# DateTime
## Date Calculation Utilities

Davin Church
Creative Software Design
November 3, 2020

# Table of Contents

## Introduction

This document describes a Dyalog namespace that contains a set of date manipulation routines written in APL by Davin Church of Creative Software Design for use in APL applications. Use of this code for any purpose is hereby granted, but credit (comments left inside the code, for instance) is requested and modification is not generally recommended (to allow for later updates).

These functions are written for use by Dyalog APL v18.0 and make use of the `⎕DT` and `1200⌶` capabilities available in that and later versions of the interpreter. They are designed to provide additional application-oriented capabilities beyond those supplied by `⎕DT`. However, all routines except for `Spell` will also run in prior versions where `⎕DT`/`1200⌶` is not available.

All these functions are designed to process integers that contain dates in Julian Day Number form, equivalent to `50 ⎕DT` form. `⎕DT` may be used to create values in that form from any other desired structure. Two functions are also available here to convert between Julian and Gregorian forms as well. These converters (and most of the other functions) also handle dates far outside the range supported by `⎕DT` and allow for additional features as well.

Simple day-based calculations can be performed by simply adding and subtracting constants to the `JDN`, such as adding or subtracting the constant 2415019 to convert between a `JDN` and the sorts of date values used by Microsoft Office for Windows. Routines are provided here to perform calculations on calendar months and years as well as other sorts of complex calendar operations.

The following routines are currently available (functionally grouped):

| | |
|---|---|
| `JD` | Convert a calendar date (*YYYYMMDD*) to a Julian Day Number (*JJJJJJJ*). |
| `GD` | Convert a Julian Day Number (*JJJJJJJ*) to a calendar date (*YYYYMMDD*). |
| Today | Return today's date as a Julian Day Number. |
| Now | Return a timestamp, like Today but including a fractional time-of-day. |
| Age | Return the calendar age from a `JD`N as of today. |
| DayOfWeek | Return the day of the week (Sun=1 thru Sat=7) of a `JD`N. |
| DayOfYear | Return the day of the calendar year (1-366) of a `JD`N. |
| IsLeapYear | Is the given year a leap year? |
| WeekOfYear | Return the week of the calendar year (1-53) of a `JD`N. |
| AddMonth | Return the date that is one (or more) months from the date given. |
| AddYear | Return the date that is one (or more) years from the date given. |
| AddWeekday | Return the `JD`N of the following week day of the given `JD`N. |
| AddWorkday | Return a date as in AddWeekday but also skip holidays. |
| ExampleHoliday | For a given date, decide if it is a U.S. national bank holiday. |
| BeginWeek | Return the `JD`N of the Sunday on or just before the given `JD`N. |
| BeginMonth | Return the date of the first of the month of the date given. |
| BeginYear | Return the date of the first of the year of the date given. |
| Easter | For a given year, return the `JD`N date of the Easter holiday. |
| Spell | For a given date or timestamp, spell it out in any specified format. |
| Local | Convert a UTC `JD`N timestamp to a local `JD`N timestamp. |
| UTC | Convert a local `JD`N timestamp to a UTC `JD`N timestamp. |
| Daylight | Return local Daylight Savings Time rules. |
| TimeZone | Return a basic description of the local time zone. |

## General Information and Background

Most routines will accept an array of any size, shape, or rank and return an array of the same dimensions. Operations on arrays of dates are fast and encouraged. All date routines (except `JD` and `GD`) use the Julian Day Number style of date for simplicity. The *YYYYMMDD* format is primarily used when passing dates to and from other systems. Other forms can be translated by using `⎕DT`.

Dates in the (calendar) form of *YYYYMMDD* are single 8-digit numeric values where groups of digits are used to describe the parts of the date. MM is the month number (01-12) and DD is the day number of the month (01-31). The year (*YYYY*) may take on almost any value, positive or negative. If the year is negative, this is indicated by making the entire date a negative number, but without affecting the month or day digits (i.e. they still appear to read as the correct decimal values). Negative dates (years) are used to indicate dates prior to 1 A.D. Since there was no year 0 (A.D. or B.C.), the year number (*YYYY*) of `0` is used to represent the calendar year 1 B.C. so that years can be consecutive. A year number (*YYYY*) of `¯1` is used to represent the year 2 B.C., and so on. Dates are using a zero date on the proleptic Julian calendar of January 1, 4713 B.C. (`¯47120101`), and most calculations can be performed on all dates both before and after that starting point.

Important note: Dates from 15 Oct 1582 and later are assumed to represent dates in the Gregorian (reform) calendar and dates from 4 Oct 1582 and earlier are assumed to represent dates in the (proleptic) Julian calendar, even those that are prior to the official adoption of the Julian calendar in 46 B.C. Other variations and complications in the use of historical dates (such as those surrounding and immediately following the adoption of the Julian calendar system, from 46 B.C. to 4 A.D., or the variable date of switchover to the Gregorian calendar) are summarily ignored. This is standard practice in many professional arenas.

The "Julian Day Number" (also called the Julian Day or `JD`, but <u>not</u> the Julian Date) form of the dates is the standard way of consistently measuring dates regardless of the local calendar in use at any given time. Its origin is at (Julian proleptic date) January 1, 4713 B.C. where the `JD` = 0. In many Julian Day Number systems, dates may be assigned fractional values so that a time of day (timestamp) might be indicated with the date. In such systems, the whole value of the Julian Day often represents noon on that day, so the previous midnight would be 0.5 days less and the following midnight would be 0.5 days more. `⎕DT` timestamps follow this convention. Such fractional day numbers are not supported by most of these routines and all dates should be considered to be the whole number (noontime) value for that day. However, `JD` and `GD` (and a few other routines) have been modified to allow special "timestamp" formats for convenience, and these can accept and produce special fractional-day values which start at <u>midnight</u> rather than noon (and thus are 12 hours or 0.5 days different than `⎕DT` day-fractions noted above). This allows timestamps herein to represent any time during that date by using a single integer value plus a variable time-fraction, just as human experience usually assumes, with a contiguous time-fraction throughout a single day.

The conversion routines were also modified slightly so that a Julian Day (*JJJJJJJ*) of exactly zero is returned and accepted when the calendar date (*YYYYMMDD*) is exactly zero. This is to facilitate the handling of logically null (e.g. empty, undefined) dates by use of the value zero in either format. Unfortunately, this means that the exact (Julian proleptic) date of January 1, 4713 B.C. (`¯47120101`) is not fully available for use since it also equates to day number zero. This is not expected to pose any practical difficulties to programmers in the real world, however.

The basis for the date conversion algorithms was derived from a detailed mathematical analysis by Peter Baum in 1998 and additional formulas and algorithms were derived from the current "calendar FAQ". At that time, Peter's information was available from his web site at *http://mysite.verizon.net/aesir_research/date/date0.htm* and the FAQ is usually multi-posted to the following Usenet news groups: *sci.astro*, *soc.history*, *sci.answers*, *soc.answers*, and *news.answers*. Wikipedia also has lots of useful and detailed information available. Please consult these extensive sources of information for more details on the inner workings of the algorithms and the historical use of calendars through the centuries.

## Syntax and Description

### *AddMonth*

*JDN* ← *[months]* `AddMonth` *JDN*

Add one calendar month to the date given and return its date. Zero, or more than one month may be added, or one or more months may be subtracted, by giving the routine a left argument specifying the number of months to add (subtract if negative). If the resulting day of the month (e.g. 31st) is not part of the resulting month (e.g. April), then the first legal day following it is returned instead (e.g. May 1st).

### *AddWeekday*

*JDN* ← *[weekdays]* `AddWeekday` *JDN*

Add one week day (Monday-Friday) to the date given and return its date. Zero, or more than one week day may be added, or one or more week days may be subtracted, by giving the routine a left argument specifying the number of week days to add (subtract if negative). If the starting date is not a week day, then the starting date is shifted forward to Monday before counting begins.

### *AddWorkday*

*JDN* ← *[workdays]* *holidayfilterfn* `AddWorkday` *JDN*

Add one non-holiday week day to the date given and return its date Zero or more work days may be added, or one or more work days may be subtracted, by giving the derived function a left argument specifying the number of work days to add (subtract if negative). If the starting date is not a work day, then the starting date is shifted forward to the next legal work day before counting begins.

This routine is identical to the `AddWeekday` function except that it skips over holidays as well as weekends. This utility is actually an operator instead of a function and it takes, as its required left operand, a programmer-provided function that determines whether or not a particular Julian Day Number is to be considered a holiday or not. For each argument date it should return a `1` (for a holiday) or a 0 (for a work day). The included function `ExampleHoliday` is an example of this sort of operand and reports official U.S. banking holidays. The programmer is expected to create their own holiday-reporting function to provide the desired results for their own needs.

## AddYear

*JDN* ← *[years]* `AddYear` *JDN*

Add one calendar year to the date given and return its date. Zero, or more than one year may be added, or one or more years may be subtracted, by giving the routine a left argument specifying the number of years to add (subtract if negative). If the starting day of the year (e.g. Feb 29th in a leap year) is not part of the resulting year (e.g. it is not a leap year), then the first legal day following it is returned instead (e.g. March 1st).

## Age

*ageinyears* ← *[today]* `Age` *JDN*

For a given starting date, return a calendar age (in calendar years and fractions thereof) that have elapsed from that date until today. A different ending date may be specified in *JDN* format as the optional left argument. Fractional years are computed by counting days between the last "birthday" and the next, and thus there may not always be exact half and quarter years for some dates. Previous, next, and nearest whole ages may be extracted by using floor (`⌊`), ceiling (`⌈`), and round (`⌊.5+`), respectively.

## BeginMonth

*JDN* ← *[start]* `BeginMonth` *JDN*

For a given date, return the date of the first day of the calendar month containing that date. If a month is to be considered to begin on a different day of the month (on the 5th, or the 25th, or other unusual boundary), then that day number (`5`, `25`, etc.) may be provided as a left argument to the routine to logically shift the notion of the beginning of the month.

## BeginWeek

*JDN* ← *[start]* `BeginWeek` *JDN*

For a given date, return the date of the first day (Sunday) of the week containing that date (i.e. the Sunday on or immediately preceding the date given). If a week is considered to begin on a different day of the week (on Monday or Saturday for example), then that day number (`2`, `7`, etc.) may be provided as a left argument to the routine to logically shift the notion of the beginning of the week.

## BeginYear

*JDN* ← *[start]* `BeginYear` *JDN*

For a given date, return the date of the first day of the calendar year containing that date. If a year is to be considered to begin on a different day of the year (on March 1st, or December 25th, or other unusual boundary such as those representing fiscal tax years), then that month and day (as a 4-digit number: `0301`, `1225`, etc.) may be provided as a left argument to the routine to logically shift the notion of the beginning of the year. Alternatively, a day-of-week number (`1-7`) may be specified to indicate that that day of the first calendar week is to be used as the logical beginning of the year. A `0` (the default) indicates the actual calendar year start.

## DayOfWeek

*dayofweek* ← `DayOfWeek` *JDN*

For any given Julian Day Number, return the day of the week on which it falls. Sunday is given as 1 and Saturday as 7.

## DayOfYear

*dayofyear* ← *[start]* `DayOfYear` *JDN*

For any given Julian Day Number, return the day of that year on which it falls. January 1 is given as 1 and December 31 as 365 or 366. If a year is to be considered to begin on a different day (on March 1st, or December 25th, or other unusual boundary such as those representing fiscal tax years), then that month and day (as a 4-digit number: `0301`, `1225`, etc.) may be provided as a left argument to the routine to logically shift the notion of the beginning of the year. Alternatively, a day-of-week number (`1-7`) may be specified to indicate that that day of the first calendar week is to be used as the logical beginning of the year. A `0` (the default) indicates the actual calendar year start.

## Daylight

*rule* ← `Daylight` *year*

Return the Daylight Savings Time rules for the specified year(s) in the local time zone. The result is in the same shape as the argument, but with each item being nested and containing three numeric values for that year:

| | |
|---|---|
| [1] | Number of hours to add to the time zone during DST |
| [2] | The *JDN* timestamp when DST begins in that year |
| [3] | The *JDN* timestamp when DST ends in that year |

If the argument is a scalar (not a `1⍴` vector) then the result is returned unnested for convenience.

## Easter

*JDN* ← `Easter` *year*

Return the date that Easter falls on in a given year. This is a difficult calculation. The rule used is: Easter Sunday is the first Sunday after the ecclesiastical full moon on or after the ecclesiastical vernal equinox. The ecclesiastical vernal equinox is always March 21st. The astronomical vernal equinox may actually occur on the 19th or 20th and the ecclesiastical full moon may differ from the actual astronomical full moon by a day either way. There are also variances to take into account for effective longitude and the effects of the International Date Line. The ecclesiastical rules define how these effects are all handled. This function uses the above rule for Gregorian calendar dates (after 1582) and a simpler calculation for older Julian calendar dates (before 1583).

## ExampleHoliday

*isholiday* ← `ExampleHoliday` *JDN*

For use as a left operand to `AddWorkday`, this is an example function for determining holidays. Return a `1` if the given date is a recognized U.S. national banking holiday or a `0` otherwise. The following holidays are recognized (on date observed):

| | | |
|---|---|---|
| New Years Day | Independence Day | Thanksgiving |
| MLK's Birthday | Labor Day | Christmas |
| President's Day | Columbus Day | |
| Memorial Day | Veterans Day | |

## GD

*YYYYMMDD* ← `GD` *JDN*
*YYYY MM DD* ← `3 GD` *JDN*
*YYYY MM DD HH MM SS TTT* ← `7 GD` *JDN*

Converts a (serial) Julian Day Number to a (calendar-form) Gregorian date. If *JDN* is zero, the returned *YYYYMMDD* result will also be zero to facilitate the implementation of logically null dates. To further enhance the ability to interface this with other code and systems, it is also possible to have the dates returned as nested numeric vectors which are the usual scalar dates separated into independent year, month, and day values similar to "`⊂3↑⎕TS`". This form of output is produced by specifying a scalar 3 as the optional left argument to the function. If a fractional Julian Day Number is provided, then a fractional result will be returned giving the time in a human-readable *YYYYMMDD.HHMMSSTTT* format. If full timestamps are desired in enclosed ⎕TS form, supply a left argument of 7. (Any number between 1 and 7 may be used instead for shorter nested results if desired.) Note that internal floating-point precision restrictions may render exact milliseconds inaccurately when returning a combined floating-point value.

## IsLeapYear

*isleap* ← `IsLeapYear` *year*

For any given year, return a `1` if it is a leap year or a `0` if not. This calculation works both before and after the adoption of the Gregorian calendar, using the currently-standard rules extending indefinitely further into both the past (even before adoption of the Julian calendar) and into the future (even beyond 2800 and other proposed leap-year changes).

## JD

*JDN* ← `JD` *YYYYMMDD*
*JDN* ← `JD` ⊂*YYYY MM DD*
*JDN* ← `JD` *YYYYMMDD.HHMMSSTTT*
*JDN* ← `JD` ⊂*YYYY MM DD HH MM SS TTT*

Converts a (calendar-form) Gregorian date to a (serial) Julian Day Number. If *YYYYMMDD* is a negative value, this indicates that the year itself is negative (not affecting the month or day) and is used to describe a date B.C. If *YYYYMMDD* is zero, then the returned *JDN* result will also be zero to facilitate the implementation of logically null dates. Input dates may also be given in expanded (`⎕TS`-like) format where the dates are provided as separate year, month, and day numbers, as long as they are nested together into an enclosed scalar anywhere a simple 8-digit scalar would normally be expected (e.g. `⊂3↑⎕TS`). Also, a time of day may be included to specify a timestamp value, either by including a decimal fraction on the date integer with the time in readable form (*YYYYMMDD.HHMMSSTTT*) or by extending the expanded format to have up to 7 items in the nested vector. In either case, the resulting Julian Day Number will no longer be an integer, but instead will contain the usual day value plus a fractional amount of a day representing the provided time of day since midnight. If any component of the argument value is out-of-range (such as an impossible day-of-month), it will be wrapped to an equivalent legal date. This and other special features mentioned provide low-cost extensions to `⎕DT`'s capabilities. Note that internal floating-point precision restrictions may not exactly represent milliseconds, so an incoming floating-point value is rounded to 10 milliseconds before conversion.

## Local

*JDN* ← `Local` *JDN*

Given a UTC (Coordinated Universal Time) timestamp in fractional Julian Day Number form, convert it to a local timezone timestamp in similar form. This conversion process respects any Daylight Savings Time adjustment that was in effect on that date.

## *Now*

*JDN* ← `Now`

Return the current date and time (`⎕TS`) in fractional *JDN* form for use where a combined timestamp value is preferred. This is the same as `Today` (a Julian Day Number), except that it also contains the current time with an additional fractional portion indicating the time of day (e.g. X.0 = midnight, X.5 = noon, X.75 = 6pm). This form can be used as a full timestamp and can be processed directly by routines designed for that. It is also recognized by `GD` (which can then return a fractional or expanded-long result) and `Spell`. The date portion may be extracted with "`⌊`" and the time portion extracted with "`1|`". The implementation is simply "`JD ⊂⎕TS`". Note that most of the other functions here do not accept fractional inputs.

## *Spell*

*text* ← *pattern* `Spell` *JDN*

Format a given date or timestamp as text. The format to be used is specified by use of a *pattern* phrase provided as the left argument. `Spell` is a cover function that calls the system facility `1200⌶` and the pattern to be provided is defined by that facility. Timestamps to be formatted must fall within the range limits set by `1200⌶` and `⎕DT`. Examples of patterns include:

| |
|---|
| MMM D, *YYYY* |
| *YYYY*-MM-DD |
| MM/DD/YY |
| t:mmp "on" Dddd, Mmmm Doo, *YYYY* |

Formatted text is returned as a variable-length character vector. A vector or matrix of timestamps is returned as a vector or matrix of nested character vectors. However, a scalar (not a `1⍴` vector) timestamp value is returned as an unnested character vector for convenience. A logically null timestamp (JDN = `0`), or a value of `⎕NULL`, is formatted as an empty character vector (regardless of the pattern format requested) to conveniently deal with missing or undefined timestamps. This additional functionality extends the definition of `1200⌶`.

## TimeZone

*tzinfo* ← `TimeZone`

Return a nested vector of static information about the local time zone, as follows:

| | |
|---|---|
| [1] | Number of standard-time hours offset from UTC (Coordinated Universal Time) |
| [2] | Name of time zone |
| [3] | Descriptive title of time zone |
| [4] | Title of standard time zone |
| [5] | Title of daylight savings time zone |

## Today

*JDN* ← `Today`

Return the current date (`⎕TS`) in *JDN* form for use by these routines or any other process that can use standard Julian Day Numbers  The implementation is simply "`JD 100⊥3↑⎕TS`".

## UTC

*JDN* ← `UTC` *JDN*

Given a local timestamp in fractional Julian Day Number form, convert it to a UTC (Coordinated Universal Time) timezone timestamp in similar form.  This conversion process respects any Daylight Savings Time adjustment that was in effect on that date.

## WeekOfYear

*weekofyear* ← *[start]* `WeekOfYear` *JDN*

For any given Julian Day Number, return the week of that year on which it falls.  January 1 would return 1 and December 31 returns 53.  If a year is to be considered to begin on a different day (on March 1st, or December 25th, or other unusual boundary such as those representing fiscal tax years), then that month and day (as a 4-digit number: `0301`, `1225`, etc.) may be provided as a left argument to the routine to logically shift the notion of the beginning of the year.  Alternatively, a day-of-week number (`1-7`) may be specified to indicate that that day of the first calendar week is to be used as the logical beginning of the year.  A `0` (the default) indicates the actual calendar year start.

If ISO weeks are needed, these may be calculated using these routines without too much difficulty or `⎕DT` can do the job much more simply.

## Examples of Application Use

```
      Today+7
```
A week from today.

```
      14 ExampleHoliday AddWorkday ω
```
14 working days from a starting date.

```
      ¯1+AddMonth BeginMonth ω
```
Find the last day of the month.

```
      6 BeginWeek ω
```
Find the most-recent Friday.

```
      ⌈7÷⍨1+ω-BeginMonth ω
```
Determine the week of the calendar month.

```
      2 BeginWeek 7+AddMonth BeginMonth ω
```
Determine the first Monday of next month.

```
      ¯3 ExampleHoliday AddWorkday AddMonth BeginMonth ω
```
The last day of the month a shipment could be sent to arrive by the first workday of next month, assuming it takes 3 working days to arrive.

```
      2415019+ω
```
Convert a date-value from Microsoft Excel into JDN notation.

```
      2 BeginWeek 7+BeginYear ω
```
The first Monday of the year.

```
      6 BeginWeek ¯8+AddMonth BeginMonth ω
```
The next-to-last Friday of the month.

```
      WW {ω+7×(α=5)∧=/BeginMonth ω+0 7} …
         DOW BeginWeek JD ⊂YYYY,MM,7×4⌊WW
```
Compute the date from a year, month, week (up to 5), and day-of-week.

```
      BeginWeek 7+AddMonth BeginMonth ω
```
The first Sunday of the following month.