Draft Status:

Draft, 2008-04-15, pratapL

Draft, 2008-04-21, pratapL (two updates based on feedback from Lars – added optional millisec field, reverted a change to the Date ctor called as a function (15.9.2.1), minor typographic cleanup.

Rationale of the ES3.1 DateTime Proposal

At a high level, there are 3 problems with Date object:

1. ECMAScript 3rd Edition (ES3) does not specify the syntax for date and time strings in 15.9.4.2. This is a source of ambiguity. By a knock-on effect, it also makes 15.9.3.2 ambiguous (refer step 2 there). Consequently implementations have diverged and have come to use many heuristics for parsing date-time formats.
2. It lacks internationalization support
3. It provides minimal support for date calculations

How should we go about fixing these problems? Should we specify the syntax for date/time strings? Should we introduce the notion of format strings (like in Java) that will be used to interpret the date string itself? Should we augment the Date type with additional methods? Should we introduce a new kind of Date object? Perhaps, a Calendar object?

Over time, libraries have evolved to handle such issues related to Dates. At the core ECMAScript level, we ought to introduce just the optimum rigour in the specification to remove any ambiguities, and let the libraries handle all other cases.

ES3 left the content of the string returned by Date.prototype.toString() as implementation dependent, and thereby impacted Date.parse. Indeed, the only recommendation was that for any Date value d whose milliseconds amount is zero, the result of Date.parse(d.toString()) is equal to d.valueOf(). It seems reasonable that a stringified representation of a specialized object like a Date conform to some format (as opposed to leaving it as implementation dependent). Given that both these methods are underspecified in ES3, we should specify their formats in detail, and suggest that implementations should try that format first, before falling back to any heuristics (so that they may retain backwards compat).

Potential formats

There are several potential formats to choose from:

1. ISO 8601 (zip-pdf)
2. RFC 3339 (html), a profile of ISO 8601 for use in internet protocols
3. W3C Date and Time Formats (html), a profile of ISO 8601 that specifically defines a few simplified date/time formats, likely to satisfy most requirements.
4. Web Forms 2.0 “time” input elements (html), a profile of ISO 8601 used by the WHATWG
5. CLI Date Time Description (html)
6. JScript’s time formats (pdf), section 3.25)

Proposed-ES4 proposal

The Proposed-ES4 Date and Time proposal (html) calls for the use of a simplified form of ISO 8601, but adds a new static function, new methods and new properties to the Date object in the process.

ES3.1 Proposal

For ES3.1, we should specify a simple format with the following goals:

- Compatible with ES3
- Subset of ES4

- Easily readable and writeable by systems
- Unambiguous
- Easily comparable and sortable
- For most representations the notation is short and of constant length
- Compatible with formats currently used in various implementations/Frameworks - for e.g. the format used by Dojo (html)

ES3.1 date and time strings are specified using a **Simplified ISO 8601 format** based on the W3C Date and Time Formats (html), a profile of ISO 8601 that specifically, with the added ability to specify extended dates (6 digit years).

DateTime string format

The Simplified ISO 8601 format is as follows: YYYY-MM-DDTHH:MM:SS.sss*TZ*

Where the components are as follows:

YYYY is the year in the Gregorian calendar

MM is the month of the year between 01 (January) and 12 (December)

DD is the day of the month between 01 and 31.

The "T" appears literally in the string, to indicate the beginning of the time element, as specified in ISO 8601.

HH is the number of complete hours that have passed since midnight

MM is the number of complete minutes since the start of the hour

SS is the number of complete seconds since the start of the minute

The '.' (dot)

sss is the number of complete milliseconds since the start of the second.

Both the '.' and the milliseconds components are optional

TZ is the timezone specified as Z (for UTC) or +/- followed by a time expression HH:MM

Extended years

ECMAScript requires the ability to specify 6 digit years (extended years); approximately 285,616 years, either forward or backward, from 01 January, 1970 UTC. To represent years before 0 or after 9999, ISO 8601 permits the expansion of the year representation, but only by prior agreement between the sender and the receiver. In the case of ES3.1 such an expanded year representation shall have 2 extra year digits and is always prefixed with a + or − sign with the convention that year 0 is positive.

Notes

- Exactly the components shown here must be present, with exactly this punctuation.
- All numbers must be base 10.
- Illegal values (out-of-bounds as well as syntax errors) in the format string shall cause Date.parse to return NaN
- As every day both starts and ends with midnight, the two notations 00:00 and 24:00 are available to distinguish the two midnights that can be associated with one date. This means that the following two notations refer to exactly the same point in time: 1995-02-04T24:00 = 1995-02-05T00:00
- There exists no international standard that specifies abbreviations for civil time zones like CET, EST, etc. and sometimes the same abbreviation is even used for two very different time zones. ISO 8601, and the convention used in ES3.1, specifies *numeric* representations of date and time.

Changes to Date methods

`Date.parse` shall first try to parse this Simplified ISO 8601 format, before falling back to any implementation-specific heuristics (to retain backwards compat).

The contents of the string returned by `Date.prototype.toString()` shall follow the format called out in "DateTime string format".

The contents of the string returned by `Date.prototype.toUTCString()` shall follow the format called out in "DateTime string format".

There shall be no change to any other API on the Date object.

15.9 Date Objects

15.9.1 Overview of Date Objects and Definitions of Internal Operators

A Date object contains a number indicating a particular instant in time to within a millisecond. The number may also be **NaN**, indicating that the Date object does not represent a specific instant of time.

The following sections define a number of functions for operating on time values. Note that, in every case, if any argument to such a function is **NaN**, the result will be **NaN**.

15.9.1.1 Time Range

Time is measured in ECMAScript in milliseconds since 01 January, 1970 UTC. Leap seconds are ignored. It is assumed that there are exactly 86,400,000 milliseconds per day. ECMAScript number values can represent all integers from –9,007,199,254,740,991 to 9,007,199,254,740,991; this range suffices to measure times to millisecond precision for any instant that is within approximately 285,616 years, either forward or backward, from 01 January, 1970 UTC.

The actual range of times supported by ECMAScript Date objects is slightly smaller: exactly –100,000,000 days to 100,000,000 days measured relative to midnight at the beginning of 01 January, 1970 UTC. This gives a range of 8,640,000,000,000,000 milliseconds to either side of 01 January, 1970 UTC.

The exact moment of midnight at the beginning of 01 January, 1970 UTC is represented by the value **+0**.

15.9.1.2 Day Number and Time within Day

A given time value *t* belongs to day number

Day(*t*) = floor(*t* / msPerDay)

where the number of milliseconds per day is

msPerDay = 86400000

The remainder is called the time within the day:

TimeWithinDay(*t*) = *t* modulo msPerDay

15.9.1.3 Year Number

ECMAScript uses an extrapolated Gregorian system to map a day number to a year number and to determine the month and date within that year. In this system, leap years are precisely those which are (divisible by 4) and ((not divisible by 100) or (divisible by 400)). The number of days in year number *y* is therefore defined by

DaysInYear(*y*) = 365 if (*y* modulo 4) ≠ 0
= 366 if (*y* modulo 4) = 0 and (y modulo 100) ≠ 0
= 365 if (*y* modulo 100) = 0 and (y modulo 400) ≠ 0
= 366 if (*y* modulo 400) = 0

All non-leap years have 365 days with the usual number of days per month and leap years have an extra day in February. The day number of the first day of year *y* is given by:

DayFromYear(y) = 365 × (y–1970) + floor((y–1969)/4) – floor((y–1901)/100) + floor((y–1601)/400)

The time value of the start of a year is:

TimeFromYear(*y*) = msPerDay × DayFromYear(*y*)

A time value determines a year by:

YearFromTime(*t*) = the largest integer *y* (closest to positive infinity) such that TimeFromYear(*y*) ≤ *t*

The leap-year function is 1 for a time within a leap year and otherwise is zero:

InLeapYear(*t*) = 0 if DaysInYear(YearFromTime(*t*)) = 365
= 1 if DaysInYear(YearFromTime(*t*)) = 366

15.9.1.4 Month Number

Months are identified by an integer in the range 0 to 11, inclusive. The mapping MonthFromTime(*t*) from a time value *t* to a month number is defined by:

MonthFromTime(*t*)	= 0	if	0	≤ DayWithinYear(*t*) < 31
	= 1	if	31	≤ DayWithinYear (*t*) < 59+InLeapYear(*t*)
	= 2	if	59+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 90+InLeapYear(*t*)
	= 3	if	90+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 120+InLeapYear(*t*)
	= 4	if	120+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 151+InLeapYear(*t*)
	= 5	if	151+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 181+InLeapYear(*t*)
	= 6	if	181+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 212+InLeapYear(*t*)
	= 7	if	212+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 243+InLeapYear(*t*)
	= 8	if	243+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 273+InLeapYear(*t*)
	= 9	if	273+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 304+InLeapYear(*t*)
	= 10	if	304+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 334+InLeapYear(*t*)
	= 11	if	334+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 365+InLeapYear(*t*)

where

DayWithinYear(*t*) = Day(*t*)–DayFromYear(YearFromTime(*t*))

A month value of 0 specifies January; 1 specifies February; 2 specifies March; 3 specifies April; 4 specifies May; 5 specifies June; 6 specifies July; 7 specifies August; 8 specifies September; 9 specifies October; 10 specifies November; and 11 specifies December. Note that MonthFromTime(0) = 0, corresponding to Thursday, 01 January, 1970.

15.9.1.5 Date Number

A date number is identified by an integer in the range 1 through 31, inclusive. The mapping DateFromTime(*t*) from a time value *t* to a month number is defined by:

DateFromTime(*t*)	= DayWithinYear(*t*)+1	if MonthFromTime(*t*)=0
	= DayWithinYear(*t*)–30	if MonthFromTime(*t*)=1
	= DayWithinYear(*t*)–58–InLeapYear(*t*)	if MonthFromTime(*t*)=2
	= DayWithinYear(*t*)–89–InLeapYear(*t*)	if MonthFromTime(*t*)=3
	= DayWithinYear(*t*)–119–InLeapYear(*t*)	if MonthFromTime(*t*)=4
	= DayWithinYear(*t*)–150–InLeapYear(*t*)	if MonthFromTime(*t*)=5
	= DayWithinYear(*t*)–180–InLeapYear(*t*)	if MonthFromTime(*t*)=6
	= DayWithinYear(*t*)–211–InLeapYear(*t*)	if MonthFromTime(*t*)=7
	= DayWithinYear(*t*)–242–InLeapYear(*t*)	if MonthFromTime(*t*)=8
	= DayWithinYear(*t*)–272–InLeapYear(*t*)	if MonthFromTime(*t*)=9
	= DayWithinYear(*t*)–303–InLeapYear(*t*)	if MonthFromTime(*t*)=10
	= DayWithinYear(*t*)–333–InLeapYear(*t*)	if MonthFromTime(*t*)=11

15.9.1.6 Week Day

The weekday for a particular time value *t* is defined as

WeekDay(*t*) = (Day(*t*) + 4) modulo 7

A weekday value of 0 specifies Sunday; 1 specifies Monday; 2 specifies Tuesday; 3 specifies Wednesday; 4 specifies Thursday; 5 specifies Friday; and 6 specifies Saturday. Note that WeekDay(0) = 4, corresponding to Thursday, 01 January, 1970.

15.9.1.8 Local Time Zone Adjustment

An implementation of ECMAScript is expected to determine the local time zone adjustment. The local time zone adjustment is a value LocalTZA measured in milliseconds which when added to UTC represents the local *standard* time. Daylight saving time is *not* reflected by LocalTZA. The value LocalTZA does not vary with time but depends only on the geographic location.

Comment [pL1]: This assertion is incorrect. It assumes time zone boundaries are fixed for eternity. It is not, and is subject to politics (as seen by the recent DST change that has happened in US.

The wording in this section needs to change.

15.9.1.9 Daylight Saving Time Adjustment

An implementation of ECMAScript is expected to determine the daylight saving time algorithm. The algorithm to determine the daylight saving time adjustment DaylightSavingTA(*t*), measured in milliseconds, must depend only on four things:

Comment [pL2]: Same as the earlier comment. This assertion about DST is incorrect. The wording needs to be changed.

(1) the time since the beginning of the year

t – TimeFromYear(YearFromTime(*t*))

(2) whether *t* is in a leap year

InLeapYear(*t*)

(3) the week day of the beginning of the year

WeekDay(TimeFromYear(YearFromTime(*t*))

and (4) the geographic location.

The implementation of ECMAScript should not try to determine whether the exact time was subject to daylight saving time, but just whether daylight saving time would have been in effect if the current daylight saving time algorithm had been used at the time. This avoids complications such as taking into account the years that the locale observed daylight saving time year round.

If the host environment provides functionality for determining daylight saving time, the implementation of ECMAScript is free to map the year in question to an equivalent year (same leap-year-ness and same starting week day for the year) for which the host environment provides daylight saving time information. The only restriction is that all equivalent years should produce the same result.

15.9.1.9 Local Time

Conversion from UTC to local time is defined by

LocalTime(*t*) = *t* + LocalTZA + DaylightSavingTA(*t*)

Conversion from local time to UTC is defined by

UTC(*t*) = *t* – LocalTZA – DaylightSavingTA(*t* – LocalTZA)

Note that UTC(LocalTime(*t*)) is not necessarily always equal to *t*.

15.9.1.10 Hours, Minutes, Second, and Milliseconds

The following functions are useful in decomposing time values:

HourFromTime(*t*) = floor(*t* / msPerHour) modulo HoursPerDay

MinFromTime(*t*) = floor(*t* / msPerMinute) modulo MinutesPerHour

SecFromTime(*t*) = floor(*t* / msPerSecond) modulo SecondsPerMinute

msFromTime(*t*) = *t* modulo msPerSecond

where

HoursPerDay = 24

MinutesPerHour = 60

SecondsPerMinute = 60

msPerSecond = 1000

msPerMinute = msPerSecond × SecondsPerMinute = 60000

msPerHour = msPerMinute × MinutesPerHour = 3600000

15.9.1.11 MakeTime (hour, min, sec, ms)

The operator MakeTime calculates a number of milliseconds from its four arguments, which must be ECMAScript number values. This operator functions as follows:

1. If *hour* is not finite or *min* is not finite or *sec* is not finite or *ms* is not finite, return **NaN**.
2. Call ToInteger(*hour*).
3. Call ToInteger(*min*).
4. Call ToInteger(*sec*).
5. Call ToInteger(*ms*).
6. Compute Result(2) ***** msPerHour **+** Result(3) ***** msPerMinute **+** Result(4) ***** msPerSecond **+** Result(5), performing the arithmetic according to IEEE 754 rules (that is, as if using the ECMAScript operators ***** and **+**).
7. Return Result(6).

15.9.1.12 MakeDay (year, month, date)

The operator MakeDay calculates a number of days from its three arguments, which must be ECMAScript number values. This operator functions as follows:

1. If *year* is not finite or *month* is not finite or *date* is not finite, return **NaN**.
2. Call ToInteger(*year*).
3. Call ToInteger(*month*).
4. Call ToInteger(*date*).
5. Compute Result(2) + floor(Result(3)/12).
6. Compute Result(3) modulo 12.
7. Find a value *t* such that YearFromTime(*t*) **==** Result(5) and MonthFromTime(*t*) **==** Result(6) and DateFromTime(*t*) **==** 1; but if this is not possible (because some argument is out of range), return **NaN**.
8. Compute Day(Result(7)) + Result(4) – 1.
9. Return Result(8).

15.9.1.13 MakeDate (day, time)

The operator MakeDate calculates a number of milliseconds from its two arguments, which must be ECMAScript number values. This operator functions as follows:

1. If *day* is not finite or *time* is not finite, return **NaN**.
2. Compute *day* × msPerDay + *time*.
3. Return Result(2).

15.9.1.14 TimeClip (time)

The operator TimeClip calculates a number of milliseconds from its argument, which must be an ECMAScript number value. This operator functions as follows:

1. If *time* is not finite, return **NaN**.
2. If abs(Result(1)) > **8.64 x 10^{15}**, return **NaN**.
3. Return an implementation-dependent choice of either ToInteger(Result(2)) or ToInteger(Result(2)) + (**+0**).
   (Adding a positive zero converts –**0** to **+0**.)

NOTE
*The point of step 3 is that an implementation is permitted a choice of internal representations of time values, for example as a 64-bit signed integer or as a 64-bit floating-point value. Depending on the implementation, this internal representation may or may not distinguish –**0** and **+0**.*

15.9.1.15 Date Time string format

The Simplified ISO 8601 format is as follows: YYYY-MM-DDTHH:MM:SS.sss*TZ*

Where the components are as follows:

YYYY is the year in the Gregorian calendar

MM is the month of the year between 01 (January) and 12 (December)

DD is the day of the month between 01 and 31.

The "T" appears literally in the string, to indicate the beginning of the time element, as specified in ISO 8601.

HH is the number of complete hours that have passed since midnight

MM is the number of complete minutes since the start of the hour

SS is the number of complete seconds since the start of the minute

The '.' (dot)

sss is the number of complete milliseconds since the start of the second.

Both the '.' and the milliseconds components are optional

TZ is the timezone specified as Z (for UTC) or +/- followed by a time expression HH:MM

Extended years

ECMAScript requires the ability to specify 6 digit years (extended years); approximately 285,616 years, either forward or backward, from 01 January, 1970 UTC. To represent years before 0 or after 9999, ISO 8601 permits the expansion of the year representation, but only by prior agreement between the sender and the receiver. In the case of ES3.1 such an expanded year representation shall have 2 extra year digits and is always prefixed with a + or − sign with the convention that year 0 is positive.

Notes

- Exactly the components shown here must be present, with exactly this punctuation.
- All numbers must be base 10.
- Illegal values (out-of-bounds as well as syntax errors) in the format string shall cause Date.parse to return NaN
- As every day both starts and ends with midnight, the two notations 00:00 and 24:00 are available to distinguish the two midnights that can be associated with one date. This means that the following two notations refer to exactly the same point in time: 1995-02-04T24:00 = 1995-02-05T00:00
- There exists no international standard that specifies abbreviations for civil time zones like CET, EST, etc. and sometimes the same abbreviation is even used for two very different time zones. ISO 8601, and the convention used in ES3.1, specifies *numeric* representations of date and time.

15.9.2 The Date Constructor Called as a Function

When **`Date`** is called as a function rather than as a constructor, it returns a string representing the current time (UTC).

NOTE
The function call ***`Date(…)`*** *is not equivalent to the object creation expression* ***`new Date(…)`*** *with the same arguments.*

15.9.2.1 Date ([year [, month [, date [, hours [, minutes [, seconds [, ms]]]]]]])

All of the arguments are optional; any arguments supplied are accepted but are completely ignored. A string is created and returned as if by the expression **`(new Date()).toString()`**.

15.9.3 The Date Constructor

When **`Date`** is called as part of a **`new`** expression, it is a constructor: it initialises the newly created object.

15.9.3.1 new Date (year, month [, date [, hours [, minutes [, seconds [, ms]]]]])

When **Date** is called with two to seven arguments, it computes the date from *year*, *month*, and (optionally) *date*, *hours*, *minutes*, *seconds* and *ms*.

The [[Prototype]] property of the newly constructed object is set to the original Date prototype object, the one that is the initial value of **`Date.prototype`** (15.9.4.1).

The [[Class]] property of the newly constructed object is set to **`"Date"`**.

The [[Value]] property of the newly constructed object is set as follows:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. If *date* is supplied use ToNumber(*date*); else use **1**.
4. If *hours* is supplied use ToNumber(*hours*); else use **0**.
5. If *minutes* is supplied use ToNumber(*minutes*); else use **0**.
6. If *seconds* is supplied use ToNumber(*seconds*); else use **0**.
7. If *ms* is supplied use ToNumber(*ms*); else use **0**.
8. If Result(1) is not **NaN** and $0 \leq$ ToInteger(Result(1)) ≤ 99, Result(8) is 1900+ToInteger(Result(1)); otherwise, Result(8) is Result(1).
9. Compute MakeDay(Result(8), Result(2), Result(3)).
10. Compute MakeTime(Result(4), Result(5), Result(6), Result(7)).
11. Compute MakeDate(Result(9), Result(10)).
12. Set the [[Value]] property of the newly constructed object to TimeClip(UTC(Result(11))).

15.9.3.2 new Date (value)

The [[Prototype]] property of the newly constructed object is set to the original Date prototype object, the one that is the initial value of **`Date.prototype`** (15.9.4.1).

The [[Class]] property of the newly constructed object is set to **`"Date"`**.

The [[Value]] property of the newly constructed object is set as follows:

1. Call ToPrimitive(*value*).
2. If Type(Result(1)) is String, then go to step 5.
3. Let *V* be ToNumber(Result(1)).
4. Set the [[Value]] property of the newly constructed object to TimeClip(*V*) and return.
5. Parse Result(1) as a date, in exactly the same manner as for the **`parse`** method (15.9.4.2); let *V* be the time value for this date.
6. Go to step 4.

15.9.3.3 new Date ( )

The [[Prototype]] property of the newly constructed object is set to the original Date prototype object, the one that is the initial value of **`Date.prototype`** (15.9.4.1).

The [[Class]] property of the newly constructed object is set to **`"Date"`**.

The [[Value]] property of the newly constructed object is set to the current time (UTC).

15.9.4 Properties of the Date Constructor

The value of the internal [[Prototype]] property of the Date constructor is the Function prototype object (15.3.4).

Besides the internal properties and the **`length`** property (whose value is **`7`**), the Date constructor has the following properties:

15.9.4.1 Date.prototype

The initial value of **`Date.prototype`** is the built-in Date prototype object (15.9.5).

This property has the attributes { DontEnum, DontDelete, ReadOnly }.

15.9.4.2 Date.parse (string)

The **`parse`** function applies the ToString operator to its argument and interprets the resulting string as a date; it returns a number, the UTC time value corresponding to the date. The string may be

interpreted as a local time, a UTC time, or a time in some other time zone, depending on the contents of the string. The format of the string accepted by the **`parse`** function is as called out in Date Time string format (15.9.1.15). Illegal values (out-of-bounds as well as syntax errors) in the format string shall cause **`Date.parse`** to return NaN. **`Date.parse`** should first try to parse this format before falling back to any implementation-specific heuristics (in order to maintain backwards compatibility).

If *x* is any Date object whose milliseconds amount is zero within a particular implementation of ECMAScript, then all of the following expressions should produce the same numeric value in that implementation, if all the properties referenced have their initial values:

```
x.valueOf()
Date.parse(x.toString())
Date.parse(x.toUTCString())
```

However, the expression

```
Date.parse(x.toLocaleString())
```

is not required to produce the same number value as the preceding three expressions and, in general, the value produced by **`Date.parse`** is implementation-dependent when given any string value that could not be produced in that implementation by the **`toString`** or **`toUTCString`** method.

15.9.4.3 Date.UTC (year, month [, date [, hours [, minutes [, seconds [, ms]]]]])

When the **`UTC`** function is called with fewer than two arguments, the behaviour is implementation-dependent. When the **`UTC`** function is called with two to seven arguments, it computes the date from *year*, *month* and (optionally) *date*, *hours*, *minutes*, *seconds* and *ms*. The following steps are taken:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. If *date* is supplied use ToNumber(*date*); else use **1**.
4. If *hours* is supplied use ToNumber(*hours*); else use **0**.
5. If *minutes* is supplied use ToNumber(*minutes*); else use **0**.
6. If *seconds* is supplied use ToNumber(*seconds*); else use **0**.
7. If *ms* is supplied use ToNumber(*ms*); else use **0**.
8. If Result(1) is not **NaN** and $0 \leq$ ToInteger(Result(1)) ≤ 99, Result(8) is 1900+ToInteger(Result(1)); otherwise, Result(8) is Result(1).
9. Compute MakeDay(Result(8), Result(2), Result(3)).
10. Compute MakeTime(Result(4), Result(5), Result(6), Result(7)).
11. Return TimeClip(MakeDate(Result(9), Result(10))).

The **`length`** property of the **`UTC`** function is **7**.

NOTE
The UTC function differs from the Date constructor in two ways: it returns a time value as a number, rather than creating a Date object, and it interprets the arguments in UTC rather than as local time.

15.9.5 Properties of the Date Prototype Object

The Date prototype object is itself a Date object (its [[Class]] is **`"Date"`**) whose value is **NaN**.

The value of the internal [[Prototype]] property of the Date prototype object is the Object prototype object (15.2.3.1).

In following descriptions of functions that are properties of the Date prototype object, the phrase "this Date object" refers to the object that is the **this** value for the invocation of the function. None of these functions are generic; a **TypeError** exception is thrown if the **this** value is not an object for which the value of the internal [[Class]] property is **`"Date"`**. Also, the phrase "this time value" refers to the number value for the time represented by this Date object, that is, the value of the internal [[Value]] property of this Date object.

15.9.5.1 Date.prototype.constructor

The initial value of **`Date.prototype.constructor`** is the built-in **`Date`** constructor.

15.9.5.2 Date.prototype.toString ( )

This function returns a string value intended to represent the Date in the current time zone in a convenient, human-readable form. The format of the string is as called out in Date Time string format (15.1.9.15).

NOTE For any Date value `d` whose milliseconds amount is zero, the result of `Date.parse(d.toString())` is equal to `d.valueOf()`. See section 15.9.4.2.

15.9.5.3 Date.prototype.toDateString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the "date" portion of the Date in the current time zone in a convenient, human-readable form.

15.9.5.4 Date.prototype.toTimeString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the "time" portion of the Date in the current time zone in a convenient, human-readable form.

15.9.5.5 Date.prototype.toLocaleString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

NOTE
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.9.5.6 Date.prototype.toLocaleDateString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the "date" portion of the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

NOTE
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.9.5.7 Date.prototype.toLocaleTimeString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the "time" portion of the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

NOTE
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.9.5.8 Date.prototype.valueOf ( )

The **`valueOf`** function returns a number, which is this time value.

15.9.5.9 Date.prototype.getTime ( )

1. If the **this** value is not an object whose [[Class]] property is **`"Date"`**, throw a **TypeError** exception.
2. Return this time value.

15.9.5.10 Date.prototype.getFullYear ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return YearFromTime(LocalTime(*t*)).

15.9.5.11 Date.prototype.getUTCFullYear ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.

Deleted: . The contents of the string are implementation-dependent, but are

Comment [pL3]: This Note is based on the ES3 errata.

Deleted: *NOTE*¶
It is intended that for any Date value `d`, the result of `Date.prototype.parse(d.toString())` (15.9.4.2) is equal to `d`.

3. Return YearFromTime(*t*).

15.9.5.12 Date.prototype.getMonth ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MonthFromTime(LocalTime(*t*)).

15.9.5.13 Date.prototype.getUTCMonth ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MonthFromTime(*t*).

15.9.5.14 Date.prototype.getDate ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return DateFromTime(LocalTime(*t*)).

15.9.5.15 Date.prototype.getUTCDate ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return DateFromTime(*t*).

15.9.5.16 Date.prototype.getDay ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return WeekDay(LocalTime(*t*)).

15.9.5.17 Date.prototype.getUTCDay ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return WeekDay(*t*).

15.9.5.18 Date.prototype.getHours ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return HourFromTime(LocalTime(*t*)).

15.9.5.19 Date.prototype.getUTCHours ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return HourFromTime(*t*).

15.9.5.20 Date.prototype.getMinutes ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MinFromTime(LocalTime(*t*)).

15.9.5.21 Date.prototype.getUTCMinutes ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MinFromTime(*t*).

15.9.5.22 Date.prototype.getSeconds ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return SecFromTime(LocalTime(*t*)).

15.9.5.23 Date.prototype.getUTCSeconds ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return SecFromTime(*t*).

15.9.5.24 Date.prototype.getMilliseconds ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return msFromTime(LocalTime(*t*)).

15.9.5.25 Date.prototype.getUTCMilliseconds ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return msFromTime(*t*).

15.9.5.26 Date.prototype.getTimezoneOffset ( )

Returns the difference between local time and UTC time in minutes.

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return (*t* – LocalTime(*t*)) / msPerMinute.

15.9.5.27 Date.prototype.setTime (time)

1. If the **this** value is not a Date object, throw a **TypeError** exception.
2. Call ToNumber(*time*).
3. Call TimeClip(Result(1)).
4. Set the [[Value]] property of the **this** value to Result(2).
5. Return the value of the [[Value]] property of the **this** value.

15.9.5.28 Date.prototype.setMilliseconds (ms)

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*ms*).
3. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), SecFromTime(*t*), Result(2)).
4. Compute UTC(MakeDate(Day(*t*), Result(3))).
5. Set the [[Value]] property of the **this** value to TimeClip(Result(4)).
6. Return the value of the [[Value]] property of the **this** value.

15.9.5.29 Date.prototype.setUTCMilliseconds (ms)

1. Let *t* be this time value.
2. Call ToNumber(*ms*).
3. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), SecFromTime(*t*), Result(2)).
4. Compute MakeDate(Day(*t*), Result(3)).
5. Set the [[Value]] property of the **this** value to TimeClip(Result(4)).
6. Return the value of the [[Value]] property of the **this** value.

15.9.5.30 Date.prototype.setSeconds (sec [, ms])

If *ms* is not specified, this behaves as if *ms* were specified with the value getMilliseconds( ).

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*sec*).
3. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
4. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), Result(2), Result(3)).
5. Compute UTC(MakeDate(Day(*t*), Result(4))).
6. Set the [[Value]] property of the **this** value to TimeClip(Result(5)).
7. Return the value of the [[Value]] property of the **this** value.

The **`length`** property of the **`setSeconds`** method is **2**.

15.9.5.31 Date.prototype.setUTCSeconds (sec [, ms])

If *ms* is not specified, this behaves as if *ms* were specified with the value getUTCMilliseconds( ).

1. Let *t* be this time value.
2. Call ToNumber(*sec*).
3. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
4. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), Result(2), Result(3)).
5. Compute MakeDate(Day(*t*), Result(4)).
6. Set the [[Value]] property of the **this** value to TimeClip(Result(5)).
7. Return the value of the [[Value]] property of the **this** value.

The **`length`** property of the **`setUTCSeconds`** method is **2**.

15.9.5.33 Date.prototype.setMinutes (min [, sec [, ms]])

If *sec* is not specified, this behaves as if *sec* were specified with the value getSeconds( ).

If *ms* is not specified, this behaves as if *ms* were specified with the value getMilliseconds( ).

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*min*).
3. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
4. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
5. Compute MakeTime(HourFromTime(*t*), Result(2), Result(3), Result(4)).
6. Compute UTC(MakeDate(Day(*t*), Result(5))).
7. Set the [[Value]] property of the **this** value to TimeClip(Result(6)).
8. Return the value of the [[Value]] property of the **this** value.

The **`length`** property of the **`setMinutes`** method is **3**.

15.9.5.34 Date.prototype.setUTCMinutes (min [, sec [, ms]])

If *sec* is not specified, this behaves as if *sec* were specified with the value getUTCSeconds( ).

If *ms* is not specified, this behaves as if *ms* were specified with the value getUTCMilliseconds( ).

1. Let *t* be this time value.
2. Call ToNumber(*min*).
3. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
4. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
5. Compute MakeTime(HourFromTime(*t*), Result(2), Result(3), Result(4)).
6. Compute MakeDate(Day(*t*), Result(5)).
7. Set the [[Value]] property of the **this** value to TimeClip(Result(6)).
8. Return the value of the [[Value]] property of the **this** value.

The **`length`** property of the **`setUTCMinutes`** method is **3**.

15.9.5.35 Date.prototype.setHours (hour [, min [, sec [, ms]]])

If *min* is not specified, this behaves as if *min* were specified with the value getMinutes( ).

If *sec* is not specified, this behaves as if *sec* were specified with the value getSeconds( ).

If *ms* is not specified, this behaves as if *ms* were specified with the value getMilliseconds( ).

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*hour*).
3. If *min* is not specified, compute MinFromTime(*t*); otherwise, call ToNumber(*min*).
4. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
5. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
6. Compute MakeTime(Result(2), Result(3), Result(4), Result(5)).
7. Compute UTC(MakeDate(Day(*t*), Result(6))).
8. Set the [[Value]] property of the **this** value to TimeClip(Result(7)).
9. Return the value of the [[Value]] property of the **this** value.

The **`length`** property of the **`setHours`** method is **4**.

15.9.5.36 Date.prototype.setUTCHours (hour [, min [, sec [, ms]]])

If *min* is not specified, this behaves as if *min* were specified with the value getUTCMinutes( ).

If *sec* is not specified, this behaves as if *sec* were specified with the value getUTCSeconds( ).

If *ms* is not specified, this behaves as if *ms* were specified with the value getUTCMilliseconds( ).

1. Let *t* be this time value.
2. Call ToNumber(*hour*).
3. If *min* is not specified, compute MinFromTime(*t*); otherwise, call ToNumber(*min*).
4. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
5. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
6. Compute MakeTime(Result(2), Result(3), Result(4), Result(5)).
7. Compute MakeDate(Day(*t*), Result(6)).
8. Set the [[Value]] property of the **this** value to TimeClip(Result(7)).
9. Return the value of the [[Value]] property of the **this** value.

The **`length`** property of the **`setUTCHours`** method is **4**.

15.9.5.36 Date.prototype.setDate (date)

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*date*).
3. Compute MakeDay(YearFromTime(*t*), MonthFromTime(*t*), Result(2)).
4. Compute UTC(MakeDate(Result(3), TimeWithinDay(*t*))).
5. Set the [[Value]] property of the **this** value to TimeClip(Result(4)).
6. Return the value of the [[Value]] property of the **this** value.

15.9.5.37 Date.prototype.setUTCDate (date)

1. Let *t* be this time value.
2. Call ToNumber(*date*).
3. Compute MakeDay(YearFromTime(*t*), MonthFromTime(*t*), Result(2)).
4. Compute MakeDate(Result(3), TimeWithinDay(*t*)).
5. Set the [[Value]] property of the **this** value to TimeClip(Result(4)).
6. Return the value of the [[Value]] property of the **this** value.

15.9.5.38 Date.prototype.setMonth (month [, date])

If *date* is not specified, this behaves as if *date* were specified with the value getDate( ).

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*month*).
3. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
4. Compute MakeDay(YearFromTime(*t*), Result(2), Result(3)).
5. Compute UTC(MakeDate(Result(4), TimeWithinDay(*t*))).
6. Set the [[Value]] property of the **this** value to TimeClip(Result(5)).
7. Return the value of the [[Value]] property of the **this** value.

The **`length`** property of the **`setMonth`** method is **2**.

15.9.5.39 Date.prototype.setUTCMonth (month [, date])

If *date* is not specified, this behaves as if *date* were specified with the value getUTCDate( ).

1. Let *t* be this time value.
2. Call ToNumber(*month*).
3. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
4. Compute MakeDay(YearFromTime(*t*), Result(2), Result(3)).
5. Compute MakeDate(Result(4), TimeWithinDay(*t*)).
6. Set the [[Value]] property of the **this** value to TimeClip(Result(5)).
7. Return the value of the [[Value]] property of the **this** value.

The **`length`** property of the **`setUTCMonth`** method is **2**.

15.9.5.40 Date.prototype.setFullYear (year [, month [, date]])

If *month* is not specified, this behaves as if *month* were specified with the value getMonth( ).

If *date* is not specified, this behaves as if *date* were specified with the value getDate( ).

1. Let *t* be the result of LocalTime(this time value); but if this time value is **NaN**, let *t* be **+0**.
2. Call ToNumber(*year*).
3. If *month* is not specified, compute MonthFromTime(*t*); otherwise, call ToNumber(*month*).
4. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
5. Compute MakeDay(Result(2), Result(3), Result(4)).
6. Compute UTC(MakeDate(Result(5), TimeWithinDay(*t*))).
7. Set the [[Value]] property of the **this** value to TimeClip(Result(6)).
8. Return the value of the [[Value]] property of the **this** value.

The **`length`** property of the **`setFullYear`** method is **3**.

15.9.5.41 Date.prototype.setUTCFullYear (year [, month [, date]])

If *month* is not specified, this behaves as if *month* were specified with the value getUTCMonth( ).

If *date* is not specified, this behaves as if *date* were specified with the value getUTCDate( ).

1. Let *t* be this time value; but if this time value is **NaN**, let *t* be **+0**.
2. Call ToNumber(*year*).
3. If *month* is not specified, compute MonthFromTime(*t*); otherwise, call ToNumber(*month*).
4. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
5. Compute MakeDay(Result(2), Result(3), Result(4)).
6. Compute MakeDate(Result(5), TimeWithinDay(*t*)).
7. Set the [[Value]] property of the **this** value to TimeClip(Result(6)).
8. Return the value of the [[Value]] property of the **this** value.

The **`length`** property of the **`setUTCFullYear`** method is **3**.

15.9.5.42 Date.prototype.toUTCString ( )

This function returns a string value intended to represent the Date in a convenient, human-readable form in UTC. The format of the string is as called out in Date Time string format (15.1.9.15).

Deleted: . The contents of the string are implementation-dependent, but are

15.9.6 Properties of Date Instances

Date instances have no special properties beyond those inherited from the Date prototype object.