28th November 2019

Date and time

Let's meet a new built-in object: Date. It stores the date, time and provides methods for date/time management.

For instance, we can use it to store creation/modification times, to measure time, or just to print out the current date.

Creation

To create a new `Date` object call `new Date()` with one of the following arguments:

`new Date()`

Without arguments – create a `Date` object for the current date and time:

```
let now = new Date();
alert( now ); // shows current date/time
```

`new Date(milliseconds)`

Create a `Date` object with the time equal to number of milliseconds (1/1000 of a second) passed after the Jan 1st of 1970 UTC+0.

```
// 0 means 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );

// now add 24 hours, get 02.01.1970 UTC+0
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 );
```

An integer number representing the number of milliseconds that has passed since the beginning of 1970 is called a *timestamp*.

It's a lightweight numeric representation of a date. We can always create a date from a timestamp using `new Date(timestamp)` and convert the existing `Date` object to a timestamp using the `date.getTime()` method (see below).

Dates before 01.01.1970 have negative timestamps, e.g.:

```
// 31 Dec 1969
let Dec31_1969 = new Date(-24 * 3600 * 1000);
alert( Dec31_1969 );
```

`new Date(datestring)`

If there is a single argument, and it's a string, then it is parsed automatically. The algorithm is the same as `Date.parse` uses, we'll cover it later.

```
let date = new Date("2017-01-26");
alert(date);
// The time is not set, so it's assumed to be midnight GMT and
// is adjusted according to the timezone the code is run in
// So the result could be
// Thu Jan 26 2017 11:00:00 GMT+1100 (Australian Eastern Daylight Time)
// or
// Wed Jan 25 2017 16:00:00 GMT-0800 (Pacific Standard Time)
```

`new Date(year, month, date, hours, minutes, seconds, ms)`

Create the date with the given components in the local time zone. Only the first two arguments are obligatory.

- The `year` must have 4 digits: `2013` is okay, `98` is not.
- The `month` count starts with `0` (Jan), up to `11` (Dec).
- The `date` parameter is actually the day of month, if absent then `1` is assumed.
- If `hours/minutes/seconds/ms` is absent, they are assumed to be equal `0`.

For instance:

```
new Date(2011, 0, 1, 0, 0, 0, 0); // 1 Jan 2011, 00:00:00
new Date(2011, 0, 1); // the same, hours etc are 0 by default
```

The minimal precision is 1 ms (1/1000 sec):

```
let date = new Date(2011, 0, 1, 2, 3, 4, 567);
alert( date ); // 1.01.2011, 02:03:04.567
```

Access date components

There are methods to access the year, month and so on from the `Date` object:

getFullYear()

Get the year (4 digits)

getMonth()

Get the month, **from 0 to 11**.

getDate()

Get the day of month, from 1 to 31, the name of the method does look a little bit strange.

getHours(), getMinutes(), getSeconds(), getMilliseconds()

Get the corresponding time components.

> **Not `getYear()`, but `getFullYear()`**
>
> Many JavaScript engines implement a non-standard method `getYear()`. This method is deprecated. It returns 2-digit year sometimes. Please never use it. There is `getFullYear()` for the year.

Additionally, we can get a day of week:

getDay()

Get the day of week, from `0` (Sunday) to `6` (Saturday). The first day is always Sunday, in some countries that's not so, but can't be changed.

All the methods above return the components relative to the local time zone.

There are also their UTC-counterparts, that return day, month, year and so on for the time zone UTC+0: getUTCFullYear(), getUTCMonth(), getUTCDay(). Just insert the `"UTC"` right after `"get"`.

If your local time zone is shifted relative to UTC, then the code below shows different hours:

```
1  // current date
2  let date = new Date();
3
4  // the hour in your current time zone
5  alert( date.getHours() );
6
7  // the hour in UTC+0 time zone (London time without daylight savings)
8  alert( date.getUTCHours() );
```

Besides the given methods, there are two special ones that do not have a UTC-variant:

getTime()

Returns the timestamp for the date – a number of milliseconds passed from the January 1st of 1970 UTC+0.

getTimezoneOffset()

Returns the difference between the local time zone and UTC, in minutes:

```
1  // if you are in timezone UTC-1, outputs 60
2  // if you are in timezone UTC+3, outputs -180
3  alert( new Date().getTimezoneOffset() );
```

Setting date components

The following methods allow to set date/time components:

- `setFullYear(year, [month], [date])`
- `setMonth(month, [date])`
- `setDate(date)`
- `setHours(hour, [min], [sec], [ms])`

- `setMinutes(min, [sec], [ms])`
- `setSeconds(sec, [ms])`
- `setMilliseconds(ms)`
- `setTime(milliseconds)` (sets the whole date by milliseconds since 01.01.1970 UTC)

Every one of them except `setTime()` has a UTC-variant, for instance: `setUTCHours()`.

As we can see, some methods can set multiple components at once, for example `setHours`. The components that are not mentioned are not modified.

For instance:

```
let today = new Date();

today.setHours(0);
alert(today); // still today, but the hour is changed to 0

today.setHours(0, 0, 0, 0);
alert(today); // still today, now 00:00:00 sharp.
```

Autocorrection

The *autocorrection* is a very handy feature of `Date` objects. We can set out-of-range values, and it will auto-adjust itself.

For instance:

```
let date = new Date(2013, 0, 32); // 32 Jan 2013 ?!?
alert(date); // ...is 1st Feb 2013!
```

Out-of-range date components are distributed automatically.

Let's say we need to increase the date "28 Feb 2016" by 2 days. It may be "2 Mar" or "1 Mar" in case of a leap-year. We don't need to think about it. Just add 2 days. The `Date` object will do the rest:

```
let date = new Date(2016, 1, 28);
date.setDate(date.getDate() + 2);

alert( date ); // 1 Mar 2016
```

That feature is often used to get the date after the given period of time. For instance, let's get the date for "70 seconds after now":

```
let date = new Date();
date.setSeconds(date.getSeconds() + 70);

alert( date ); // shows the correct date
```

We can also set zero or even negative values. For example:

```
let date = new Date(2016, 0, 2); // 2 Jan 2016

date.setDate(1); // set day 1 of month
alert( date );

date.setDate(0); // min day is 1, so the last day of the previous month is as
alert( date ); // 31 Dec 2015
```

Date to number, date diff

When a `Date` object is converted to number, it becomes the timestamp same as `date.getTime()`:

```
let date = new Date();
alert(+date); // the number of milliseconds, same as date.getTime()
```

The important side effect: dates can be subtracted, the result is their difference in ms.

That can be used for time measurements:

```
let start = new Date(); // start measuring time

// do the job
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}

let end = new Date(); // end measuring time

alert( `The loop took ${end - start} ms` );
```

Date.now()

If we only want to measure time, we don't need the `Date` object.

There's a special method `Date.now()` that returns the current timestamp.

It is semantically equivalent to `new Date().getTime()`, but it doesn't create an intermediate `Date` object. So it's faster and doesn't put pressure on garbage collection.

It is used mostly for convenience or when performance matters, like in games in JavaScript or other specialized applications.

So this is probably better:

```
let start = Date.now(); // milliseconds count from 1 Jan 1970

// do the job
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}

```

```
let end = Date.now(); // done

alert( `The loop took ${end - start} ms` ); // subtract numbers, not dates
```

Benchmarking

If we want a reliable benchmark of CPU-hungry function, we should be careful.

For instance, let's measure two functions that calculate the difference between two dates: which one is faster?

Such performance measurements are often called "benchmarks".

```
// we have date1 and date2, which function faster returns their difference in
function diffSubtract(date1, date2) {
  return date2 - date1;
}

// or
function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}
```

These two do exactly the same thing, but one of them uses an explicit `date.getTime()` to get the date in ms, and the other one relies on a date-to-number transform. Their result is always the same.

So, which one is faster?

The first idea may be to run them many times in a row and measure the time difference. For our case, functions are very simple, so we have to do it at least 100000 times.

Let's measure:

```
function diffSubtract(date1, date2) {
  return date2 - date1;
}

function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}

function bench(f) {
  let date1 = new Date(0);
  let date2 = new Date();

  let start = Date.now();
  for (let i = 0; i < 100000; i++) f(date1, date2);
  return Date.now() - start;
}

alert( 'Time of diffSubtract: ' + bench(diffSubtract) + 'ms' );
alert( 'Time of diffGetTime: ' + bench(diffGetTime) + 'ms' );
```

Wow! Using `getTime()` is so much faster! That's because there's no type conversion, it is much easier for engines to optimize.

Okay, we have something. But that's not a good benchmark yet.

Imagine that at the time of running `bench(diffSubtract)` CPU was doing something in parallel, and it was taking resources. And by the time of running `bench(diffGetTime)` that work has finished.

A pretty real scenario for a modern multi-process OS.

As a result, the first benchmark will have less CPU resources than the second. That may lead to wrong results.

For more reliable benchmarking, the whole pack of benchmarks should be rerun multiple times.

For example, like this:

```
function diffSubtract(date1, date2) {
  return date2 - date1;
}

function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}

function bench(f) {
  let date1 = new Date(0);
  let date2 = new Date();

  let start = Date.now();
  for (let i = 0; i < 100000; i++) f(date1, date2);
  return Date.now() - start;
}

let time1 = 0;
let time2 = 0;

// run bench(upperSlice) and bench(upperLoop) each 10 times alternating
for (let i = 0; i < 10; i++) {
  time1 += bench(diffSubtract);
  time2 += bench(diffGetTime);
}

alert( 'Total time for diffSubtract: ' + time1 );
alert( 'Total time for diffGetTime: ' + time2 );
```

Modern JavaScript engines start applying advanced optimizations only to "hot code" that executes many times (no need to optimize rarely executed things). So, in the example above, first executions are not well-optimized. We may want to add a heat-up run:

```
// added for "heating up" prior to the main loop
bench(diffSubtract);
bench(diffGetTime);

// now benchmark
for (let i = 0; i < 10; i++) {
  time1 += bench(diffSubtract);

```

```
    time2 += bench(diffGetTime);
  }
```

Be careful doing microbenchmarking

Modern JavaScript engines perform many optimizations. They may tweak results of “artificial tests” compared to “normal usage”, especially when we benchmark something very small, such as how an operator works, or a built-in function. So if you seriously want to understand performance, then please study how the JavaScript engine works. And then you probably won’t need microbenchmarks at all.

The great pack of articles about V8 can be found at http://mrale.ph.

Date.parse from a string

The method Date.parse(str) can read a date from a string.

The string format should be: `YYYY-MM-DDTHH:mm:ss.sssZ`, where:

- `YYYY-MM-DD` – is the date: year-month-day.
- The character `"T"` is used as the delimiter.
- `HH:mm:ss.sss` – is the time: hours, minutes, seconds and milliseconds.
- The optional `'Z'` part denotes the time zone in the format `+-hh:mm`. A single letter `Z` that would mean UTC+0.

Shorter variants are also possible, like `YYYY-MM-DD` or `YYYY-MM` or even `YYYY`.

The call to `Date.parse(str)` parses the string in the given format and returns the timestamp (number of milliseconds from 1 Jan 1970 UTC+0). If the format is invalid, returns `NaN`.

For instance:

```
let ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert(ms); // 1327611110417  (timestamp)
```

We can instantly create a `new Date` object from the timestamp:

```
let date = new Date( Date.parse('2012-01-26T13:51:50.417-07:00') );

alert(date);
```

Summary

- Date and time in JavaScript are represented with the Date object. We can’t create “only date” or “only time”: `Date` objects always carry both.
- Months are counted from zero (yes, January is a zero month).
- Days of week in `getDay()` are also counted from zero (that’s Sunday).

- `Date` auto-corrects itself when out-of-range components are set. Good for adding/subtracting days/months/hours.
- Dates can be subtracted, giving their difference in milliseconds. That's because a `Date` becomes the timestamp when converted to a number.
- Use `Date.now()` to get the current timestamp fast.

Note that unlike many other systems, timestamps in JavaScript are in milliseconds, not in seconds.

Sometimes we need more precise time measurements. JavaScript itself does not have a way to measure time in microseconds (1 millionth of a second), but most environments provide it. For instance, browser has performance.now() that gives the number of milliseconds from the start of page loading with microsecond precision (3 digits after the point):

```
1  alert(`Loading started ${performance.now()}ms ago`);
2  // Something like: "Loading started 34731.26000000001ms ago"
3  // .26 is microseconds (260 microseconds)
4  // more than 3 digits after the decimal point are precision errors, but only
```

Node.js has `microtime` module and other ways. Technically, almost any device and environment allows to get more precision, it's just not in `Date` .

Tasks

Create a date

importance: 5

Create a `Date` object for the date: Feb 20, 2012, 3:12am. The time zone is local.

Show it using `alert` .

solution

Show a weekday

importance: 5

Write a function `getWeekDay(date)` to show the weekday in short format: 'MO', 'TU', 'WE', 'TH', 'FR', 'SA', 'SU'.

For instance:

```
1  let date = new Date(2012, 0, 3);  // 3 Jan 2012
2  alert( getWeekDay(date) );        // should output "TU"
```

Open a sandbox with tests.

solution

European weekday

importance: 5

European countries have days of week starting with Monday (number 1), then Tuesday (number 2) and till Sunday (number 7). Write a function `getLocalDay(date)` that returns the "European" day of week for `date`.

```
let date = new Date(2012, 0, 3);  // 3 Jan 2012
alert( getLocalDay(date) );       // tuesday, should show 2
```

Open a sandbox with tests.

solution

Which day of month was many days ago?

importance: 4

Create a function `getDateAgo(date, days)` to return the day of month `days` ago from the `date`.

For instance, if today is 20th, then `getDateAgo(new Date(), 1)` should be 19th and `getDateAgo(new Date(), 2)` should be 18th.

Should work reliably for `days=365` or more:

```
let date = new Date(2015, 0, 2);

alert( getDateAgo(date, 1) ); // 1, (1 Jan 2015)
alert( getDateAgo(date, 2) ); // 31, (31 Dec 2014)
alert( getDateAgo(date, 365) ); // 2, (2 Jan 2014)
```

P.S. The function should not modify the given `date`.

Open a sandbox with tests.

solution

Last day of month?

importance: 5

Write a function `getLastDayOfMonth(year, month)` that returns the last day of month. Sometimes it is 30th, 31st or even 28/29th for Feb.

Parameters:

- `year` – four-digits year, for instance 2012.
- `month` – month, from 0 to 11.

For instance, `getLastDayOfMonth(2012, 1) = 29` (leap year, Feb).

Open a sandbox with tests.

solution

How many seconds have passed today?

importance: 5

Write a function `getSecondsToday()` that returns the number of seconds from the beginning of today.

For instance, if now were `10:00 am`, and there was no daylight savings shift, then:

```
getSecondsToday() == 36000 // (3600 * 10)
```

The function should work in any day. That is, it should not have a hard-coded value of "today".

solution

How many seconds till tomorrow?

importance: 5

Create a function `getSecondsToTomorrow()` that returns the number of seconds till tomorrow.

For instance, if now is `23:00`, then:

```
getSecondsToTomorrow() == 3600
```

P.S. The function should work at any day, the "today" is not hardcoded.

solution

Format the relative date

importance: 4

Write a function `formatDate(date)` that should format `date` as follows:

- If since `date` passed less than 1 second, then `"right now"`.
- Otherwise, if since `date` passed less than 1 minute, then `"n sec. ago"`.
- Otherwise, if less than an hour, then `"m min. ago"`.
- Otherwise, the full date in the format `"DD.MM.YY HH:mm"`. That is: `"day.month.year hours:minutes"`, all in 2-digit format, e.g. `31.12.16 10:00`.

For instance:

```
alert( formatDate(new Date(new Date - 1)) ); // "right now"

alert( formatDate(new Date(new Date - 30 * 1000)) ); // "30 sec. ago"

alert( formatDate(new Date(new Date - 5 * 60 * 1000)) ); // "5 min. ago"

// yesterday's date like 31.12.16, 20:00
alert( formatDate(new Date(new Date - 86400 * 1000)) );
```

Open a sandbox with tests.

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)