
export class LazyModule {}

```

With this configuration, the components of both modules will have access to the `CounterService`. We are going to use this service in `EagerComponent` and `LazyComponent` in exactly the same way. Just a button to increase the internal `counter` property of the service.

`app/eager.component.ts`

```

import { Component } from '@angular/core';
import { CounterService } from './shared/counter.service';

@Component({
  template: `
    <p>Eager Component</p>
    <button (click)="increaseCounter()">Increase Counter</button>
    <p>Counter: {{ counterService.counter }}</p>
  `
})
export class EagerComponent {
  constructor(public counterService: CounterService) {}

  increaseCounter() {
    this.counterService.counter += 1;
  }
}

```

[View Example](#)

If you play with the live example, you will notice that the `counter` seems to behave independently in `EagerComponent` and `LazyComponent`, we can increase the value of one counter without altering the other one. In other words, we have ended up with two instances of the `CounterService`, one that lives in the root of the DI tree of the `AppModule` and another that lives in a lower branch of the DI tree accessible by the `LazyModule`.

This is not necessarily wrong, you may find situations where you could need different instances of the same service, but I bet most of the time that's not what you want. Think for example of an authentication service, you need to have the same instance with the same

information available everywhere disregarding if we are using the service in an eagerly or lazy loaded module.

In the next section we are going to learn how to have only one instance of a shared service.

Sharing the Same Dependency Injection Tree

So far our problem is that we are creating two instances of the same services in different levels of the DI tree. The instance created in the lower branch of the tree is shadowing the one defined at the root level. The solution? To avoid creating a second instance in a lower level of the DI tree for the lazy loaded module and only use the service instance registered at the root of the tree.

To accomplish that, we need to modify the definition of the `SharedModule` and instead of defining our service in the `providers` property, we need to create a static method called `forRoot` that exports the service along with the module itself.

app/shared/shared.module.ts

```
import { NgModule, ModuleWithProviders } from '@angular/core';
import { CounterService } from './counter.service';

@NgModule({})
export class SharedModule {
  static forRoot(): ModuleWithProviders {
    return {
      ngModule: SharedModule,
      providers: [CounterService]
    };
  }
}
```

With this setup, we can import this module in our root module `AppModule` calling the `forRoot` method to register the module and the service.

app/app.module.ts

```
...
import { SharedModule } from './shared/shared.module';

@NgModule({
  imports: [
    SharedModule.forRoot(),
    ...
  ],
  ...
})
export class AppModule {}
```

In contrast, when import the same module in our `LazyModule` we will not call the `forRoot` method because we don't want to register the service again in a different level of the DI tree, so the declaration of the `LazyModule` doesn't change.

`app/lazy/lazy.module.ts`

```
...
import { SharedModule } from './shared/shared.module';

@NgModule({
  imports: [
    SharedModule,
    ...
  ],
  ...
})
export class LazyModule {}
```

[View Example](#)

This time, whenever we change the value of the `counter` property, this value is shared between the `EagerComponent` and the `LazyComponent` proving that we are using the same instance of the `CounterService`.

Project Setup

Proper tooling and setup is good for any project, but it's especially important for Angular 2 due to all of the pieces that are involved. We've decided to use [webpack](#), a powerful tool that attempts to handle our complex integrations. Due to the number of parts of our project that webpack touches, it's important to go over the configuration to get a good understanding of what gets generated client-side.

Webpack

Table of Content

- [Installation and Usage](#)
- [Loaders](#)
- [Plugins](#)
- [Summary](#)

A modern JavaScript web application includes a lot of different packages and dependencies, and it's important to have something that makes sense of it all in a simple way.

Angular 2 takes the approach of breaking your application apart into many different components, each of which can have several files. Separating application logic this way is good for the programmer, but can detract from user experience since doing this can increase page loading time. HTTP2 aims to solve this problem in one way, but until more is known about its effects we will want to bundle different parts of our application together and compress it.

Our platform, the browser, must continue to provide backwards compatibility for all existing code and this necessitates slow movement of additions to the base functionality of HTML/CSS/JS. The community has created different tools that transform their preferred syntax/feature set to what the browser supports to avoid binding themselves to the constraints of the web platform. This is especially evident in Angular 2 applications, where [TypeScript](#) is used heavily. Although we don't do this in our course, projects may also involve different CSS preprocessors (sass, stylus) or templating engines (jade, Mustache, EJS) that must be integrated.

Webpack solves these problems by providing a common interface to integrate all of these tools and that allows us to streamline our workflow and avoid complexity.

Installation

The easiest way to include webpack and its plugins is through NPM and save it to your

```
devDependencies :
```

```
npm install -D webpack ts-loader html-webpack-plugin tslint-loader
```

Setup and Usage

The most common way to use webpack is through the CLI. By default, running the command executes `webpack.config.js` which is the configuration file for your webpack setup.

Bundle

The core concept of webpack is the *bundle*. A bundle is simply a collection of modules, where we define the boundaries for how they are separated. In this project, we have two bundles:

- `app` for our application-specific client-side logic
- `vendor` for third party libraries

In webpack, bundles are configured through *entry points*. Webpack goes through each entry point one by one. It maps out a dependency graph by going through each module's references. All the dependencies that it encounters are then packaged into that bundle.

Packages installed through NPM are referenced using *CommonJS* module resolution. In a JavaScript file, this would look like:

```
const app = require('./src/index.ts');
```

or TypeScript/ES6 file:

```
import { Component } from '@angular/core';
```

We will use those string values as the module names we pass to webpack.

Let's look at the entry points we have defined in our sample app:

```
{  
  ...  
  entry: {  
    app: './src/index.ts',  
    vendor: [  
      '@angular/core',  
      '@angular/compiler',  
      '@angular/common',  
      '@angular/http',  
      '@angular/platform-browser',  
      '@angular/platform-browser-dynamic',  
      '@angular/router',  
      'es6-shim',  
      'redux',  
      'redux-thunk',  
      'redux-logger',  
      'reflect-metadata',  
      'ng2-redux',  
      'zone.js',  
    ]  
  },  
  ...  
}
```

The entry point for `app`, `./src/index.ts`, is the base file of our Angular 2 application. If we've defined the dependencies of each module correctly, those references should connect all the parts of our application from here. The entry point for `vendor` is a list of modules that we need for our application code to work correctly. Even if these files are referenced by some module in our app bundle, we want to separate these resources in a bundle just for third party code.

Output Configuration

In most cases we don't just want to configure how webpack generates bundles - we also want to configure how those bundles are output.

- Often, we will want to re-route where files are saved. For example into a `bin` or `dist` folder. This is because we want to optimize our builds for production.
- Webpack transforms the code when bundling our modules and outputting them. We want to have a way of connecting the code that's been generated by webpack and the code that we've written.
- Server routes can be configured in many different ways. We probably want some way of configuring webpack to take our server routing setup into consideration.

All of these configuration options are handled by the config's `output` property. Let's look at how we've set up our config to address these issues:

```
{  
  ...  
  output: {  
    path: path.resolve(__dirname, 'dist'),  
    filename: '[name].[hash].js',  
    publicPath: "/",  
    sourceMapFilename: '[name].[hash].js.map'  
  }  
  ...  
}
```

Some options have words wrapped in square brackets. Webpack has the ability to parse parameters for these properties, with each property having a different set of parameters available for substitution. Here, we're using `name` (the name of the bundle) and `hash` (a hash value of the bundle's content).

To save bundled files in a different folder, we use the `path` property. Here, `path` tells webpack that all of the output files must be saved to `path.resolve(__dirname, 'dist')`. In our case, we save each bundle into a separate file. The name of this file is specified by the `filename` property.

Linking these bundled files and the files we've actually coded is done using what's known as source maps. There are different ways to configure source maps. What we want is to save these source maps in a separate file specified by the `sourceMapFilename` property. The way the server accesses the files might not directly follow the filesystem tree. For us, we want to use the files saved under `dist` as the root folder for our server. To let webpack know this, we've set the `publicPath` property to `/`.

Plugins

Plugins allow us to inject custom build steps during the bundling process.

A commonly used plugin is the `html-webpack-plugin`. This allows us to generate HTML files required for production. For example it can be used to inject script tags for the output bundles.

```
new HtmlWebpackPlugin({
  template: './src/index.html',
  inject: 'body',
  minify: false
});
```

Summary

When we put everything together, our complete `webpack.config.js` file looks something like this:

```
'use strict';

const path = require("path");
const webpack = require('webpack');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const basePlugins = [
  new webpack.optimize.CommonsChunkPlugin('vendor', '[name].[hash].bundle.js'),
  new HtmlWebpackPlugin({
    template: './src/index.html',
    inject: 'body',
    minify: false
  })
];

const envPlugins = {
  production: [
    new webpack.optimize.UglifyJsPlugin({
      compress: {
        warnings: false
      }
    })
  ],
  development: []
};

const plugins = basePlugins.concat(envPlugins[process.env.NODE_ENV] || []);

module.exports = {
  entry: {
    app: './src/index.ts',
    vendor: [
      '@angular/core',
      '@angular/compiler',
      '@angular/common',
      '@angular/http',
      '@angular/platform-browser',
      '@angular/platform-browser-dynamic',
      '@angular/router',
      'es6-shim',
      'redux',
      'redux-thunk',
      'redux-logger',
    ]
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].[hash].bundle.js',
    publicPath: '/dist/'
  },
  resolve: {
    extensions: ['.ts', '.js']
  },
  module: {
    rules: [
      { test: /\.ts$/, loader: 'ts-loader' },
      { test: /\.css$/, loader: 'style-loader!css-loader' }
    ]
  },
  plugins: plugins,
  optimization: {
    minimize: true
  }
};
```

```
'reflect-metadata',
'ng2-redux',
'zone.js',
],
},

output: {
  path: path.resolve(__dirname, 'dist'),
  filename: '[name].[hash].js',
  publicPath: "/",
  sourceMapFilename: '[name].[hash].js.map'
},
devtool: 'source-map',

resolve: {
  extensions: ['.webpack.js', '.web.js', '.ts', '.js']
},
plugins: plugins,

module: {
  rules: [
    { test: /\.ts$/, loader: 'tslint' },
    { test: /\.ts$/, loader: 'ts', exclude: /node_modules/ },
    { test: /\.html$/, loader: 'raw' },
    { test: /\.css$/, loader: 'style!css?sourceMap' },
    { test: /\.svg/, loader: 'url' },
    { test: /\.eot/, loader: 'url' },
    { test: /\.woff/, loader: 'url' },
    { test: /\.woff2/, loader: 'url' },
    { test: /\.ttf/, loader: 'url' },
  ],
  noParse: [ /zone\.js\/dist\/.+, /angular2\/bundles\/.+/ ]
}
}
```

Going Further

Webpack also does things like hot code reloading and code optimization which we haven't covered. For more information you can check out the [official documentation](#). The source is also available on [Github](#).

NPM Scripts Integration

NPM allows us to define custom scripts in the `package.json` file. These can then execute tasks using the NPM CLI.

We rely on these scripts to manage most of our project tasks and webpack fits in as well.

The scripts are defined in the `scripts` property of the `package.json` file. For example:

```
...
scripts: {
  "clean": "rimraf dist",
  "prebuild": "npm run clean",
  "build": "NODE_ENV=production webpack",
}
...
```

NPM allows pre and post task binding by prepending the word `pre` or `post` respectively to the task name. Here, our `prebuild` task is executed before our `build` task.

We can run an NPM script from inside another NPM script.

To invoke the `build` script we run the command `npm run build`:

1. The `prebuild` task executes.
2. The `prebuild` task runs the `clean` task, which executes the `rimraf dist` command.
3. `rimraf` (an NPM package) recursively deletes everything inside a specified folder.
4. The `build` task is executed. This sets the `NODE_ENV` environment variable to `production` and starts the webpack bundling process.
5. Webpack generates bundles based on the `webpack.config.js` available in the project root folder.

Angular CLI

Table of Content

- [Setup](#)
- [Creating a New App](#)
- [Serving the App](#)
- [Creating Components](#)
- [Creating Routes](#)
- [Creating Other Things](#)
- [Testing](#)
- [Linting](#)
- [CLI Command Overview](#)
- [Adding Third Party Libraries](#)
- [Integrating an Existing App](#)

With all of the new features Angular 2 takes advantage of, like static typing, decorators and ES6 module resolution, comes the added cost of setup and maintenance. Spending a lot of time with different build setups and configuring all of the different tools used to serve a modern JavaScript application can really take a lot of time and drain productivity by not being able to actually work on the app itself.

Seeing the popularity of [ember-cli](#), Angular 2 decided they would provide their own CLI to solve this problem. [Angular CLI](#) is geared to be the tool used to create and manage your Angular 2 app. It provides the ability to:

- create a project from scratch
- scaffold components, directives, services, etc.
- lint your code
- serve the application
- run your unit tests and end to end tests.

The Angular 2 CLI currently only generates scaffolding in TypeScript, with other dialects to come later.

Setup

Prerequisites

Angular CLI is currently only distributed through npm and requires Node version 4 or greater.

Installation

The Angular 2 CLI can be installed with the following command:

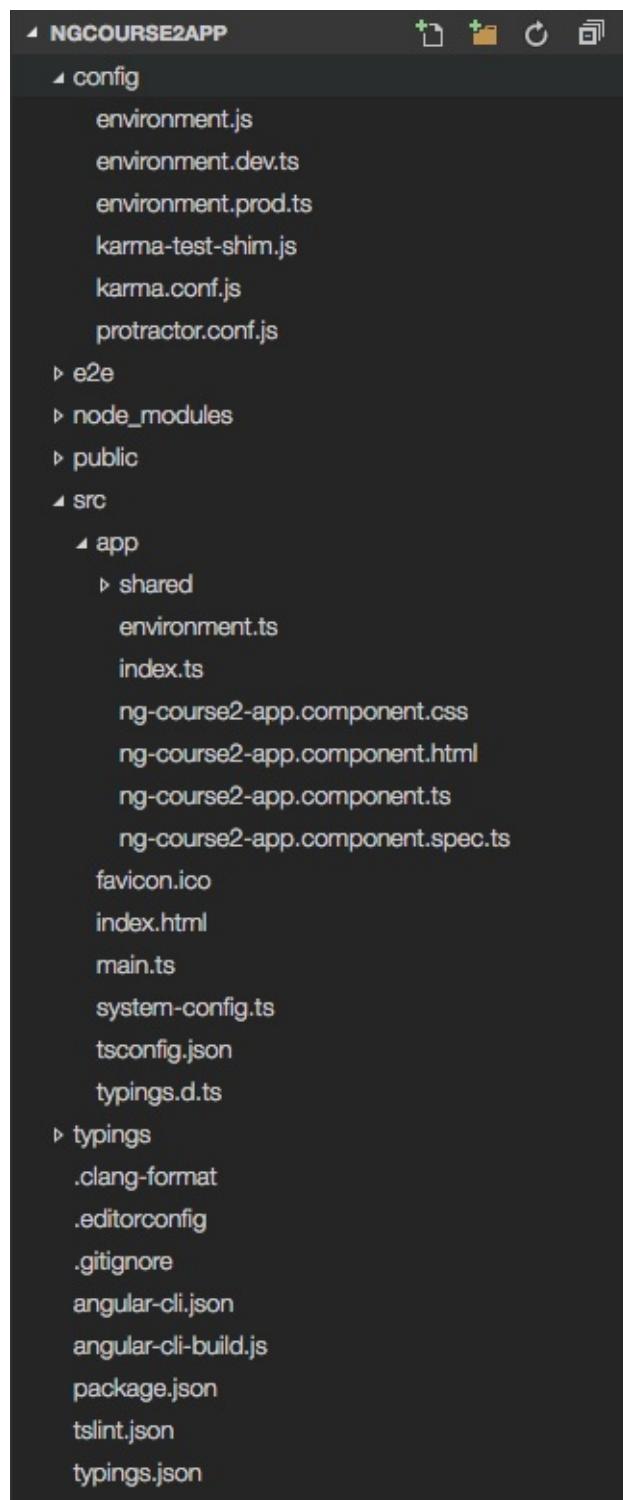
```
npm install -g angular-cli
```

Creating a New App

Use the `ng new [app-name]` command to create a new app. This will generate a basic app in the folder of the app name provided. The app has all of the features available to work with the CLI commands. Creating an app may take a few minutes to complete since npm will need to install all of the dependencies. The directory is automatically set up as a new **git** repository as well. If git is not your version control of choice, simply remove the `.git` folder and `.gitignore` file.

File and Folder Setup

The generated app folder will look like this:



Application configuration is stored in different places, some located in the `config` folder, such as test configuration, and some being stored in the project root such as linting information and build information. The CLI stores application-specific files in the `src` folder and Angular 2-specific code in the `src/app` folder. Files and folders generated by the CLI will follow the [official style guide](#).

Warning: The CLI relies on some of the settings defined in the configuration files to be able to execute the commands. Take care when modifying them, particularly the `package.json` file.

The CLI has installed everything a basic Angular 2 application needs to run properly. To make sure everything has run and installed correctly we can run the server.

Serving the App

The CLI provides the ability to serve the app with live reload. To serve an application, simply run the command `ng serve`. This will compile the app and copy all of the application-specific files to the `dist` folder before serving.

By default, `ng serve` serves the application locally on port 4200 (<http://localhost:4200>) but this can be changed by using a command line argument: `ng serve --port=8080`.

Creating Components

The CLI can scaffold Angular 2 components through the `generate` command. To create a new component run:

```
ng generate component [component-name]
```

Executing the command creates a folder, `[component-name]`, in the project's `src/app` path or the current path the command is executed in if it's a child folder of the project. The folder has the following:

- `[component-name].component.ts` the component class file
- `[component-name].component.css` for styling the component
- `[component-name].component.html` component html
- `[component-name].component.spec.ts` tests for the component
- `index.ts` which exports the component

Creating Routes

The `ng g route [route-name]` command will spin up a new folder and route files for you.

At the time of writing this feature was temporarily disabled due to ongoing changes happening with Angular 2 routing.

Creating Other Things

The CLI can scaffold other Angular 2 entities such as services, pipes and directives using the generate command.

```
ng generate [entity] [entity-name]
```

This creates the entity at `src/app/[entity-name].[entity].ts` along with a spec file, or at the current path if the command is executed in a child folder of the project. The CLI provides blueprints for the following entities out of the box:

Item	Command	Files generated
Component:	<code>ng g component [name]</code>	component, HTML, CSS, test spec files
Directive:	<code>ng g directive [name]</code>	component, test spec files
Pipe:	<code>ng g pipe [name]</code>	component, test spec files
Service:	<code>ng g service [name]</code>	component, test spec files
Class:	<code>ng g class [name]</code>	component, test spec files
Route:	<code>ng g route [name]</code>	component, HTML, CSS, test spec files (in new folder)

Testing

Apps generated by the CLI integrate automated tests. The CLI does this by using the [Karma test runner](#).

Unit Tests

To execute unit tests, run `ng test`. This will run all the tests that are matched by the Karma configuration file at `config/karma.conf.js`. It's set to match all TypeScript files that end in `.spec.ts` by default.

End-to-End Tests

End-to-end tests can be executed by running `ng e2e`. Before end-to-end tests can be performed, the application must be served at some address. Angular CLI uses protractor. It will attempt to access `localhost:4200` by default; if another port is being used, you will have to update the configuration settings located at `config/protractor.conf.js`.

Linting

To encourage coding best practices Angular CLI provides built-in linting. By default the app will look at the project's `tslint.json` for configuration. Linting can be executed by running the command `ng lint`.

For a reference of tslint rules have a look at: <https://palantir.github.io/tslint/rules/>.

CLI Command Overview

One of the advantages of using the Angular CLI is that it automatically configures a number of useful tools that you can use right away. To get more details on the options for each task, use `ng --help`.

Linting

`ng lint` lints the code in your project using [tslint](#). You can customize the rules for your project by editing `tslint.json`.

You can switch some of these to use your preferred tool by editing the scripts in `package.json`.

Testing

`ng test` triggers a build and then runs the unit tests set up for your app using [Karma](#). Use the `--watch` option to rebuild and retest the app automatically whenever source files change.

Build

`ng build` will build your app (and minify your code) and place it into the default output path, `dist/`.

Serve

`ng serve` builds and serves your app on a local server and will automatically rebuild on file changes. By default, your app will be served on <http://localhost:4200/>.

Include `--port [number]` to serve your app on a different HTTP port.

E2E

Once your app is served, you can run end-to-end tests using `ng e2e`. The CLI uses [Protractor](#) for these tests.

Deploy

`ng deploy` deploys to GitHub pages or Firebase.

Adding Third Party Libraries

The CLI generates development automation code which has the ability to integrate third party libraries into the application. Packages are installed using `npm` and the development environment is setup to check the installed libraries mentioned in `package.json` and bundle these third party libraries within the application. For more information see <https://github.com/angular/angular-cli#3rd-party-library-installation>

Integrating an Existing App

Apps that were created without CLI can be integrated to use CLI later on. This is done by going to the existing app's folder and running `ng init`.

Since the folder structure for an existing app might not follow the same format as one created by the CLI, the `init` command has some configuration options.

- `--source-dir` identifies the relative path to the source files (default = `src`)
- `--prefix` identifies the path within the source dir that Angular 2 application files reside (default = `app`)
- `--style` identifies the path where additional style files are located (default = `css`).

Deploying

Glossary

Other Resources