Working with Dates

Lesson Objectives

When you complete this lesson, you will be able to:

- *use various methods to get the date and time.*
- *set the date and time.*
- *compare and set dates relative to the present.*
- *convert strings to dates.*

Earlier, you learned how to create a unique key for your to-do items in *Local Storage* using the *Date* object and the `getTime()` method, which returns the current date and time represented as the number of milliseconds since 1970.

It's time we return to the *Date* object, and its methods, and explore the power of this object further. The Date object has many of methods for getting and setting the date and/or the time. We'll focus on a few of the most useful methods.

Dates and times are a bit tricky to work with on the computer. If all you care about is the date right now on your own computer, it's fairly straightforward, but if you're creating a web page on the internet, then you'll have users from all around the world in different time zones visiting your site. Working with dates and times in code can be tricky when you're considering all time zones or, say, converting the way we write dates in the US to the way people write dates in other countries. Unfortunately, there's no one best way to tackle this stuff, so we'll look at the methods we have available, and you can experiment with your own code to see what works best for you.

What's the Date and Time Right Now?

Let's start with a web page that displays the current date and time when you load the page. Create a new HTML file as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<title>Dates</title>
  <meta charset="utf-8">
```

```
  <script src="dates.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
    span {
      font-weight: bold;
    }
  </style>
</head>
<body>
    <div>
      Right now, the date and time is:
      <span id="datetime"></span>
    </div>
</body>
</html>
```

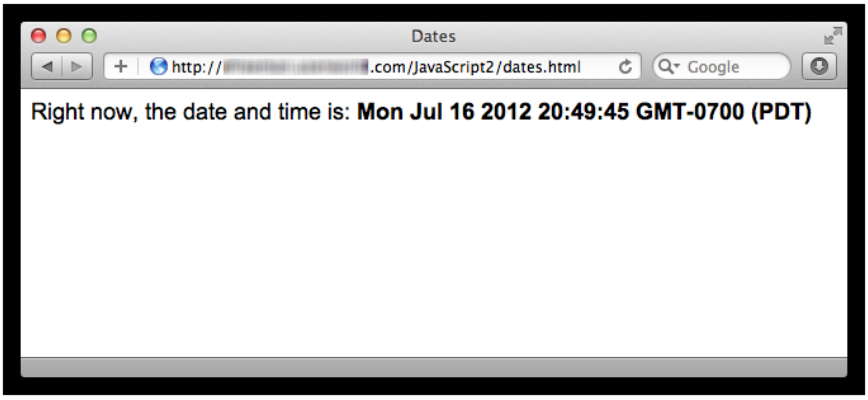
Save it as *dates.html* in your work folder. Next, we'll create a new JavaScript file:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var now = new Date();
    datetime.innerHTML = now;
}
```

Now, save this file as *date.js* in your work folder. Once you've saved it, open *dates.html* in a browser. You see a page with the current date and time. Because the JavaScript is running in the browser that's on your computer, you'll see the date and time for where you are.



In this code, we create a new Date object, and then set the content of the "`datetime`" `<span>` element in the HTML to that date. By default, the value that you get is a string that shows you both the date and time.

OBSERVE:

```
datetime.innerHTML = now;
```

Here, we set the value of a property, innerHTML, that expects a String, not a Date. So JavaScript automatically calls the Date method `toString()` on the now Date object to convert it to a String. Try changing the code to:

OBSERVE:

```
datetime.innerHTML = now.toString();
```

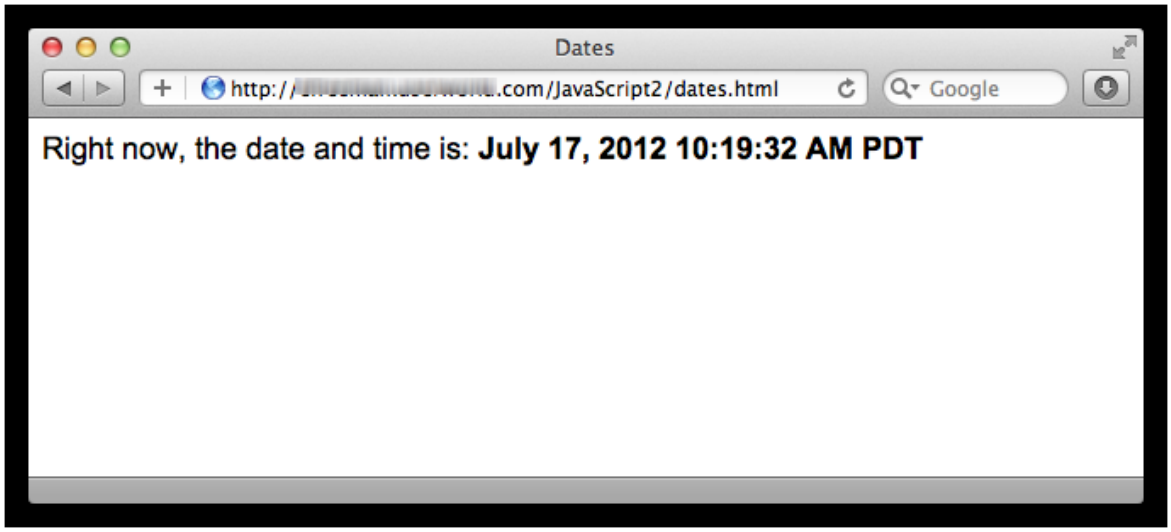
You get exactly the same result.

Other Methods for Getting the Date and Time

Along with `toString()`, there are several other methods for getting a string that represents the Date in a readable format, including: `toDateString()`, `toLocaleDateString()`, `toTimeString()`, `toLocaleTimeString()` and `toLocaleString()`. Try each of these by changing the code in the JavaScript in dates.js.

- **`toString()`**: Displays the date and time as a string using the local time zone.
- **`toDateString()`**: Displays the date as a string using the local time zone.
- **`toLocaleDateString()`**: Displays the date as a string using the local time zone, formatted using local conventions.
- **`toTimeString()`**: Displays the time as a string using the local time zone.
- **`toLocaleTimeString()`**: Displays the time as a string using the local time zone, formatted using local conventions.
- **`toLocaleString()`**: Displays the date and time as a string using the local time zone, formatted using local conventions.

Notice the differences in these various methods of creating a string from the Date object. For instance, compare the way the date string is displayed using `toLocaleString()` with how it was displayed using `toString()` (above):



Dates and Time Zones

In the previous example using `toString()`, in the date and time information displayed in the web page, you can see the time zone information: GMT-0700 (PDT)? I'm in the Pacific time zone in the United States, so my time is currently 7 hours behind the GMT (Greenwich Mean Time) measured from Greenwich, in London, England. GMT is similar to the Coordinated Universal Time (UTC), which is now the worldwide standard for measuring time. For most purposes (and certainly our purposes here), GMT and UTC are equivalent.

You can use the Date methods, `getTimezoneOffset()` and `toUTCString()` to help figure out differences in the local time where you are (or where someone using your web page is), and the UTC time. Let's update *dates.js* to use these methods to show how many hours we are from UTC time:

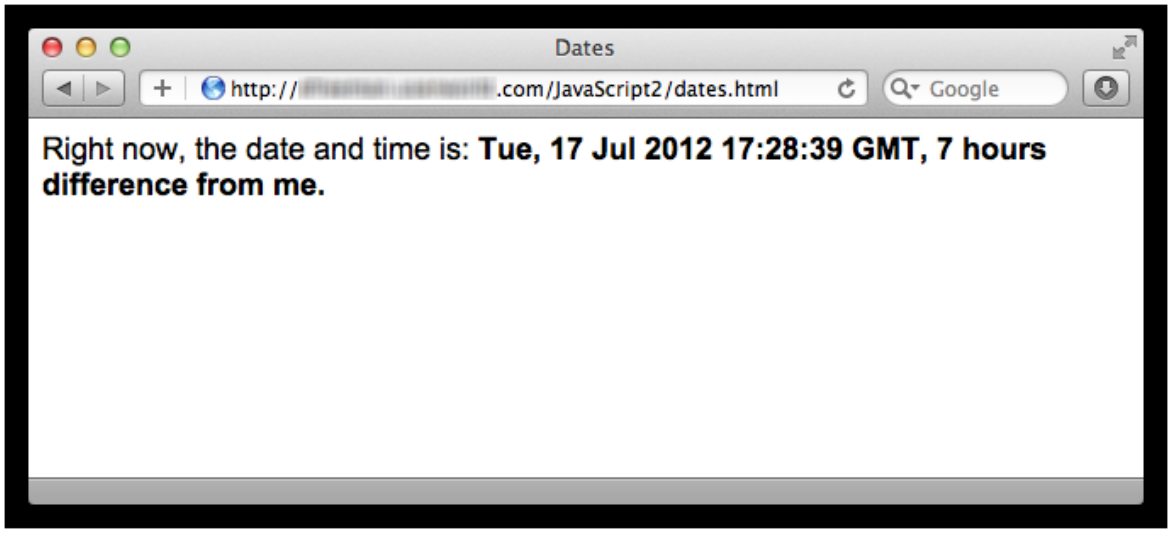
CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var now = new Date();
    datetime.innerHTML = now.toUTCString();
    var hoursDiff = (now.getTimezoneOffset()) / 60;
    datetime.innerHTML += ", " + hoursDiff + " hours difference from
me.";
}
```

Save the changes to *dates.js*, and open ore refresh *dates.html* in your browser. Now you see the current time expressed in universal time (or GMT), and the number of hours difference from your local time to GMT.

The `getTimezoneOffset()` method returns the difference in minutes, not hours, so here, we divide by 60 to get the hours. Although you may want the time in minutes, because in some places the time difference from GMT does not occur on the hour. For instance, some places will be several hours plus a half hour different from GMT. I'm exactly 7 hours behind GMT, so my result is:



Setting a Date and Time

So far, all we've done with Date is get the current date, and convert it to a String for display in a web page. But what if you want to create a specific date?

The `Date()` constructor function creates a date that represents *now*, if you don't pass in any arguments. But you can also create specific dates, by passing in the year, month, day, hours, minutes, seconds, and even milliseconds of a specific date and time you want. Let's create the Date that represents New Year's Day, 2050. Modify *dates.html* as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<title>Dates</title>
  <meta charset="utf-8">
  <script src="dates.js"></script>
  <style>
```

```
    body {
      font-family: Arial, sans-serif;
    }
    span {
      font-weight: bold;
    }
  </style>
</head>
<body>
    <div>
      Right now, tThe date and time is:
      <span id="datetime"></span>
    </div>
</body>
</html>
```

Save these changes, and now modify *dates.js* as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var now = new Date();
    datetime.innerHTML = now.toUTCString();
    var hoursDiff = (now.getTimezoneOffset()) / 60;
    datetime.innerHTML += ", " + hoursDiff + " hours difference from
me.";
    var nyd = new Date(2050, 0, 1);
    datetime.innerHTML = nyd.toString();
}
```

Save your changes to *dates.js*, and open or refresh *dates.html* in the browser.

OBSERVE:

```
var nyd = new Date(2050, 0, 1);
```

In this example, we created a Date object for New Year's Day, 2050, by passing in the year 2050, the month 0 (for January), and the day 1 (for the 1st of January). Notice that the month is 0, not 1. Why? Because in JavaScript, the months start at 0 and go to 11. That's a little weird, but that's the way it works. Days, however, do start at 1, and go up to 31 depending on the month.

Because we didn't supply any arguments for the time, JavaScript assumed we wanted 12:00AM (midnight) on January 1, 2050 (which is perfect for celebrating the New Year!). We could have supplied time arguments, like this:

OBSERVE:

```
var nyd = new Date(2050, 0, 1, 0, 1, 0);
```

...where 0 is the hour (midnight), 1 is the minute (1 minute after midnight), and 0 is the seconds. Try other dates and times. Notice that if you want to specify a time, you *must* also specify a date. That is, while all the arguments to `Date()` are optional, you must supply them in order, and you can't skip any. So, if you want to supply minutes, you must also supply a year, month, day, and hour, but you can skip the seconds and milliseconds.

Try changing the code so you display the date and time using `toLocaleString()` instead. Modify *dates.js* as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var nyd = new Date(2050, 0, 1, 0, 1, 0);
    datetime.innerHTML = nyd.toLocaleString();
}
```

Save these changes, an open or refresh *dates.html* in the browser. Compare your result with the previous result using `toString()`. Which do you like better?

Setting Date and Time Elements Separately with Methods

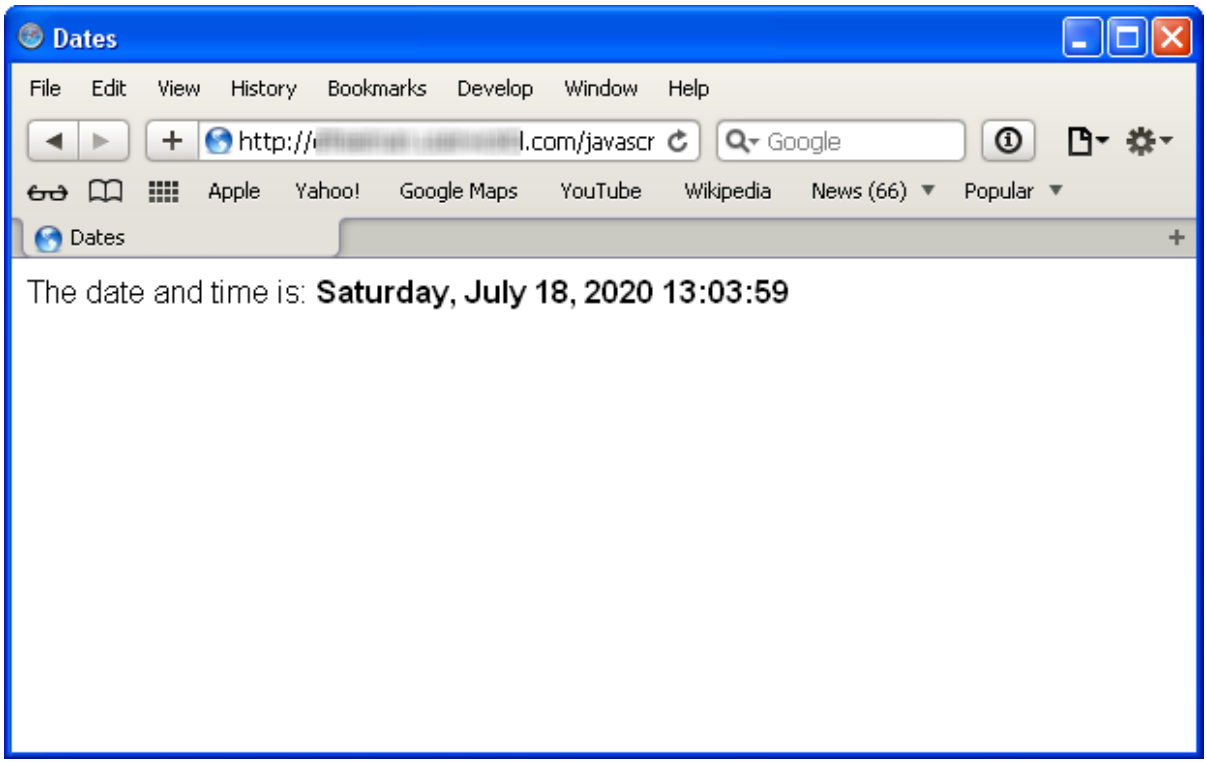
Using the Date constructor, you can specify the year, month, day, hour, minutes, seconds, and milliseconds all together to create a date and time. But what if you need to set the various values separately? In that case, you can use the Date object's *set* methods. Modify *dates.js* as shown

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var nyd = new Date(2050, 0, 1, 0, 1, 0);
    datetime.innerHTML = nyd.toLocaleString();
    var aDate = new Date();
    aDate.setFullYear(2020);
    aDate.setHours(13);
    aDate.setMinutes(3);
    aDate.setSeconds(59);
    datetime.innerHTML = aDate.toLocaleString();
}
```

Save your changes, and open or refresh *dates.html* in the browser. You see your current month and day (that is, the day you are doing this lesson), but in the year 2020. And the time should be set to 1:03pm in your time zone.



Here, we created a new Date object that, by default, represents the present time, and then changed the year to 2020, the hour to 1pm, and the minute to 3 minutes after 1pm. Everything else stays the same (that is, the values in place when you created the Date representing the present). There are methods for setting every aspect of a Date, including:

- **`setFullYear()`**: sets the year.
- **`setMonth()`**: sets the month.
- **`setDate()`**: sets the day of the month.
- **`setHours()`**: sets the hour of the day.
- **`setMinutes()`**: sets the minutes after the hour.
- **`setSeconds()`**: sets the seconds after the minute.
- **`setMilliseconds()`**: sets the milliseconds after the seconds.
- **`setTime()`**: takes a date represented as milliseconds after 1970 and sets the full date.

Experiment with these other methods and try using them in your code to set specific dates and times.

Comparing and Setting Dates Relative to the Present

Two tasks you might need to do fairly often when working with dates are *comparing dates* and *setting dates relative to the present*. For both of these tasks, working with dates expressed in milliseconds since 1970 is the way to go.

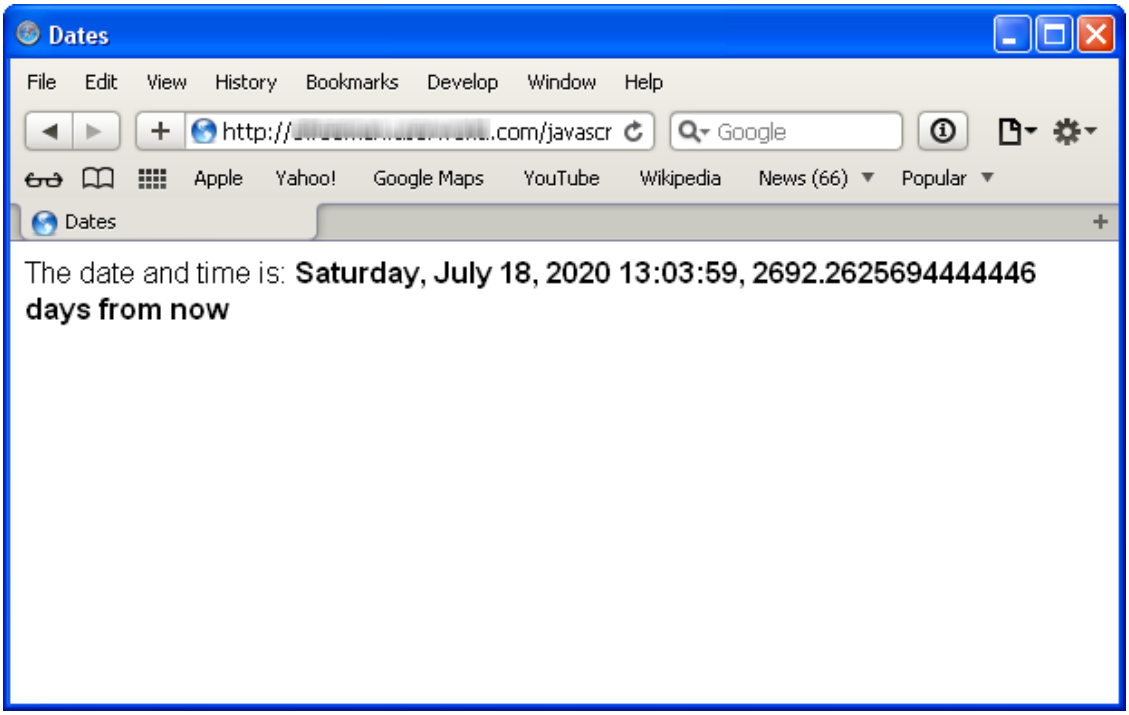
First, let's try comparing dates. Modify *dates.js* as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var aDate = new Date();
    aDate.setFullYear(2020);
    aDate.setHours(13);
    aDate.setMinutes(3);
    aDate.setSeconds(59);
    datetime.innerHTML = aDate.toLocaleString();
    var now = new Date();
    var diff = aDate.getTime() - now.getTime();
    var days = diff / 1000 / 60 / 60 / 24;
    datetime.innerHTML = aDate.toLocaleString() + ", " + days + " days
from now";
}
```

Save your changes, and open or refresh *dates.html* in the browser again. You'll see the number of days between now and the same date in 2020 (and remember, your date will be different from mine because the dates are based on *now*, that is, the date and time you are doing this lesson).



To compute the difference between two dates, we create two Date objects. We have one Date object for the date in 2020, and one Date object for now. Then, we can use the `getTime()` method to get the date and time in milliseconds since 1970. The number of milliseconds to the date in 2020 will be longer than the number of milliseconds to now (assuming it's still before 2020 of course!).

So we subtract the milliseconds to now from the milliseconds to the date in 2020 to get the difference in milliseconds. Then, we convert from milliseconds to days by dividing by 1000 (the number of milliseconds in a second), then 60 (the number of seconds in a minute), then 60 again (the number of minutes in an hour), and then 24 (the number of hours in a day). The result is the number of days between now and the date in 2020.

That number is kind of ugly when we display it because of the precision of the number after the decimal point. Let's cut off everything after the decimal point to make it easier to read. We can use the `Math.floor()` method to do this. Modify *dates.js* as shown:

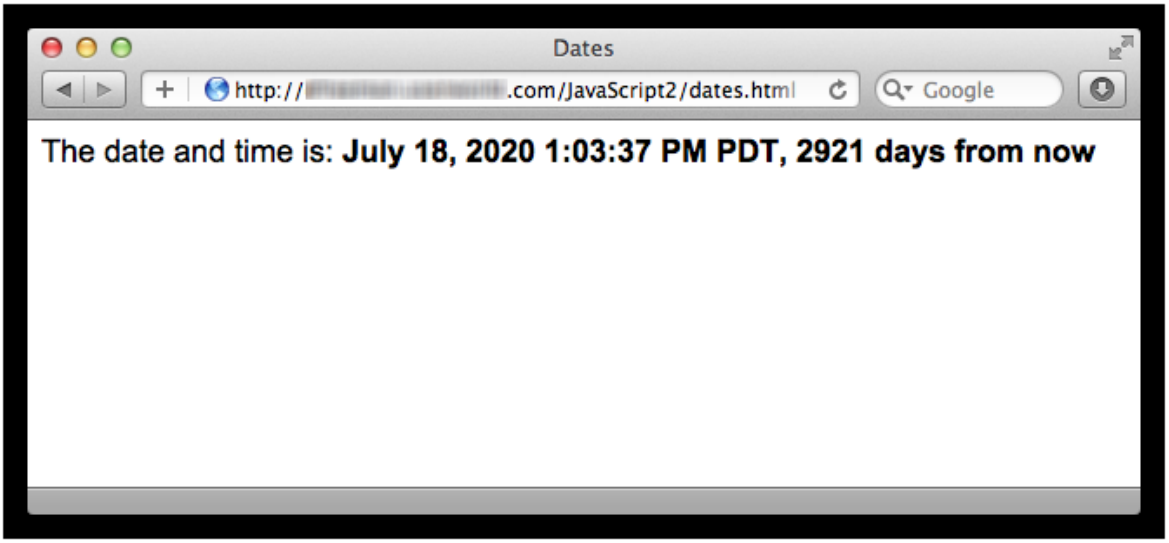
CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var aDate = new Date();
    aDate.setFullYear(2020);
    aDate.setHours(13);
```

```
    aDate.setMinutes(3);
    var now = new Date();
    var diff = aDate.getTime() - now.getTime();
    var days = Math.floor(diff / 1000 / 60 / 60 / 24);
    datetime.innerHTML = aDate.toLocaleString() + ", " + days + " days
from now";
}
```

Now save these changes, and once again open or refresh *dates.html* in the browser. Now the number of days is be easier to read:



Remember that *Math* is a built-in JavaScript object with lots of handy methods you can use for doing math computations. `Math.floor()` drops all of the numbers after the decimal point in a floating point number, so you get a number that is less than (or equal to, if the number is even) the original. Compare that to `Math.ceil()` which rounds up and then drops the numbers.

Now let's try creating a date that's three days from now. Modify *dates.js* as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var aDate = new Date();
    aDate.setFullYear(2020);
    aDate.setHours(13);
```

```
    aDate.setMinutes(3);
    var now = new Date();
    var diff = aDate.getTime() - now.getTime();
    var days = Math.floor(diff / 1000 / 60 / 60 / 24);
    datetime.innerHTML = aDate.toLocaleString() + ", " + days + " days
from now";
    var now = new Date();
    var threeDays = (24 * 60 * 60 * 1000) * 3;
    var threeDaysFromNow = new Date(now.getTime() + threeDays);
    datetime.innerHTML = now.toLocaleString() + "; 3 days from now: "
+ threeDaysFromNow.toLocaleString();
}
```

Save your changes, and open or refresh *dates.html* in your browser. You see two dates: now, and three days from now.

We also use the date expressed as milliseconds since 1970 to create a date three days from now. Here, we create a Date representing now. Then we figure out how many milliseconds are in three days, and create another Date that is now (expressed in milliseconds) plus the number of milliseconds in three days. That gives us a Date three days from now. Experiment by creating other dates. Can you create a date that is three days in the past from now?

Converting Strings to Dates

You might have an application, like the To-Do List application we've been building in this course, that asks the user to submit a date. In the To-Do List application, we ask the user to submit a date for when the to-do item is due.

When you submit a date using a form, whether you're using `<input type="date">` or `<input type="text">`, the value you get when you process the input data with JavaScript

is a String. But sometimes you might need that value as a Date object. That's when you'll use the techniques we're learning in this lesson.

The Date object has a method named `parse()` that you can use to parse a string representing a date. Let's see how we can use it in an example. First, we'll update our HTML to add a form entry for a date, and then update our JavaScript to process the string value we get from the form. We'll convert the string into a Date, and then display the Date in the page, in the "datetime" `<span>`. Modify *dates.html* as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<title>Dates</title>
  <meta charset="utf-8">
  <script src="dates.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
    span {
      font-weight: bold;
    }
    form {
      margin-bottom: 20px;
    }
  </style>
</head>
<body>
    <form>
      <label>Enter a date:</label>
      <input type="date" id="aDate">
      <input type="button" id="submit" value="Submit">
    </form>
    <div>
      The date and time is:
      <span id="datetime"></span>
    </div>
</body>
</html>
```

Save the changes, and then modify *dates.js* as shown:

CODE TO TYPE:

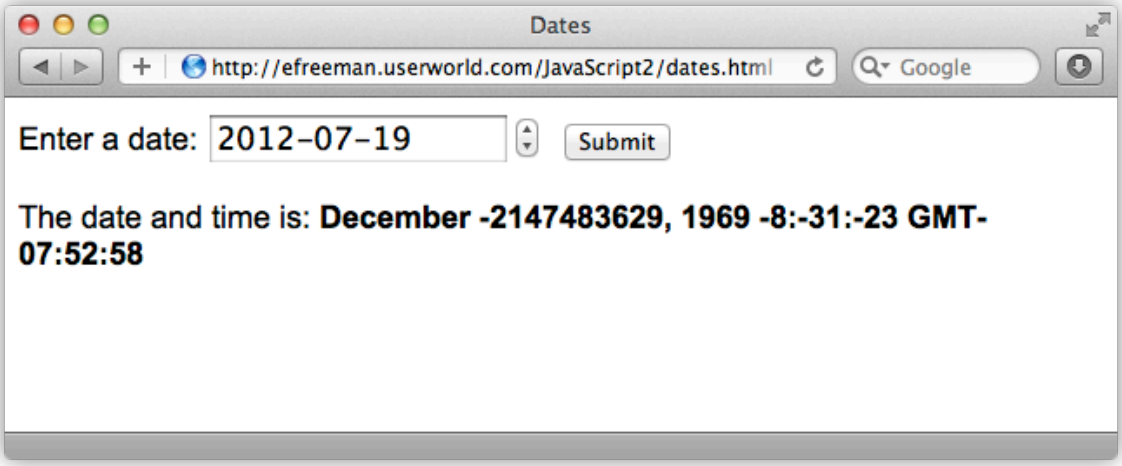
```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var now = new Date();
    var threeDays = (24 * 60 * 60 * 1000) * 3;
    var threeDaysFromNow = new Date(now.getTime() + threeDays);
    datetime.innerHTML = now.toLocaleString() + ", 3 days from now: "
+ threeDaysFromNow.toLocaleString();
    var submit = document.getElementById("submit");
    submit.onclick = getDate;
}
function getDate() {
    var aDateString = document.getElementById("aDate").value;
    if (aDateString == null || aDateString == "") {
        alert("Please enter a date");
        return;
    }
    var aDateMillis = Date.parse(aDateString);
    alert(aDateMillis);
    var aDate = new Date(aDateMillis);

    var datetime = document.getElementById("datetime");
    datetime.innerHTML = aDate.toLocaleString();
}
```

Save the changes to *dates.js*, and open or refresh *dates.html* in the browser. Enter a date in the date input field. Click *Submit*. You first see an alert, and then you see a date displayed in the page below the form input. Try writing the date in different formats.

You probably see the string "NaN" in the alert, and you probably get a date that makes no sense in the page. If you're using a browser like Safari or Chrome that displays arrows next to the "date" input control, and you use these to enter a date, like "07-19-2012," you'll still see a nonsensical date displayed in the page.

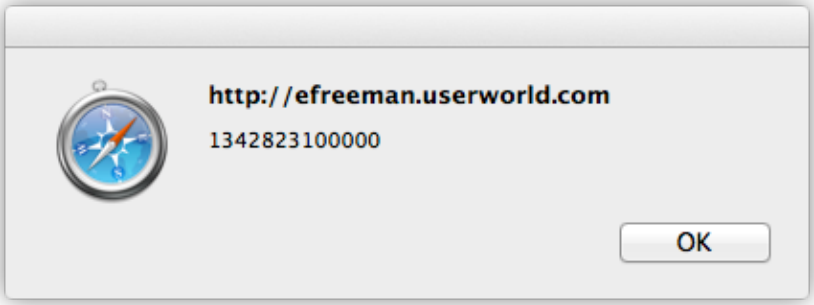


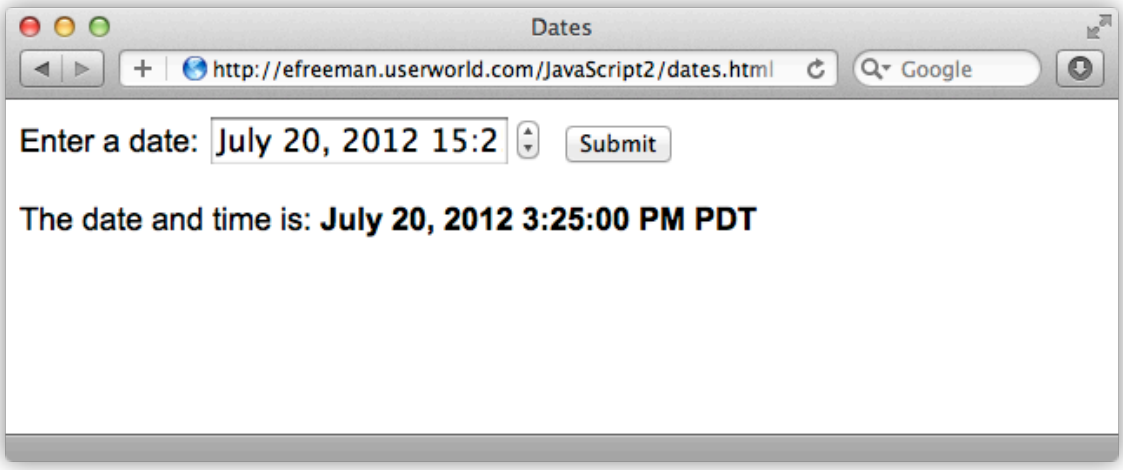
Something's definitely gone wrong, because that date doesn't make sense.

You'll likely find that this code works properly only when you enter dates in particular formats. One of the formats you can use is the same format you see when you use `toString()` or `toLocaleString()`, as we've done in these examples. Try entering a date using this format:

July 20, 2012 15:25 PDT

Now it should work. You see an alert with the number of milliseconds representing that date, and the date appears properly under the form.





Here are some other formats you can try:

- July 20, 2012
- 2012/7/20
- 2012.7.20
- 2012-7-20
- 7/20/2012
- 7-20-2012
- 7.20.2012

Take note of the different formats you try, which ones work, and which ones don't.

Let's go through the code and see what's happening:

OBSERVE:

```
function getDate() {
    var aDateString = document.getElementById("aDate").value;
    if (aDateString == null || aDateString == "") {
        alert("Please enter a date");
        return;
    }
    var aDateMillis = Date.parse(aDateString);
    alert(aDateMillis);
    var aDate = new Date(aDateMillis);

    var datetime = document.getElementById("datetime");
    datetime.innerHTML = aDate.toLocaleString();
}
```

First, we get the value of the "`aDate`" input as a string. We check to make sure the string isn't empty. If it is, we alert the user and ask them to enter a date, and then return from the function.

If we get a string, we try to parse it using `Date.parse()`. `Date.parse()` will return the date in milliseconds from the string you pass in, but only if the method can parse the string. As you've discovered, there are only certain string formats that `Date.parse()` will parse correctly. If `Date.parse()` fails, then instead of returning the milliseconds (a large number), it returns "`NaN`," meaning "Not a Number." That's JavaScript's way of letting you know it couldn't parse this string into a number. So, if you enter a date using the wrong type of string format, you see "`NaN`" in the alert.

After getting the date (or trying to get the date) in milliseconds using `Date.parse()`, we create a new Date object from that value. If we're successful, `aDate` will be a valid Date object. If not, `aDate` will be a nonsensical date, because the `Date()` constructor can't make sense of the value, "`NaN`".

Finally, we display the Date in the "datetime" `<span>` using the `toLocaleString()` method.

Note We've been using *Date* objects by using the `Date()` constructor to create a date object and then calling methods on that object. So what is `Date.parse()`? `Date.parse()` is an example of a *static method*. In JavaScript, functions are objects, so the `Date()` constructor function that you use to create new date objects is, itself, an object. And it so happens that that object has a method named `parse()`. You call static methods using

the name of the constructor function, *Date*, but without the parentheses *()* that invoke the function. Don't worry if you don't fully understand this. This topic is more advanced JavaScript and isn't within the scope of this course. Still, you're getting a taste of the kinds of things you can do with JavaScript when you delve into *object-oriented programming*. But, that's a topic for another course...

Dates and HTML Input Types

When you enter a date into an `<input>` element and you parse it using the `Date.parse()` method, you must enter a date string that the `parse()` method can understand. For browsers that support the "date" `<input>` type, and offer a date picker for entering a date, this becomes less of an issue because the date picker usually creates the date as a string with the correct date format, a format that can be parsed using `Date.parse()`. However, if you enter a date using an invalid format, that can create an error in your code, and possibly cause your web page to malfunction.

We already checked to make sure the user is entering something into the field (by checking to see if the input is null or the empty string). We should also be checking to make sure that the user is entering a valid date. We'll tackle this using *Exception Handling* in the next lesson.

In this lesson, we've explored the JavaScript *Date* object. You've learned how to create Dates, display Dates, compare Dates, and convert strings to Dates. You've also learned a few ins and outs of time zones. As you can see, working with Dates can sometimes be tricky! But fun too, right? Take a short break and review this lesson before moving on to the next.

More about the material in this lesson

The Riverside JS Workshop would like to acknowledge the generosity of O'Reilly Media, Inc. for making this material available to us through the Creative Commons License. We would also like to acknowledge the great work of Elisabeth Robson who authored this content. She is one of our favorites. Elisabeth has written a number of *Head First* programming books for web developers. We recommend them highly. You can browse her work here.