Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

Chapter 10

JavaScript 3:

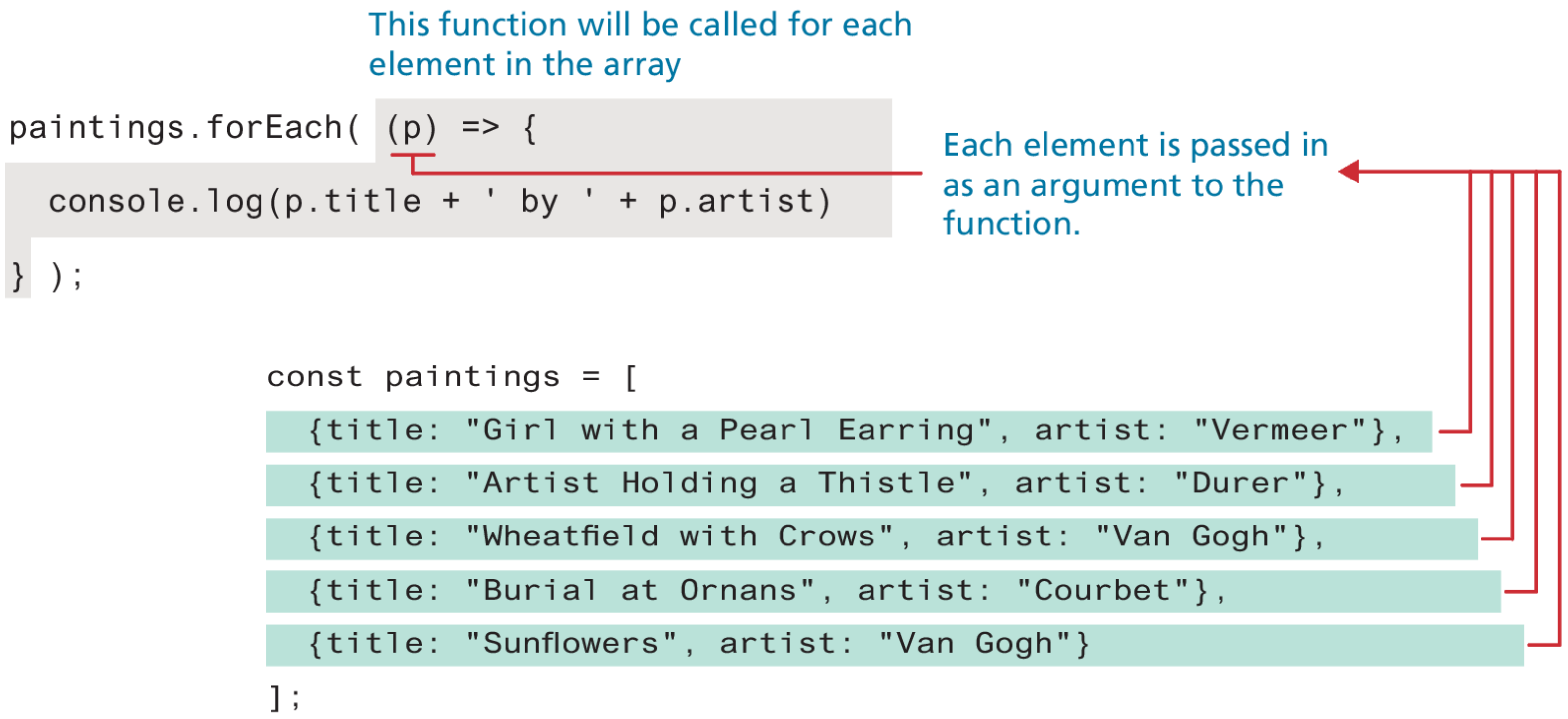
Additional Features

Pearson

Array Functions

- **forEach()** iterate through an array
- **find()** find the *first* object whose property matches some condition
- **filter()** find all matches whose property matches some condition
- **map()** is similar manner to filter except it creates a new array of the same size whose values have been transformed by the passed function
- **reduce()** reduces an array into a single value
- **sort()** sorts a one-dimensional array

Array forEach()



This function will be called for each element in the array

```
paintings.forEach( (p) => {
  console.log(p.title + ' by ' + p.artist)
} );
```

Each element is passed in as an argument to the function.

```
const paintings = [
  {title: "Girl with a Pearl Earring", artist: "Vermeer"},
  {title: "Artist Holding a Thistle", artist: "Durer"},
  {title: "Wheatfield with Crows", artist: "Van Gogh"},
  {title: "Burial at Ornans", artist: "Courbet"},
  {title: "Sunflowers", artist: "Van Gogh"}
];
```

Array find()

One of the more common coding scenarios with an array of objects is to find the *first* object whose property matches some condition. This can be achieved via the **find**() method of the array object, as shown below.

```
const courbet = paintings.find( p => p.artist === 'Courbet' );
console.log(courbet.title); // Burial at Ornans
```

Like the **forEach**() method, the **find**() method is passed a function

This function must return either true (if condition matches) or false (if condition does not match). In the example code above, it returns the results of the conditional check on the artist name.

Returns the first element in the provided array that satisfies the provided testing function. If no values satisfy the testing function, undefined is returned.

Array filter()

If you were interested in finding all matches you can use the **filter()** method, as shown in the following:

```
// vangoghs will be an array containing two painting objects
const vangoghs = paintings.filter(p => p.artist === 'Van Gogh’);
```

Since the function passed to the filter simply needs to return a true/false value, you can make use of other functions that return true/false. For instance, you could perform a more sophisticated search using regular expressions.

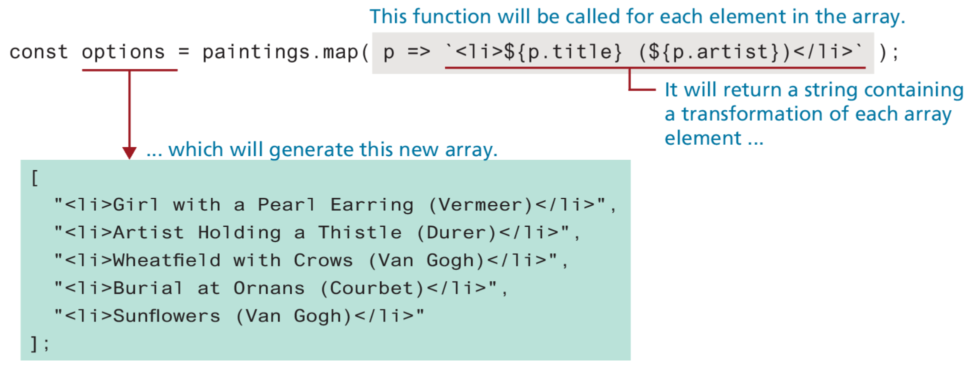
Array map()

The **map()** function operates in a similar manner except it creates a new array of the same size but whose values have been transformed by the passed function.

Listing 10.2 shows map using DOM nodes. Figure 10.2 uses strings.

```
// create array of DOM nodes
const options = paintings.map( p => {
   let item = document.createElement("li");
   item.textContent = `${p.title} (${p.artist})`;

   return item;
});
```

LISTING 10.2 Using the map() function


This function will be called for each element in the array.
It will return a string containing a transformation of each array element ...
... which will generate this new array.

```
const options = paintings.map( p => `<li>${p.title} (${p.artist})</li>` );
```

```
[
  "<li>Girl with a Pearl Earring (Vermeer)</li>",
  "<li>Artist Holding a Thistle (Durer)</li>",
  "<li>Wheatfield with Crows (Van Gogh)</li>",
  "<li>Burial at Ornans (Courbet)</li>",
  "<li>Sunflowers (Van Gogh)</li>"
];
```

Reduce

The **reduce**() function is used to reduce an array into a single value. Like the other array functions in this section, the **reduce**() function is passed a function that is invoked for each element in the array.

This callback function takes up to four parameters, two of which are required: the previous accumulated value and the current element in the array.

For instance, the following example illustrates how this function can be used to sum the **value** property of each painting object in our sample paintings array:

```
let initial = 0;
const total = paintings.reduce( (prev, p) => prev + p.value, initial);
const myarr = [10, 20, 30, 40];

// 0 + 10 + 20 + 30 + 40
const iniziale = 0;
const somma = myarr.reduce(
                  (accumulatore, corrente) => accumulatore + corrente,
                  iniziale,
              );

console.log(somma);
// Output: 100
```

Sort

sort() function sorts in ascending order (after converting to strings if necessary)

```
const names = ['Bob', 'Sue', 'Ann', 'Tom', 'Jill'];
const sortedNames = names.sort();
// sortedNames contains ["Ann", "Bob", "Jill", "Sue", "Tom"]
```

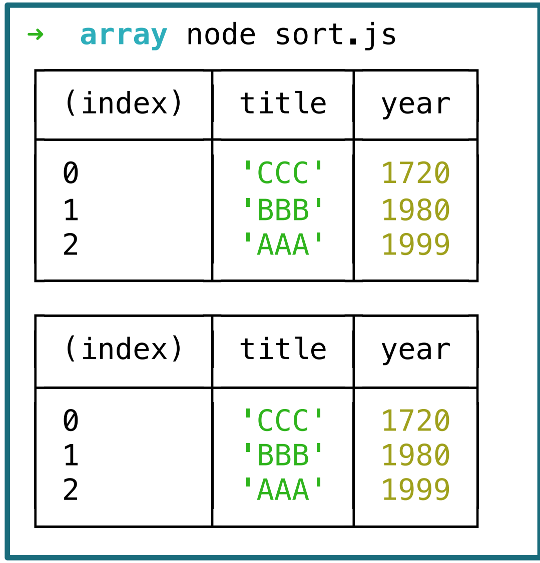
If you need to sort an array of objects based on one of the object properties, you will need to supply the sort() method with a compare function that returns either 0, <0, or >0, depending on whether two values are equal (0), the first value is greater than the second (>0), or the first value is less than the second (<0).

Custom sort

```
const sortedPaintingsByYear = paintings.sort( function(a,b) {
   if (a.year < b.year)
      return -1;
   else if (a.year > b.year)
      return 1;
   else
      return 0;
} );

// more concise version using ternary operator and arrow syntax
const sorted2 = paintings.sort( (a,b) => a.year < b.year? -1: 1 );
```

LISTING 10.3 Sorting an array based on the properties of an object

```javascript
const paintings = [
  {title: "AAA", year: 1999},
  {title: "BBB", year: 1980},
  {title: "CCC", year: 1720}
];

const sortedPaintingsByYear = paintings.sort( function(a,b) {
if (a.year < b.year) return -1;
else if (a.year > b.year) return 1;
else return 0;
} );

// more concise version using ternary operator and arrow syntax
const sorted2 = paintings.sort( (a,b) => a.year < b.year? -1: 1 );
console.table(sorted2);
// anche meglio...
const sorted3 = paintings.sort((a, b) => a.year - b.year);
console.table(sorted3);
```

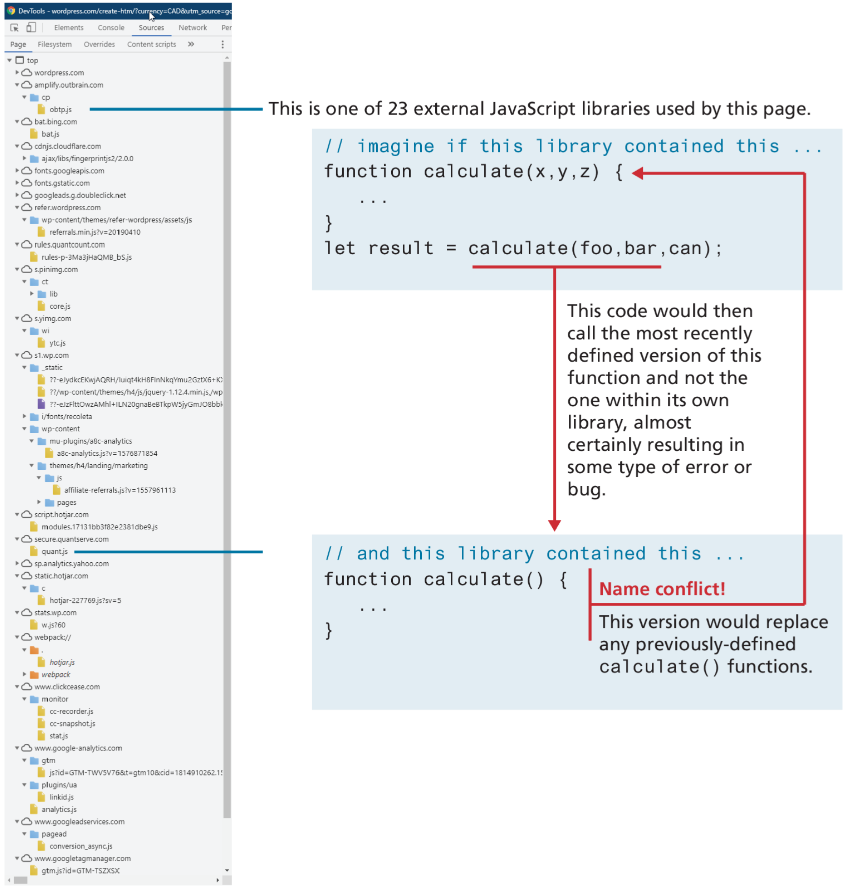


```
➜  array node sort.js
┌─────────┬───────┬──────┐
│ (index) │ title │ year │
├─────────┼───────┼──────┤
│ 0       │ 'CCC' │ 1720 │
│ 1       │ 'BBB' │ 1980 │
│ 2       │ 'AAA' │ 1999 │
└─────────┴───────┴──────┘
┌─────────┬───────┬──────┐
│ (index) │ title │ year │
├─────────┼───────┼──────┤
│ 0       │ 'CCC' │ 1720 │
│ 1       │ 'BBB' │ 1980 │
│ 2       │ 'AAA' │ 1999 │
└─────────┴───────┴──────┘
```

Name Conflicts

As shown in Figure 10.5, complex contemporary JavaScript applications might contain hundreds of literals defined in dozens of .js files, so some way of preventing name conflicts becomes especially important

wordpress.com Home Page (Jan 2020)



```
top
  wordpress.com
  amplify.outbrain.com
    cp
      obtp.js
  bat.bing.com
    bat.js
  cdnjs.cloudflare.com
    ajax/libs/fingerprintjs2/2.0.0
  fonts.googleapis.com
  fonts.gstatic.com
  googleads.g.doubleclick.net
  refer.wordpress.com
    wp-content/themes/refer-wordpress/assets/js
      referrals.min.js?v=20190410
  rules.quantcount.com
    rules-p-3Ma3jHaQMB_bS.js
  s.pinimg.com
    ct
      lib
      core.js
  s.yimg.com
    wi
      ytc.js
  s1.wp.com
    _static
      ??-eJydkcEKwjAQRH/Iuiqt4kH8FInNkqYmu2GztX6+K
      ??/wp-content/themes/h4/js/jquery-1.12.4.min.js,/wp
      ??-eJzFlttOwzAMhl+ILN20gnaBeBTkpW5jyGmJO8bbk
    i/fonts/recoleta
    wp-content
      mu-plugins/a8c-analytics
        a8c-analytics.js?v=1576871854
      themes/h4/landing/marketing
        js
          affiliate-referrals.js?v=1557961113
        pages
  script.hotjar.com
    modules.17131bb3f82e2381dbe9.js
  secure.quantserve.com
    quant.js
  sp.analytics.yahoo.com
  static.hotjar.com
    c
      hotjar-227769.js?sv=5
  stats.wp.com
    w.js?60
  webpack://
    .
      hotjar.js
    webpack
  www.clickcease.com
    monitor
      cc-recorder.js
      cc-snapshot.js
      stat.js
  www.google-analytics.com
    gtm
      js?id=GTM-TWV5V76&t=gtm10&cid=1814910262.1
    plugins/ua
      linkid.js
    analytics.js
  www.googleadservices.com
    pagead
      conversion_async.js
  www.googletagmanager.com
    gtm.js?id=GTM-TSZXSX
```

This is one of 23 external JavaScript libraries used by this page.

```
// imagine if this library contained this ...
function calculate(x,y,z) {
   ...
}
let result = calculate(foo,bar,can);
```

This code would then call the most recently defined version of this function and not the one within its own library, almost certainly resulting in some type of error or bug.

```
// and this library contained this ...
function calculate() {
   ...
}
```

Name conflict!

This version would replace any previously-defined `calculate()` functions.

Modules

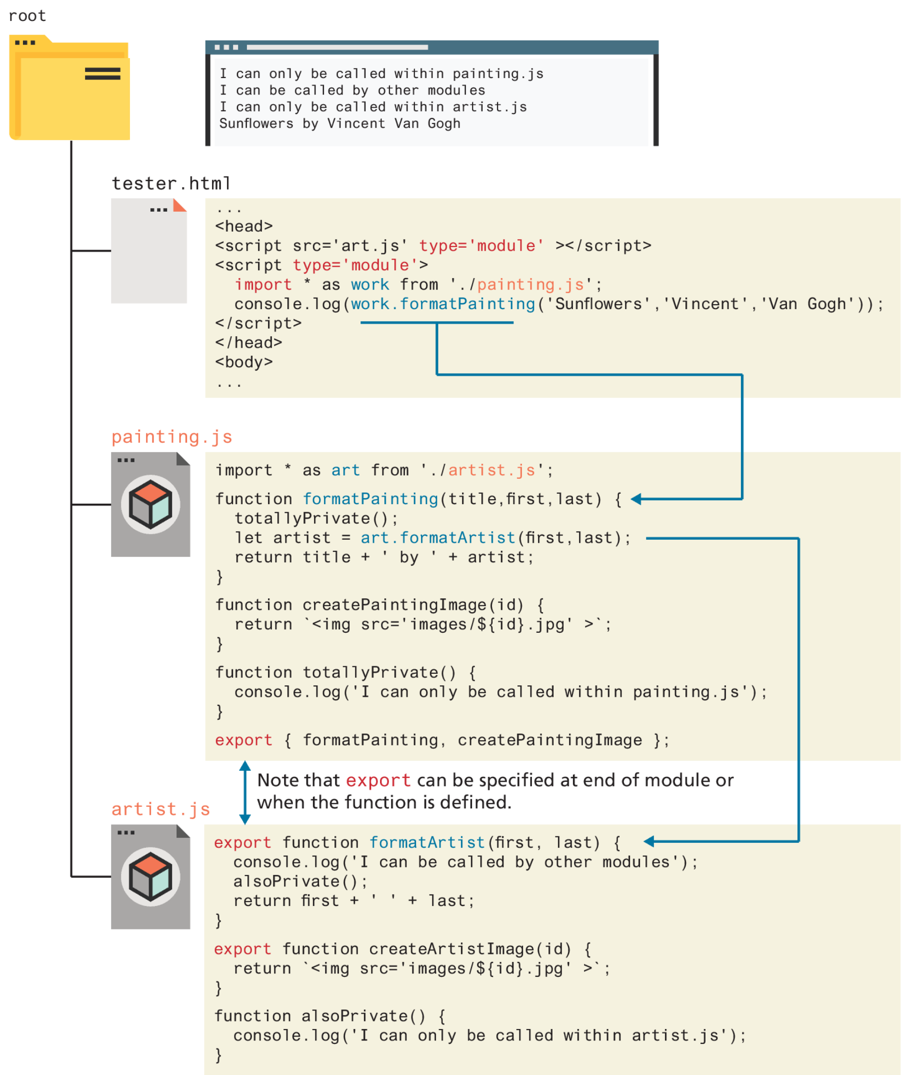
An ES6 **module** is simply a file that contains JavaScript. Unlike a regular JavaScript external file, literals defined within the module are scoped to that module.

You do have to tell the browser that a JavaScript file is a module and not just a regular external JavaScript file within the <script> element. This is achieved via the **type** attribute as shown in the following:

```
<script src="art.js" type="module"></script>
```

Within a module, any literals are private to that module. To make content in the module file available to other scripts outside the module, you have to make use of the **export** keyword.

Modules



```
I can only be called within painting.js
I can be called by other modules
I can only be called within artist.js
Sunflowers by Vincent Van Gogh
```

tester.html

```
...
<head>
<script src='art.js' type='module' ></script>
<script type='module'>
  import * as work from './painting.js';
  console.log(work.formatPainting('Sunflowers','Vincent','Van Gogh'));
</script>
</head>
<body>
...
```

painting.js

```
import * as art from './artist.js';

function formatPainting(title,first,last) {
  totallyPrivate();
  let artist = art.formatArtist(first,last);
  return title + ' by ' + artist;
}

function createPaintingImage(id) {
  return `<img src='images/${id}.jpg' >`;
}

function totallyPrivate() {
  console.log('I can only be called within painting.js');
}

export { formatPainting, createPaintingImage };
```

Note that export can be specified at end of module or when the function is defined.

artist.js

```
export function formatArtist(first, last) {
  console.log('I can be called by other modules');
  alsoPrivate();
  return first + ' ' + last;
}

export function createArtistImage(id) {
  return `<img src='images/${id}.jpg' >`;
}

function alsoPrivate() {
  console.log('I can only be called within artist.js');
}
```

Pearson

Export

- Ogni modulo può esportare zero o più identificatori usando la parola chiave **export**

```
//Contenuto del file mod1.js

//Funzione privata del modulo
function moltiplica(a, b) {
  return a * b;
}

export function quadrato(x) {
  return moltiplica(x, x);
}
export const VOTO_MAX = 30;
```

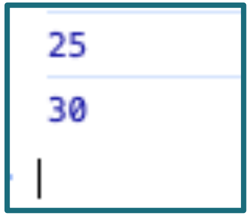
Import

- Gli identificatori possono essere importati con la parola chiave **import**

```
// File mod2.js

import {quadrato} from './mod1.js';

console.log(quadrato(5));
```

```
// File mod3.js

import {VOTO_MAX as VM} from './mod1.js';

console.log(VM);
```

```html
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Moduli JS</title>
<script type="module" src="./mod2.js"></script>
<script type="module" src="./mod3.js"></script>
</head>
<body>
Guarda la console.
</body>
</html>
```



```
25
30
```

namespace import

- Importo tutti gli identificatori esportati da un altro modulo. Diventano proprietà di un oggetto di cui scelgo il nome.

```
// File mod4.js

import * as nomeLocale from './mod1.js';

console.log(nomeLocale.quadrato(7));
console.log(nomeLocale.VOTO_MAX);
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Moduli JS</title>
<script type="module" src="./mod4.js"></script
</head>
<body>
Guarda la console.
</body>
</html>
```

export

- Gli identificatori da esportare possono essere indicati tutti alla fine del file

```
// File mod1bis.js
// analogo a mod1.js

// Funzione privata del modulo
function moltiplica(a, b) {
  return a * b;
}

function quadrato(x) {
  return moltiplica(x, x);
}

const VOTO_MAX = 30;

export {quadrato, VOTO_MAX};
```

```
// File mod5.js

import * as x from './mod1bis.js';

console.log(x.quadrato(11));
console.log(x.VOTO_MAX);
```

```
// File mod5bis.js

import {quadrato, VOTO_MAX} from
'./mod1bis.js';

console.log(quadrato(11));
console.log(VOTO_MAX);
```

export

- ... e possono essere rinominati

```
// File mod1tris.js

// Funzione privata del modulo
function moltiplica(a, b) {
  return a * b;
}

function quadrato(x) {
  return moltiplica(x, x);
}

const VOTO_MAX = 30;

export {quadrato as q,
        VOTO_MAX as VMAX};
```

```
// File mod6.js

import * as x from './mod1tris.js';

console.log(x.q(11));
console.log(x.VMAX);
```

```
// File mod6bis.js

import {q, VMAX} from './mod1tris.js';

console.log(q(11));
console.log(VMAX);
```

```
// File mod6tris.js

import {q as mioq, VMAX as miovmax} from
'./mod1tris.js';

console.log(mioq(11));
console.log(miovmax);
```

Default export e import

- Ogni modulo può avere un'entità esportata per default
  - non mescolare esportazione default e con nome

```
// File modA.js

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

```
// File modB.js, no parentesi

import somma from './modA.js';

console.log(somma(10, 20));
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Moduli JS</title>
<script type="module" src="./modB.js"></script>
</head>
<body>
Guarda la console.
</body>
</html>
```

Web Speech API

The Web Speech API provides a mechanism for turning text into speech (sounds) and for turning speech (microphone input) into text.

To verbalize a string of text, you can simply make use of the **SpeechSynthesisUtterance** and **speechSynthesis** objects:

```
const utterance = new SpeechSynthesisUtterance('Hello world');

speechSynthesis.speak(utterance);
```

Some browsers provide different voices: for instance, U.S. male, U.S. female, U.K. male, etc. You can also adjust the speed and pitch of the speech.

```html
<!DOCTYPE html>
<html>
<head>
<meta charset='utf-8'>
<title>Esempio speech API</title>
<meta name='viewport' content='width=device-width, initial-scale=1'>
<script>
function f(){
  const frase = new SpeechSynthesisUtterance('La progettazione Web è meravigliosa');
  const synth = window.speechSynthesis;
  synth.speak(frase);
}
function inizializza(){
  const b = document.getElementById('btn');
  b.addEventListener('click', f);
}
document.addEventListener('DOMContentLoaded', inizializza);
</script>
</head>
<body>
<button id="btn">Parla</button>
</body>
</html>
```

GeoLocation

The **Geolocation API** provides a way for JavaScript to obtain the user's location (accuracy/availability dependent on permission and device)

```
if (navigator.geolocation) {
   navigator.geolocation.getCurrentPosition(hav
   eLocation, geoError);
} else {
  // geolocation not supported or accepted
  ...
}
```

```
function haveLocation(position) {
   const latitude = position.coords.latitude;
   const longitude = position.coords.longitude;
   const altitude = position.coords.altitude;
   const accuracy = position.coords.accuracy;
   // now do something with this information
   ...
}
function geoError(error) { ... }
```

LISTING 10.13 Sample GeoLocation API usage

Using External APIs

An **external API** refers to objects with events and properties that perform a specific task that you can use in your pages.

Unlike browser APIs, these external APIs are **not** built into the browser but are external JavaScript libraries that need to be downloaded or referenced and added to a page via a <script> tag.

In this section, we will look at two of the most popular ones: the Google Maps API and the plotly API.

Google Maps

The Google Maps code used in the 1st edition of this book (2014) no longer worked by the time of the second edition (2017). That code no longer works now at the time of writing (2020). Hopefully, when you use this edition, the Google Maps code still works—but it might not!

- The point here is that external APIs are an externality, meaning that you have no control over them and that change over time should be expected with them.

- Use the latest Documentation from the API.

Overview | Maps JavaScript API | Google Developers

Google Maps (ii)

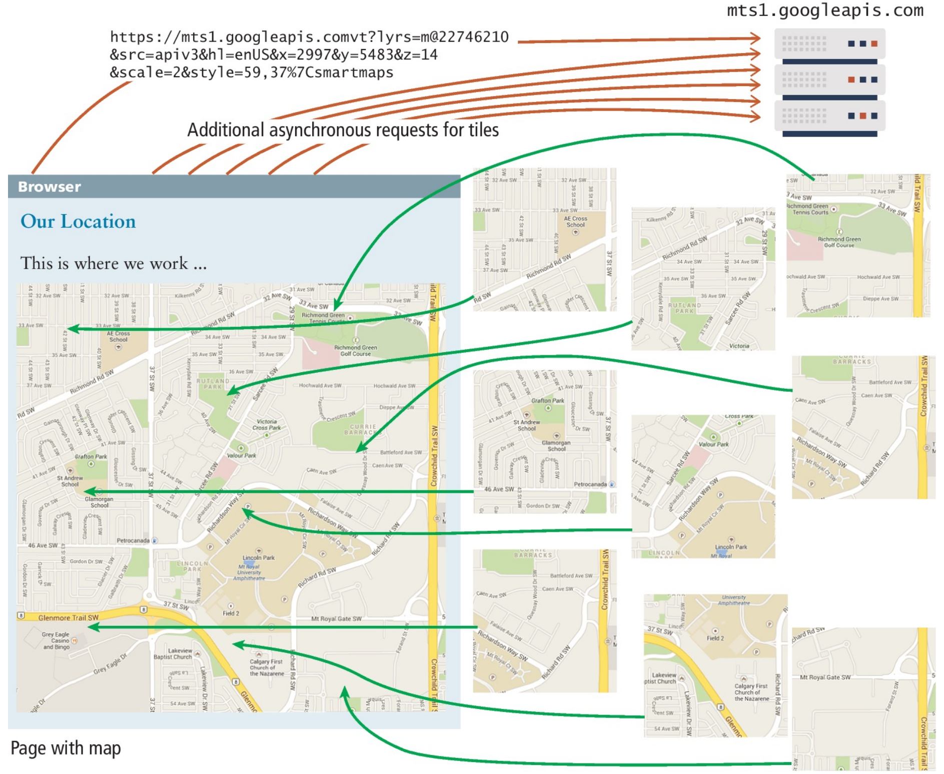
Notice that the API is made available to your page by referencing it in a **<script>** tag.

Our **initMap**() function creates a map object via the **google.maps.Map()** function constructor, which is defined within the library downloaded in our <script> element.

```
<head>
…
<style>
    #map {
        height: 500px;
    }
</style>
<script>
function initMap() {
    const map = new
    google.maps.Map(document.querySelector('#map'),
{
    center: {lat: 51.011179,
                lng: -114.132866},
                zoom: 14
    });
}
</script>
<script
src="https://maps.googleapis.com/maps/api/js?key=YO
UR-API-KEY&callback=initMap" async defer>
</script>
</head>
<body>
    Populating a Google Map
    <div id="map"></div>
</body>
</html>
```

LISTING 10.14 Webpage to output map centered on a location

Google Maps at work


mts1.googleapis.com
https://mts1.googleapis.comvt?lyrs=m@22746210
&src=apiv3&hl=enUS&x=2997&y=5483&z=14
&scale=2&style=59,37%7Csmartmaps
Additional asynchronous requests for tiles
Browser
Our Location
This is where we work ...
Page with map

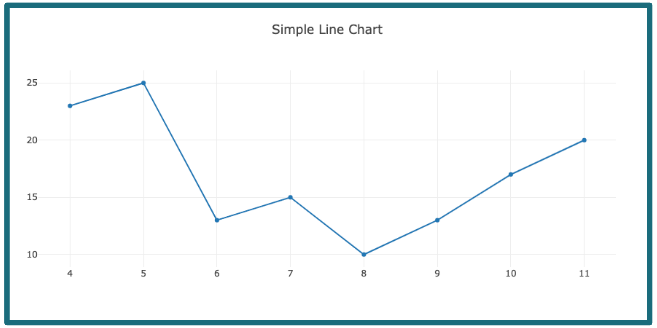
Charting with Plotly.js

Charting is a common need for many websites. This section makes use of Plotly, which is open-source and available in a variety of other languages besides JavaScript.

Creating a simple chart is quite straightforward. Simply include the library, add an empty <div> element that will contain the chart, and then make use of the **newPlot()** method, as shown in the following slide:

Charting with Plotly.js (ii)

```
<script>window.addEventListener("load", function() {
        const data = [
                                        { x: [4,5,6,7,8,9,10,11],
                                         y: [23,25,13,15,10,13,17,20]
                                        }
                                ];
        const layout = { title:'Simple Line Chart' };
        const options = { responsive: true };
        Plotly.newPlot("chartDiv", data, layout, options);
});
</script>
<script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
<div id="chartDiv"></div>
```


Simple Line Chart