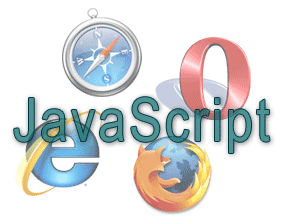
****

**How to Detect Features Instead of Browsers**

Traditionally, many web developers have used browser detection in an attempt to provide a consistent experience between browsers. The typical implementation performs a single comparison operation, usually involving the user-agent string, and then makes several design assumptions about the features supported by that browser. In practice, however, feature detection has proven to be a more effective technique that requires less maintenance. This article shows how to use feature detection to verify support for standards-based featured and demonstrates different ways to detection features effectively.

The Case Against Browser Detection

As typically implemented, browser detection has several drawbacks, including but not limited to the following:

* When a new browser is released or an existing browser is updated, you must factor the new browser into your browser detection code. Updated browsers may support standards and features that were not supported when the browser detection code was designed.
* Conclusions about feature support may not be correct or appropriate.
* As new devices become available, they frequently include new versions of browsers; consequently, browser detection code must be reviewed and potentially modified to support the new browsers. In some cases it becomes more complicated to create customized implementations for each browser.
* A browser detection technique may not accurately identify a given browser. For example, many browsers support the ability to modify the user-agent string.

To illustrate, consider the following (invalid) code sample, which inappropriately attempts to use browser-specific techniques to assign an event handler.

JavaScript

function getInternetExplorerVersion()

// Returns the version of Internet Explorer or a -1

// (indicating the use of another browser).

{

var rv = -1; // Default value assumes failure.

var ua = navigator.userAgent;

// If user agent string contains "MSIE x.y", assume

// Internet Explorer and use "x.y" to determine the

// version.

var re = new RegExp("MSIE ([0-9]{1,}[\.0-9]{0,})");

if (re.exec(ua) != null)

rv = parseFloat( RegExp.$1 );

return rv;

}

function registerEvent( sTargetID, sEventName, fnHandler )

{

var oTarget = document.getElementById( sTargetID );

if ( oTarget != null ) {

// This is not an optimal example; it demonstrates

// techniques that you should not follow.

if ( getInternetExplorerVersion() > -1 ) {

oTarget.attachEvent( "on" + sEventName, fnHandler );

} else {

oTarget.addEventListener( sEventName, fnHandler );

}

}

}

This example attempts to use the user-agent string to determine whether the browser is Windows Internet Explorer. While the example works in Internet Explorer and most other browsers, it contains multiple problems, including but not limited to the following:

* This code sample focuses on the browser (Internet Explorer), not the feature (DOM Level 3 event handler registration). As a result, if Internet Explorer is used to view the webpage, the example uses **[attachEvent](http://msdn.microsoft.com/en-in/library/ie/ms536343(v=vs.85).aspx)** to register the event handlers even if the version of Internet Explorer supports the **[addEventListener](http://msdn.microsoft.com/en-in/library/ie/ff975245(v=vs.85).aspx)** method. In addition, this example would need to be tested with any (hypothetical) new versions of Internet Explorer.
* This example assumes that Internet Explorer does not support the **[addEventListener](http://msdn.microsoft.com/en-in/library/ie/ff975245(v=vs.85).aspx)** method, which is an incorrect assumption with regard to Internet Explorer 9. Many browser detection implementations make similar assumptions.
* While this particular sample does not attempt to distinguish between versions of Internet Explorer, there have been changes in the way Internet Explorer handles the user-agent string. Applications designed according to the behavior of earlier version may report incorrect results for webpages displayed in Compatibility View, which was introduced in Internet Explorer 8. For more information, see [Understanding User-Agent Strings](http://msdn.microsoft.com/en-in/library/ie/ms537503(v=vs.85).aspx) and [Understanding the Compatibility View List](http://msdn.microsoft.com/en-in/library/ie/dd567845(v=vs.85).aspx).

This example illustrates many of the weaknesses with browser detection as a technique for determining the features supported by a web browser. A more effective approach is to detect support for the feature directly, as shown in the following code sample.

JavaScript

function registerEvent( sTargetID, sEventName, fnHandler )

{

var oTarget = document.getElementById( sTargetID );

if ( oTarget != null )

{

if ( oTarget.addEventListener ) {

oTarget.addEventListener( sEventName, fnToBeRun, false );

} else {

var sOnEvent = "on" + sEventName;

if ( oTarget.attachEvent )

{

oTarget.attachEvent( sOnEvent, fnHandler );

}

}

}

}

This example provides two improvements over the previous one:

* It focuses on the feature rather than the browser. If the user happens to be using a browser that supports the[**addEventListener**](http://msdn.microsoft.com/en-in/library/ie/ff975245(v=vs.85).aspx) method (as such Internet Explorer 9 and many other browsers), then that method is used to define the event handler.
* It emphasizes the standards-based technique over the proprietary technique. In this case, the sample verifies support for the preferred technique (the **[addEventListener](http://msdn.microsoft.com/en-in/library/ie/ff975245(v=vs.85).aspx)** method) before attempting to use the alternate technique.

This example is effective because it does not assume the behavior of any given browser. The technique does not need to be updated to support any (hypothetical) new versions of Internet Explorer, nor does it need to be expanded to support new browsers or devices. The sample simply focuses on whether the feature is available. This is the main difference between feature detection and browser detection.

In an ideal world, all browsers would support the same standards and implement those standards in precisely the same way. In practice, however, a number of variations exist between browsers and their respective implementation of various standards.

For web developers, the best defense is to rely on features that are defined in widely supported, stable standards, such as HTML5, Cascading Style Sheets, Level 3 (CSS3), Scalable Vector Graphics (SVG), and so on. Detect support for the features that you want to use and provide alternative approaches only if necessary.

There are a number of ways to detect features, including:

* Search for Document Object Model (DOM) objects or properties associated with the feature.
* Attempt to create an object or attribute related to the feature.
* Use the **[hasFeature](http://msdn.microsoft.com/en-in/library/ie/ms536446(v=vs.85).aspx)** method to determine whether the DOM implementation supports the feature.

For more information, see [How to Create Effective Fallback Strategies](http://msdn.microsoft.com/en-in/library/ie/hh273396(v=vs.85).aspx).

Detecting DOM Objects and Properties

The most common way to detect features is to examine the DOM for objects or properties associated with the feature you want to use. For example, the following code sample shows how to determine whether the browser supports the[**addEventListener**](http://msdn.microsoft.com/en-in/library/ie/ff975245(v=vs.85).aspx) method in order to define an event handler.

JavaScript

function registerEvent( sTargetID, sEventName, fnHandler )

{

var oTarget = document.getElementById( sTargetID );

if ( oTarget != null )

{

if ( oTarget.addEventListener ) {

oTarget.addEventListener( sEventName, fnToBeRun, false );

} else {

var sOnEvent = "on" + sEventName;

if ( oTarget.attachEvent )

{

oTarget.attachEvent( sOnEvent, fnHandler );

}

}

}

}

In this example, the event registration code verifies that the object that handles the event provides support for the [**addEventListener**](http://msdn.microsoft.com/en-in/library/ie/ff975245(v=vs.85).aspx) method before calling the method to register the event handler. This avoids runtime syntax errors and helps provide an alternative approach (also called a *fallback strategy*) if the preferred technique is not supported.

To determine whether there are objects, properties, attributes, or methods that help you detect a feature, refer to the specification that defines the feature you want to use.

For example, Internet Explorer 9 supports the [**performance**](http://msdn.microsoft.com/en-in/library/ie/ff974680(v=vs.85).aspx) object of the [Navigation Timing](http://go.microsoft.com/fwlink/p/?LinkId=220639) specification. As of this writing, this specification defines the "window.performance" attribute. As a result, you can use the presence of a**performance** property on the [**window**](http://msdn.microsoft.com/en-in/library/ie/ms535873(v=vs.85).aspx) object to determine whether the current web browser supports the [Navigation Timing](http://go.microsoft.com/fwlink/p/?LinkId=220639) specification, as shown in the following code sample.

JavaScript

if ( window.performance ) {

showLoadTimes( window.performance );

}

In this example, the showLoadTimes() function is called only when the [**window**](http://msdn.microsoft.com/en-in/library/ie/ms535873(v=vs.85).aspx) object supports a [**performance**](http://msdn.microsoft.com/en-in/library/ie/ff974680(v=vs.85).aspx) property.

Creating Objects to Support Features

Some features cannot be detected in the DOM until they are rendered by the browser. For example, Internet Explorer 9 supports the [**audio**](http://msdn.microsoft.com/en-in/library/ie/ff975061(v=vs.85).aspx) element only when a webpage is displayed in IE9 Standards mode. If a webpage containing an**audio** element is displayed in IE5 (Quirks) mode (or displayed by an earlier version of Internet Explorer), the **audio**element is rendered as an unknown (generic) element.

Based on the previous section, the following code sample might seem sufficient to determine whether a browser supports the [**audio**](http://msdn.microsoft.com/en-in/library/ie/ff975061(v=vs.85).aspx) element.

JavaScript

<!doctype html>

<head>

<title>Simple Audio Support Test</title>

<script type="text/javascript">

function supportsAudio()

{

var d = document;

var o = d.getElementById( "audio1" );

return ( o != null );

}

function checkAudioSupport()

{

var s = ( supportsAudio() == true ) ?

"supports " : "does not support ";

var o = document.getElementById( "dOutput" );

o.outerText = "Your browser " + s + "the audio element.";

}

</script>

</head>

<body>

<audio id="audio1" src="audiofile.mp3" />

<button onclick="checkAudioSupport();">

Click to test audio support.

</button>

<br />

<div id="dOutput">Please click a button</div>

</body>

</html>

This example tries to create an [**audio**](http://msdn.microsoft.com/en-in/library/ie/ff975061(v=vs.85).aspx) element and bases its feature support evaluation on whether the object was created. While this example would correctly indicate that Internet Explorer 9 supports the **audio** element when the page is displayed in IE9 mode, it would incorrectly make the same claim when the same webpage is displayed in IE5 mode. The supportsAudio() function presumes that the ability to get a reference to an object created by an **audio** element is, in fact, an **audio** element that supports music playback. This is a problem because web browsers are supposed to create generic references to HTML elements that they do not support.

JavaScript

function supportsAudio()

{

var o = document.createElement( 'audio' );

return ( o.canPlayType );

}

In addition, this example does not require the webpage to contain an [**audio**](http://msdn.microsoft.com/en-in/library/ie/ff975061(v=vs.85).aspx) element before support for the **audio**element can be determined.

In cases where the DOM does not contain a built-in DOM object or property for the feature you want to detect, use[**createElement**](http://msdn.microsoft.com/en-in/library/ie/ms536389(v=vs.85).aspx) or a similar method to create a temporary object and then verify that the temporary object is the type of object you are interested in.

**Note**  For best results, avoid assumptions about browsers between devices. A desktop browser may support different features than a mobile version of the same browser, which itself may differ from the same browser on a different device. With feature detection, you test all browsers for the features you need.

Searching for DOM Features

The **[hasFeature](http://msdn.microsoft.com/en-in/library/ie/ms536446(v=vs.85).aspx)** method indicates support for individual specifications or sets of features defined by a specification. For example, the following code sample shows how to determine whether a browser supports SVG.

JavaScript

bResult = document.implementation.hasFeature("org.w3c.svg", "1.0")

Specific feature strings are defined by the specification that defines the feature. For example, the World Wide Web Consortium (W3C) Document Object Model (DOM) Level 2 specification supports a variety of feature strings that correspond to modules of the overall specification.

Keep in mind that while the **[hasFeature](http://msdn.microsoft.com/en-in/library/ie/ms536446(v=vs.85).aspx)** method may indicate that a given browser supports a specific feature, it is entirely possible that the browser does not fully support every aspect of the feature. When in doubt, take time to research the implementation of a given feature in popular browsers.

For more information about the status of a given specification or set of features, refer to the resources provided by the organization that sponsors the specification. For example, the [World Wide Web Consortium (W3C) website](http://go.microsoft.com/fwlink/p/?LinkId=220646) provides a[Participation section](http://go.microsoft.com/fwlink/p/?LinkId=220648) that outlines resources regarding the development of W3C standards, including public discussion areas, status updates, and related information.

There are many ways to detect features supported by a web browser. Be sure to use detection techniques that relate directly to the feature that you want to use. For example, it is not appropriate to evaluate the color of a textbox in order to determine the font used to display its content. Associations like this tend to lead to difficulties down the road.

There are many strategies to choose from when developing a modern, device independent website nowadays. How should capabilities of the device or browser be determined? Should the presentation logic be server side or client side? Traditionally, mobile optimization had to happen server side.

Over the last couple of years, Responsive Web Design and tools like [Modernizr](http://modernizr.com/) have become very popular. Recently, combination techniques (often called [RESS](http://www.lukew.com/ff/entry.asp?1392)), where optimization is done both server-side and client-side, has become a trend. The recently launched [WURFL.js](http://wurfl.io/) tool, fits into this category.

In this article, we will look at some basic use cases of how to use WURFL.js to optimize the user experience both in HTML and CSS, and an example of how to choose the right ads to display on different devices. We will also see how WURFL.js is different from, but complements, the popular feature-detection library Modernizr.

### Once Upon A Time, Device Detection

Whether we are using regular expressions in JavaScript, Modernizr or a complete [device-description repository](http://www.smashingmagazine.com/2012/09/24/server-side-device-detection-history-benefits-how-to/) (DDR) for server-side detection, the purpose is usually the same: to give users a better experience. This typically happens at two levels:

* presentation of content and interaction with the service,
* analysis of user behavior to determine usage patterns.

The challenge is to do this in ways that are both scalable, maintainable and, as much as possible, easy to implement. For some projects, the cost and complexity of deploying third-party tools on servers is too high. Yet a low-maintenance solution that lets a website look good and perform well is possible, despite the constant diversification of devices. This is where WURFL.js plays a role, by providing a scalable alternative to traditional server-side device detection, all the while complementing other client-side techniques and tools.

Before diving in, let’s look at the basics.

### Copy, Paste, Done

No registration is required, and WURFL.js can be used at no charge. So, the first thing to do is copy and paste this line of HTML into your page:

<script type='text/javascript' src=“//wurfl.io/wurfl.js"></script>

Both HTTP and HTTPS are supported. If you plan to use the device information provided by the script to make rendering decisions, then you might want to include the script in the <head> element. Otherwise, you can load it asynchronously.

Now that the script is in your HTML page, you can access the WURFL object in JavaScript. The WURFL object looks like this and is ready to use:

{

complete\_device\_name:"Apple iPhone 5",

form\_factor:"Smartphone",

is\_mobile:true

}

The object has three properties:

* complete\_device\_name  
  This is the name by which the device is known — typically, the make and model or a category of devices or a more generic definition.
* form\_factor
  + desktop
  + app
  + tablet
  + smartphone
  + feature phone
  + smart TV
  + robot
  + other non-mobile
  + other mobile
* is\_mobile  
  This is true or false — true if the device is a tablet or other mobile device.

Of course, you can immediately do things like this:

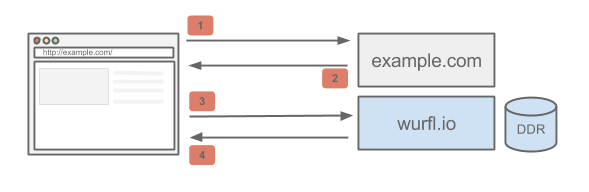
console.log(WURFL);

Or this:

alert(WURFL.complete\_device\_name);

### Under The Hood

Because WURFL.js detects the device based on the User-Agent string and other information provided in the HTTP header, the contents of the JavaScript file will depend on the device. So, you can’t just copy the contents of the file and put it inline in the HTML or combine it with another JavaScript resource.

[](http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2014/04/wurflIOsimple.png)WURFL.js’ basic flow

### A Note On Performance

If you use WURFL.js to make rendering decisions or, for some reason, you need to place the <script> tag inside <head> (without deferring it), then the browser will wait for the script to be downloaded and evaluated before rendering the page. Depending on the use case, this might be the only way; but, for the record, WURFL.js can also be loaded asynchronously to increase rendering performance.

The size of the returned JSON object will be fairly small, varying from 0.5 to 3 or 4 KB, depending on the device. Compared to Modernizr (about 14 KB) and jQuery (96 KB), WURFL.js is arguably light.

### Use Cases

Assuming that you have WURFL.js up and running, let’s look at some cases in which using WURFL.js makes the most sense, either by itself or in conjunction with Modernizr and/or other solutions. To illustrate, we’ll refer to the [WURFL.io website](http://wurfl.io/) itself, which utilizes WURFL.js in multiple ways.

#### OPTIMIZING THE USER EXPERIENCE

When it comes to mobile, responsive and adaptive design and all that, the most common thing to do on a website is improve the user experience for certain device families or form factors. Much can be handled by media queries, of course, but sometimes you need the help of some JavaScript.

When you visit [WURFL.io](http://wurfl.io/) on your laptop, the top section of the page has a video background, some simple parallax scrolling and text that changes dynamically according to the device or browser. It looks very cool on a laptop, but video backgrounds, not to mention parallax scrolling, would not be ideal on a tablet or smartphone, to put it mildly.

Differences in presentation in desktop Safari and iPhone Safari.

We could use Modernizr, of course, or decide whether to implement these features in other ways. But in many cases, knowing the physical device is just as important as — perhaps more important than — knowing whether the browser claims support for a feature. We might encounter a problem whereby the browser claims support, but the support is actually not good enough to make a great user experience.

To avoid these situations, you would use WURFL.js and Modernizer together. Note also that comparing WURFL.js and Modernizr directly is not quite fair. Modernizr detects features claimed by the browser, whereas WURFL.js categorizes the device in different ways. So, if you don’t know whether a particular device or form factor supports a certain browser-detectable feature, then you are better off with Modernizr or a full-fledged[device-detection solution](http://scientiamobile.com/cloud).

However, in this example, we’ll rely on WURFL.js and demand that only non-mobile clients get the video background and parallax scrolling:

/\*video background\*/

if(!WURFL.is\_mobile){

$('#vid').videoBG({

mp4:'assets/Birds\_Animation.mp4.mp4',

ogv:'assets/Birds\_Animation.oggtheora.ogv',

webm:'assets/Birds\_Animation.webmhd.webm'

});

}

/\*The parallax scrolling\*/

window.onscroll = function () {

if (!WURFL.is\_mobile){

heroImage.style[prefixedTransform] = "translate3d(0px," + window.scrollY / 2.3 + "px, 0px)";

herovid.style[prefixedTransform] = "translate3d(0px," + window.scrollY / 1.1 + "px, 0px)";

heroText.style["opacity"] = (1 - ((window.scrollY / 6) / 100));

}

}

The example above simply checks whether the device is mobile (a phone or tablet) and introduces features accordingly. Of course, we could also leverage the more fine-grained WURFL.form\_factor.

#### PUT MORE IN CSS?

The examples above show how to make use of the device’s data in JavaScript. However, we can make the device’s information available in CSS, too. We can assign different styles depending on the device, form factor and whether it is mobile. The first technique we will look at is similar to how Modernizr works. Modernizr adds a certain class to the HTML document depending on whether its test returns true or false.

Let’s say you want some specific behavior defined in the CSS for mobile devices. You would need to add the following JavaScript snippet to your page:

document.documentElement.className += ' ' + (WURFL.is\_mobile ? '' : 'no-') + "mobile";

This will add a class to the html element. For mobile devices, it would say <html class=”is\_mobile”>; for other devices, it would say <html class=”no-is\_mobile”>.

If you know Modernizr, then you are probably familiar with this approach. Your CSS might take the following form:

.mobile #menu a{

padding .5em;

}

.no-mobile #menu a{

padding .1em;

}

This method can be used for all of WURFL.js’ capabilities. However, becausecomplete\_device\_name and form\_factor are not boolean values (like is\_mobile), the CSS part can become quite a headache. A bit more flexibility might come in handy, then. Here is an example using data- attributes:

document.documentElement.setAttribute('data-device\_name', WURFL.complete\_device\_name);

document.documentElement.setAttribute('data-form\_factor', WURFL.form\_factor );

This will put data attributes with WURFL capabilities in the html element. We get several cool features with this method: We can target specific devices, form factors and even groups of devices combined with form factors by using CSS selectors:

html[data-form\_factor = 'Smartphone'] #menu a{

background: green;

}

Thanks to the wildcard [attribute selector](http://www.w3schools.com/css/css_attribute_selectors.asp) \*, we can even match strings:

html[data-device\_name\*='Nokia'] [data-form\_factor = 'Feature Phone'] {

background: yellow;

}

The CSS above will match Nokia feature phones of any model. It also illustrates what the DOM looks like with the two methods implemented — in this case, with an iPhone 5S.

#### HELP WITH BANNER ADS

Many different ad networks are out there, each with its own specialization. Some are good for mobile, others for desktop. Some support text ads, other have ads of fixed size. If you are beyond a beginner’s level in ad networks, then you might want to assume some control over this. WURFL.js can help you make your own decisions or influence the network to make the right decisions for you.

The obvious approach is to ask WURFL.is\_mobile to choose networks or ads that are good for mobile and others that are good for non-mobile.

if(WURFL.is\_mobile){

displayMobileAd();

}else{

displayDesktopAd();

}

Moreover, from a design perspective, being able to fit the sizes and proportions of ads to your breakpoints and to design for different form factors of ads is nice. In the extreme, you could do something like this:

switch(WURFL.form\_factor){

case "Smartphone":

if(WURFL.complete\_device\_name.indexOf("Apple") !=-1){

showAppStoreAds();

}else(

showWebAds();

)

break;

case "Tablet":

showSpecificProportionAds();

break;

case "Feature Phone":

showTextAds();

break;

default:

showGoogleAdwords();

break;

}

JavaScript Object Prototypes

Every JavaScript object has a prototype. The prototype is also an object.

All JavaScript objects inherit their properties and methods from their prototype.

JavaScript Prototypes

All JavaScript objects inherit the properties and methods from their prototype.

Objects created using an object literal, or with new Object(), inherit from a prototype called Object.prototype.

Objects created with new Date() inherit the Date.prototype.

The Object.prototype is on the top of the prototype chain.

All JavaScript objects (Date, Array, RegExp, Function, ....) inherit from the Object.prototype.

Creating a Prototype

The standard way to create an object prototype is to use an object constructor function:

Example

function person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
}

With a constructor function, you can use the **new** keyword to create new objects from the same prototype:

Example

var myFather = new person("John", "Doe", 50, "blue");  
var myMother = new person("Sally", "Rally", 48, "green");

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

function person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

var myFather = new person("John", "Doe", 50, "blue");

var myMother = new person("Sally", "Rally", 48, "green");

myFather.nationality = "English";

document.getElementById("demo").innerHTML =

"My faher is " + myFather.nationality;

</script>

</body>

</html>

Adding Properties and Methods to Objects

Sometimes you want to add new properties (or methods) to an existing object.

Sometimes you want to add new properties (or methods) to all existing objects of a given type.

Sometimes you want to add new properties (or methods) to an object prototype.

*This is the first post in a series on JavaScript. In this post I’m going to explain how JavaScript’s prototype chain works, and how you can use it to achieve inheritance.*

First, it’s important to understand that while JavaScript is an object-oriented language, it is prototype-based and does not implement a traditional class system. Keep in mind that when I mention a *class* in this post, I am simply referring to JavaScript objects and the prototype chain – more on this in a bit.

[We connect highly qualified audiences with highly relevant services, products, and brands.](http://srv.buysellads.com/ads/click/x/GTND423JCABIE5QMCWA4YKQWC6SIP53UCWYD6Z3JCEYI52QYCE7I6K7KC6BD427MF6YDTK3EHJNCLSIZ?segment=placement:wildlyinaccuratecom;)[ads via Carbon](http://carbonads.net/)

Almost everything in JavaScript is an object, which you can think of as sort of like associative arrays – objects contain named properties which can be accessed with obj.propName or obj['propName']. Each object has an internal property called *prototype*, which links to another object. The prototype object has a prototype object of its own, and so on – this is referred to as the *prototype chain*. If you follow an object’s prototype chain, you will eventually reach the core Object prototype whose prototype is null, signalling the end of the chain.

So what is the prototype chain used for? When you request a property which the object does not contain, JavaScript will look down the prototype chain until it either finds the requested property, or until it reaches the end of the chain. This behaviour is what allows us to create “classes”, and implement inheritance.

Don’t worry if this doesn’t make sense yet. To see prototypes in action, let’s take a look at the simplest example of a “class” within JavaScript, which is created with a function object:

**function** **Animal**() {}

**var** animal = **new** Animal();

We can add properties to the Animal class in two ways: either by setting them as *instance properties*, or by adding them to theAnimal prototype.

**function** **Animal**(name) {

*// Instance properties can be set on each instance of the class*

**this**.name = name;

}

*// Prototype properties are shared across all instances of the class. However, they can still be overwritten on a per-instance basis with the `this` keyword.*

Animal.prototype.speak = **function**() {

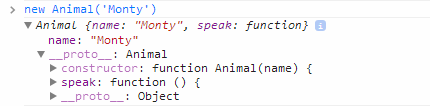
console.log("My name is " + **this**.name);

};

**var** animal = **new** Animal('Monty');

animal.speak(); *// My name is Monty*

The structure of the Animal object becomes clear when we inspect it in the console. We can see that the name property belongs to the object itself, while speak is part of the Animal prototype.



Now let’s look at how we can extend the Animal class to create a Cat class:

**function** **Cat**(name) {

Animal.call(**this**, name);

}

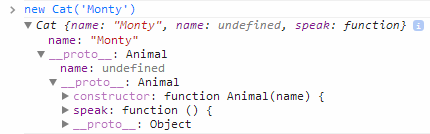
Cat.prototype = **new** Animal();

**var** cat = **new** Cat('Monty');

cat.speak(); *// My name is Monty*

What we are doing here is setting Cat‘s prototype to an instance of Animal, so that Cat inherits all of Animal's properties. We’re also usingAnimal.call to inherit the Animal constructor (sort of like parent or super in other languages). call is a special function which lets us call a function and specify the value of this *within that function*. So when this.name is set inside the Animal constructor, it’s the Cat‘s name property being set, not the Animal‘s.

Let’s take a look at the Cat object to get a better view of what’s going on.



The Cat object has its own name instance property, like we expected. When we look at the object’s prototype we see that it has also inheritedAnimal‘s name instance property as well as the speak prototype property. This is where the prototype chain comes in – when we requestcat.name, JavaScript finds the name instance property and doesn’t bother going down the prototype chain. However when we requestcat.speak, JavaScript has to travel down the prototype chain until it finds the speak property inherited from Animal.

# Prototypes and Prototype Inheritance

In JavaScript, a **prototype** is a property of functions and of objects that are created by constructor functions. The prototype of a function is an object. Its main use is when a function is used as a constructor.

JavaScript

function Vehicle(wheels, engine) {

this.wheels = wheels;

this.engine = engine;

}

In the example above, the prototype of the Vehicle function is the prototype of any object that is instantiated with the Vehicle constructor.

## [Using Prototypes to Add Properties and Methods](javascript:void(0))

You can use the **prototype** property to add properties and methods to objects, even the ones that have already been created:

JavaScript

var testVehicle = new Vehicle(2, false);

Vehicle.prototype.color = "red";

var testColor = testVehicle.color;

The value of testColor is "red".

You can even add properties and methods to predefined objects. For example, you can define a **Trim**method on the **String** prototype object, and all the strings in your script will inherit the method.

JavaScript

String.prototype.trim = function()

{

// Replace leading and trailing spaces with the empty string

return this.replace(/(^\s\*)|(\s\*$)/g, "");

}

var s = " leading and trailing spaces ";

// Displays " leading and trailing spaces (35)"

window.alert(s + " (" + s.length + ")");

// Remove the leading and trailing spaces

s = s.trim();

// Displays "leading and trailing spaces (27)"

window.alert(s + " (" + s.length + ")");

### [Using Prototypes to Derive One Object from Another with Object.create](javascript:void(0))

The **prototype** object can be used to derive one object from another. For example, you can use the[Object.create](http://msdn.microsoft.com/en-us/library/ie/ff925952(v=vs.94).aspx) function to derive a new object Bicycle using the prototype of the Vehicle object we defined earlier (plus any new properties you need).

JavaScript

var Bicycle = Object.create(Object.getPrototypeOf(Vehicle), {

"pedals" :{value: true}

});

The Bicycle object has the properties wheels, engine, color, and pedals, and its prototype is**Vehicle.prototype**. The JavaScript engine finds the pedals property on Bicycle, and it looks up the prototype chain to find the wheels, engine, and color properties on Vehicle.

### [Changing an Object's Prototype](javascript:void(0))

In Internet Explorer 11, you can replace the internal prototype of an object or function with a new prototype by using the [\_\_proto\_\_](http://msdn.microsoft.com/en-us/library/ie/dn342818(v=vs.94).aspx) property. When you use this property, you inherit the properties and methods of the new prototype along with other properties and methods in its prototype chain.

The following example shows how you can change the prototype of an object. This example shows how the object's inherited properties change when you change its prototype.

JavaScript

function Friend() {

this.demeanor = "happy";

}

function Foe() {

this.demeanor = "suspicious";

}

var friend = new Friend();

var foe = new Foe();

var player = new Object();

player.\_\_proto\_\_ = foe;

friend.ally = "Tom";

if (console && console.log) {

console.log(player.demeanor === "happy" ); // Returns false

console.log(player.demeanor === "suspicious"); // Returns true

console.log(player.ally === "Tom"); // Returns false

// Turn the foe to a friend.

player.\_\_proto\_\_ = friend;

console.log(player.demeanor === "happy"); // Returns true

console.log(player.demeanor === "suspicious"); // Returns false

console.log(player.ally === "Tom"); // Returns true

}

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

function person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

person.prototype.name = function() {

return this.firstName + " " + this.lastName

};

var myFather = new person("John", "Doe", 50, "blue");

document.getElementById("demo").innerHTML =

"My father is " + myFather.name();

</script>

</body>

</html>

**AJAX**

AJAX = Asynchronous JavaScript and XML.

AJAX is not a new programming language, but a new way to use existing standards.

AJAX is the art of exchanging data with a server, and updating parts of a web page - without reloading the whole page.

**<!DOCTYPE html>**

**<html>**

**<head>**

**<script>**

**function loadXMLDoc()**

**{**

**var xmlhttp;**

**if (window.XMLHttpRequest)**

**{// code for IE7+, Firefox, Chrome, Opera, Safari**

**xmlhttp=new XMLHttpRequest();**

**}**

**else**

**{// code for IE6, IE5**

**xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");**

**}**

**xmlhttp.onreadystatechange=function()**

**{**

**if (xmlhttp.readyState==4 && xmlhttp.status==200)**

**{**

**document.getElementById("myDiv").innerHTML=xmlhttp.responseText;**

**}**

**}**

**xmlhttp.open("GET","ajax\_info.txt",true);**

**xmlhttp.send();**

**}**

**</script>**

**</head>**

**<body>**

**<div id="myDiv"><h2>Let AJAX change this text</h2></div>**

**<button type="button" onclick="loadXMLDoc()">Change Content</button>**

**</body>**

**</html>**

**Ajax** (also **AJAX**; [/](http://en.wikipedia.org/wiki/Help:IPA_for_English)[ˈeɪdʒæks](http://en.wikipedia.org/wiki/Help:IPA_for_English#Key)[/](http://en.wikipedia.org/wiki/Help:IPA_for_English); short for **asynchronous**[**JavaScript**](http://en.wikipedia.org/wiki/JavaScript)**+**[**XML**](http://en.wikipedia.org/wiki/XML)) is a group of interrelated [Web development](http://en.wikipedia.org/wiki/Web_development) techniques used on the [client-side](http://en.wikipedia.org/wiki/Client-side) to create[asynchronous](http://en.wikipedia.org/wiki/Asynchronous_I/O) [Web applications](http://en.wikipedia.org/wiki/Web_application). With Ajax, Web applications can send data to and retrieve from a [server](http://en.wikipedia.org/wiki/Web_server) asynchronously (in the background) without interfering with the display and behavior of the existing page. Data can be retrieved using the [XMLHttpRequest](http://en.wikipedia.org/wiki/XMLHttpRequest" \o "XMLHttpRequest) [object](http://en.wikipedia.org/wiki/Object_(computer_science)). Despite the name, the use of XML is not required ([JSON](http://en.wikipedia.org/wiki/JavaScript_Object_Notation) is often used in the [AJAJ](http://en.wikipedia.org/wiki/AJAJ" \o "AJAJ)variant), and the requests do not need to be asynchronous.[[4]](http://en.wikipedia.org/wiki/Ajax_%28programming%29#cite_note-wrox-4)

Ajax is not a single technology, but a group of technologies. [HTML](http://en.wikipedia.org/wiki/Hypertext_Markup_Language) and [CSS](http://en.wikipedia.org/wiki/Cascading_Style_Sheets) can be used in combination to mark up and style information. The [DOM](http://en.wikipedia.org/wiki/Document_Object_Model) is accessed with JavaScript to dynamically display – and allow the user to interact with – the information presented. JavaScript and the XMLHttpRequest object provide a method for exchanging data asynchronously between browser and server to avoid full page reloads.

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

## WHAT IS JSON?

[JSON](http://en.wikipedia.org/wiki/JSON) is short for **JavaScript Object Notation**, and is a way to store information in an organized, easy-to-access manner. In a nutshell, it gives us a human-readable collection of data that we can access in a really logical manner.

JSON is built on two structures:

* A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
* An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

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

In JSON, they take on these forms:

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

**JSONP** or "[JSON](http://en.wikipedia.org/wiki/JSON) with padding" is a communication technique used in [JavaScript](http://en.wikipedia.org/wiki/JavaScript) programs running in web browsers to request data from a server in a different domain, something prohibited by typical web browsers because of the [same-origin policy](http://en.wikipedia.org/wiki/Same-origin_policy). JSONP takes advantage of the fact that browsers do not enforce the same-origin policy on<script> tags.

Note that for JSONP to work, a server must know how to reply with JSONP-formatted results. JSONP does not work with JSON-formatted results. The JSONP parameters passed as arguments to a script are defined by the server.

## Padding

While the padding (prefix) is *typically* the name of a callback function that is defined within the execution context of the browser, it may also be a variable assignment, an if statement, or any other JavaScript statement. The response to a JSONP request is not JSON and is not parsed as JSON; the returned payload can be any arbitrary JavaScript expression, and it does not need to include any JSON at all. But conventionally, it is a JavaScript fragment that invokes a function call on some JSON-formatted data.

Said differently, the typical use of JSONP provides cross-domain access to an existing JSON API, by wrapping a JSON payload in a function call

## Cross domain requests

For security reasons, the default behavior of a web browser is to block all queries that are going to a domain that is different from the website they are sent from. So when using an external HTTP-based search API, all your queries should be blocked because they are sent to an external domain. There are two methods to call an external API from the browser:

The [JSONP](http://en.wikipedia.org/wiki/JSONP) approach is a workaround that consists of calling an external API  with a DOM  <script>  tag. The  <script> tag is allowed to load content from any domains without security restrictions. The targeted API needs to expose a HTTP GET endpoint and return JavaScript code instead of the regular JSON data. You can use this jQuery code to dynamically call a JSONP URL: