NEWS · JULY 27, 2026

DateUtils, Teil 1: Datum und Uhrzeit in Delphi

By Dr. Holger Flick



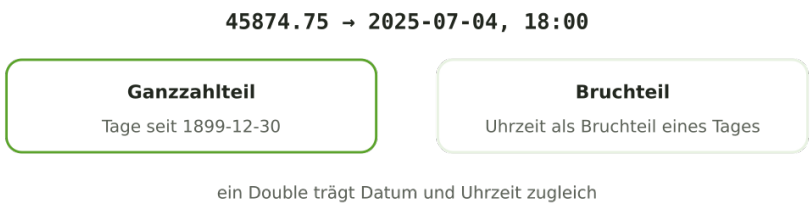
Datums- und Zeitcode hat die Angewohnheit, selbst erfahrene Entwickler leise zu demütigen. Sie brauchen "den ersten Tag des nächsten Monats", "wie viele Tage bis zur Deadline" oder "dieselbe Uhrzeit gestern" — und ertappen sich dabei, wie Sie zu `DecodeDate` greifen, mit den Einzelteilen rechnen, sich über Monatslängen den Kopf zerbrechen und alles mit `EncodeDate` wieder zusammensetzen. Es funktioniert — Delphis `TDateTime` ist seit jeher grundsolide — aber es ist mehr Zeremonie, als das Problem verdient.

Es gibt ein gut gehütetes Geheimnis in der RTL: die Unit `System.DateUtils`. Sie ist seit Delphi 6 dabei, ist über die Jahre stetig gewachsen und verwandelt die meisten dieser fummeligen Berechnungen in einen einzigen, lesbaren Funktionsaufruf. "Erster Tag des nächsten Monats" wird zu `StartOfTheMonth(IncMonth(Now))`. "Tage bis zur Deadline" wird zu `DaysBetween(Now, Deadline)`. Kein Dekodieren, kein Zusammensetzen, keine Off-by-one-Monatsrechnung.

Dies ist eine zweiteilige praktische Tour durch `DateUtils`, wie gewohnt im Vergleich alter Weg vs. neuer Weg. **Teil 1 — dieser hier — behandelt alltägliche Datums- und Zeitoperationen**: zunächst, was ein `TDateTime` wirklich ist, dann das Addieren und Subtrahieren von Zeit, das Messen von Zeitspannen, das Vergleichen von Daten und das Einrasten auf Grenzen wie "Wochenanfang". Teil 2 nimmt sich den wirklich schwierigen Teil vor — **Zeitzonen, UTC und Austauschformate wie ISO 8601** —, also genau den Bereich, in dem Datumsfehler von ärgerlich zu teuer werden. Räumen wir Ihren Datumscode auf.

Zuerst: Was ein `TDateTime` eigentlich ist

Erstaunlich viel Datumsverwirrung verfliegt, sobald man die eine Tatsache kennt, die allem zugrunde liegt: Ein `TDateTime` ist einfach eine **Gleitkommazahl**. Der *ganzzahlige* Teil zählt Tage seit einer Epoche (Mitternacht am 30. Dezember 1899), und der *gebrochene* Teil ist die Uhrzeit als Bruchteil von 24 Stunden. `0.5` ist also Mittag; `1.75` ist 18 Uhr am Tag nach der Epoche.



Ein TDateTime ist ein Double: Ganzzahlteil = Tage seit 1899, Bruchteil = Uhrzeit

Das Diagramm erklärt vieles auf einmal. Weil ein `TDateTime` eine einzige Zahl ist, bewegt Sie das Addieren von `1` genau einen Tag vorwärts, und das Addieren von `1/24` eine Stunde — genau *deshalb* funktioniert die klassische Arithmetik überhaupt. Aber genau deshalb ist Selberrechnen auch fehleranfällig: `IncMonth` kann nicht einfach "addiere 30" sein, weil Monate unterschiedlich lang sind. `DateUtils` existiert, damit Sie darüber nie wieder nachdenken müssen.

> **Zwei verwandte Units, die Ihnen daneben begegnen**
> Einige der Funktionen, die man gemeinhin \"DateUtils\" nennt, leben tatsächlich in `System.SysUtils` — namentlich `Now`, `Date`, `Time`, `EncodeDate` und `IncMonth`. `DateUtils` baut darauf auf. In der Praxis haben Sie beide in Ihrer `uses`-Klausel und müssen sich nicht darum kümmern, aus welcher Unit eine bestimmte Funktion stammt; ich weise auf die SysUtils-Funktionen hin, wo es für die Genauigkeit zählt. Alles Übrige unten ist `System.DateUtils`.

Rezept: Zeit addieren und subtrahieren

Der klassische Weg, einen Tag zu addieren, nutzt den Gleitkomma-Trick — `Now + 1` — was clever, aber für den nächsten Leser undurchsichtig ist und in dem Moment zusammenbricht, in dem Sie Monate oder Jahre brauchen. `DateUtils` gibt jeder Einheit ihre eigene benannte Funktion, und sie lesen sich wie Klartext:

```
uses
  System.DateUtils;

var
  T: TDateTime;
begin
  T := IncDay(Now, 3);      // drei Tage ab jetzt
  T := IncDay(Now, -1);     // gestern (negativ zählt rückwärts)
  T := IncHour(Now, 6);     // sechs Stunden ab jetzt
  T := IncWeek(Now, 2);     // zwei Wochen weiter
  T := IncYear(Now, 1);     // dieses Datum, nächstes Jahr
  // IncMinute, IncSecond, IncMilliSecond vervollständigen das Set
end;
```

Jedes `IncXxx` akzeptiert eine negative Zahl, um rückwärts zu gehen, sodass Sie nie eine separate "Subtrahieren"-Funktion brauchen. Und entscheidend: Sie sind *kalenderbewusst* — `IncMonth(EncodeDate(2026, 1, 31), 1)` landet korrekt auf dem 28. Februar 2026, nicht auf einem ungültigen "31. Februar". (`IncMonth` ist

diejenige, die in SysUtils lebt — gut zu wissen, denn genau die versuchen Leute am häufigsten mit Arithmetik nachzubauen, und machen es falsch.)

Rezept: die Spanne zwischen zwei Daten messen

Hier zeigt der alte Ansatz sein Alter am deutlichsten. Das Subtrahieren zweier `TDateTime`-Werte liefert einen `Double` als Anzahl von *Tagen samt Bruchteilen*, den Sie dann konvertieren und runden müssen — und zwar richtig runden. `DateUtils` bietet für jede Einheit eine `XxxBetween`-Funktion, die eine saubere Ganzzahl zurückgibt:

```
DaysBetween(StartDate, EndDate);      // ganze Tage dazwischen
HoursBetween(Clock1, Clock2);         // ganze Stunden
MinutesBetween(A, B);                 // ganze Minuten
MonthsBetween(A, B);                  // ganze Kalendermonate
YearsBetween(BirthDate, Now);         // Alter in Jahren in einer Zeile!
// dazu WeeksBetween, SecondsBetween, MilliSecondsBetween
```

Dass `YearsBetween(BirthDate, Now)` das Alter einer Person in einer lesbaren Zeile berechnet, ist genau die Art von Aufgabe, für die früher jede Codebasis eine kleine Hilfsfunktion hatte. Beachten Sie aber die Semantik, denn sie zählt.

!!! warning "\"Between\" zählt *vollendete* Einheiten — Vorsicht an den Grenzen" Diese Funktionen geben die Anzahl **ganzer** verstrichener Einheiten zurück und schneiden den Rest ab. `DaysBetween(Today10am, Tomorrow9am)` ist `0`, nicht `1`, weil noch keine vollen 24 Stunden vergangen sind. Für Zeitdauern ist das meist genau richtig — aber wenn Sie in *Kalender*-Begriffen denken (\"das sind verschiedene Tage\"), vergleichen Sie stattdessen mit `CompareDate` oder `IsSameDay` — siehe die nächsten Rezepte. Zu wissen, welche Frage Sie stellen — \"wie viel Zeit ist vergangen\" vs. \"sind das verschiedene Kalendertage\" —, ist der ganze Trick zu fehlerfreier Datumsmathematik.

Rezept: Daten so vergleichen, wie Sie es meinen

Der Vergleich von `TDateTime`-Werten mit `<` und `=` vergleicht sie *bis auf die Millisekunde* — was selten das ist, was Sie wollen, wenn ein Benutzer fragt: \"Ist die Rechnung auf heute datiert?\" `DateUtils` gibt Ihnen Vergleichsfunktionen, mit denen Sie die Granularität wählen:

- `SameDate(A, B)` / `IsSameDay(A, D)` — derselbe Kalendertag, Uhrzeit ignoriert.
- `CompareDate(A, B)` — wie `CompareValue`, aber nur auf dem Datumsteil (gibt -1, 0 oder 1 zurück).
- `CompareTime(A, B)` — nur auf dem Uhrzeitteil.
- `WithinPastDays(Now, Then, 7)` — lag `Then` innerhalb der letzten 7 Tage?

```
if IsSameDay(Invoice.Date, Now) then
  ShowMessage('dated today');

if CompareDate(Deadline, Now) < 0 then
  ShowMessage('overdue');   // das Datum der Deadline liegt vor heute
```

Der Mehrwert hier ist *Intention*. `IsSameDay` sagt exakt, was es prüft; ein rohes `Trunc(A) = Trunc(B)` tut dasselbe, zwingt den Leser aber, es zu entschlüsseln. Code, der seine Absicht ausspricht, ist Code, der überlebt.

Rezept: auf eine Grenze einrasten — Anfang/Ende von Tag, Woche, Monat, Jahr

Das ist meine Lieblingsecke der Unit, denn die manuelle Variante ist wirklich lästig korrekt hinzubekommen. \"Der letzte Moment dieses Monats\" heißt zu wissen, wie viele Tage der Monat hat; \"der Anfang dieser Woche\" heißt zu wissen, an welchem Tag Ihre Woche beginnt. `DateUtils` hat einen passenden Satz von `StartOfTheXxx`- / `EndOfTheXxx`-Funktionen, die all das übernehmen:

```
StartOfTheDay(Now);     // heute um 00:00:00.000
EndOfTheDay(Now);       // heute um 23:59:59.999
StartOfTheWeek(Now);    // Montag dieser Woche, um Mitternacht (ISO-8601-Woche)
EndOfTheWeek(Now);
StartOfTheMonth(Now);   // der 1., um Mitternacht
EndOfTheMonth(Now);     // der 28./30./31., letzte Millisekunde – Länge wird für Sie behandelt
StartOfTheYear(Now);
EndOfTheYear(Now);
```

Das sind die Bausteine für nahezu jede Reporting-Abfrage, die Sie je schreiben werden. \"Alle Bestellungen dieses Monats\" ist einfach der Bereich von `StartOfTheMonth(Now)` bis `EndOfTheMonth(Now)` — und dass `EndOfTheMonth` weiß, ob das der 28., 30. oder 31. ist, ob Schaltjahr oder nicht, ist genau die Fleißarbeit, die Sie der RTL überlassen wollen.



StartOfTheMonth / EndOfTheMonth klammern einen ganzen Monat ein — ohne Tagezählen

Das Diagramm ist das Reporting-Muster auf einen Blick: Zwei Funktionsaufrufe klammern den gesamten Monat ein, und jeder Zeitstempel dazwischen gehört dazu — kein Tagezählen, kein Schaltjahr-Sonderfall in Ihrem Code.

Rezept: ein Datum zerlegen (wenn Sie es wirklich brauchen)

Manchmal wollen Sie tatsächlich nur das Jahr oder den Wochentag. Statt `DecodeDate` (das drei `var`-Parameter auf einmal füllt) hat `DateUtils` saubere Einzelwert-Extraktoren:

```
YearOf(Now);            // 2026
MonthOf(Now);           // 8
DayOf(Now);             // 10
DayOfTheWeek(Now);      // 1 = Montag … 7 = Sonntag (ISO 8601)
DayOfTheYear(Now);      // 222  (Tagesnummer innerhalb des Jahres)
DateOf(Now);            // nur der Datumsteil, Zeit genullt
TimeOf(Now);            // nur der Zeitteil, Datum genullt
```

`DateOf` und `TimeOf` sind still und leise nützlich, um einen Zeitstempel auf genau die Hälfte zu reduzieren, die Sie interessiert — die saubere Art, \"Mitternacht dieses Datums\" zu sagen, ist `DateOf(SomeTimestamp)`.

Die andere Seite: wann die klassischen Funktionen weiterhin die richtige Wahl sind

Ein Versprechen an den Leser — bei `DateUtils` geht es um *Lesbarkeit und Korrektheit*, nicht darum, alles zu ersetzen. Ein paar ehrliche Einordnungen:

- Für rohe Performance in einer sehr heißen Schleife über Millionen Zeilen ist direkte Arithmetik auf dem `Double` (`T + 1`) marginal schneller als ein Funktionsaufruf. Das ist den Verlust an Klarheit fast nie wert — aber wenn Sie profiliert haben und es zählt, steht Ihnen die Option offen.
- `EncodeDate` / `DecodeDate` (SysUtils) bleiben das richtige Werkzeug, wenn Sie ein Datum aus separaten Integer-Komponenten aufbauen oder in solche zerlegen — `DateUtils` ergänzt sie, statt sie zu ersetzen.

Die Faustregel: Greifen Sie zu `DateUtils`, wenn Sie eine *Kalender-Intention* ausdrücken (\"nächster Monat\", \"derselbe Tag\", \"Tage dazwischen\"), und zu den klassischen Funktionen, wenn Sie *Komponenten zusammensetzen* oder einen gemessenen Hot Path optimieren.

> **Alternativen quer durch den Stack**
> Jedes Ökosystem hat sich auf reichhaltige Datumsbibliotheken zubewegt, weil alle dieselbe Lektion gelernt haben. Wenn Ihre Arbeit mehrere Stacks umfasst: .NET hat `DateTime`/`DateTimeOffset` und die neueren Date-only-Typen; die JavaScript-/TypeScript-Welt setzt auf `date-fns` oder Luxon; Python hat das Standard-`datetime` plus `dateutil`. `DateUtils` ist Delphis Antwort im selben Geist — und sie ist eingebaut, nativ, ohne Abhängigkeit. Die Konzepte (Einheiten addieren, Einheiten differenzieren, Grenzen) lassen sich fast eins zu eins übertragen.

Fazit

Teil 1 wollte erreichen, dass sich alltäglicher Datumscode wie die Absicht dahinter liest. Was Sie in Teil 2 mitnehmen sollten:

- Ein `TDateTime` ist **ein einziges** `Double` — Ganzzahlteil = Tage, Bruchteil = Uhrzeit. Das erklärt sowohl, warum die Arithmetik funktioniert, als auch, warum selbstgebaute Monats-/Jahresrechnung fehleranfällig ist.
- `IncDay`/`IncHour`/`IncYear` (und `IncMonth` aus SysUtils) addieren oder subtrahieren kalenderkorrekt; negative Werte gehen rückwärts.
- `DaysBetween`/`MonthsBetween`/`YearsBetween` geben ganze verstrichene Einheiten zurück (Alter in einer Zeile!) — aber sie zählen *vollendete* Einheiten, also Vorsicht an den Grenzen.
- `IsSameDay`/`CompareDate` vergleichen auf der Granularität, die Sie *meinen*; `StartOfTheMonth`/`EndOfTheMonth` (und Tag/Woche/Jahr) klammern Zeiträume ein, ohne dass Sie Tage zählen.

> `System.DateUtils` verwandelt fummelige `TDateTime`-Arithmetik in lesbare, kalenderkorrekte Einzeiler — sagen Sie die Absicht (\"nächster Monat\", \"Tage dazwischen\", \"Ende der Woche\"), und überlassen Sie der RTL die Randfälle.

Alles bisher setzte eines stillschweigend voraus: dass alle Ihre Daten in *derselben* Zeitzone leben. In dem Moment, in dem das nicht mehr gilt — ein Server in UTC, Benutzer rund um den Globus, ein Zeitstempel aus einer API —, taucht eine neue Klasse von Bugs auf. Teil 2 dreht sich ganz um den korrekten Umgang damit: `TTimeZone`, Konvertieren von und nach UTC und die Austauschformate (ISO 8601, Unix-Zeit), die Daten systemübergreifend eindeutig halten. Dort wird Datumsbehandlung wirklich wichtig. Wir sehen uns dort.

> **Die Serie: Moderne Datumsbehandlung in Delphi (DateUtils)**
> Zwei Teile: 1. Alltägliche Datums- und Zeitoperationen **— Sie sind hier** 2. Zeitzonen, UTC und ISO 8601

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —