

# Eine ganze Datei in einer Zeile lesen: Das IOUtils-Kochbuch (1/2)

By Dr. Holger Flick

**IOUtils**

**# READ A WHOLE FILE IN ONE LINE: THE IOUtils COOKBOOK (1/2)**

**PART 1**  
DESIGN (WHY RECORDS?) & FILES

Files in one line.  
TFile.ReadAllText

Directories in one call.  
TDirectory.GetFiles

Paths done right.  
TPath.Combine

Records, not classes.  
No Create. No Free.  
No try/finally.

**SYSTUTILS (THE WAY WE KNOW)**

```
if FileExists(FileName) then
begin
  var SL := TStringList.Create;
  try
    SL.LoadFromFile(FileName);
    // ...
  finally
    SL.Free;
  end;
end;
```

**IOUtils (THE MODERN WAY)**

```
if TFile.Exists(FileName) then
begin
  var Contents :=
    TFile.ReadAllText(FileName);
  // ...
end;
```

Delphi 2010 and still going strong.

**R** RECORDS

**{ }** CLASS ... STATIC

**🚀** SIMPLE POWERFUL

Everything you need. Nothing you don't.

**IOUtils COOKBOOK**  
RECIPES FOR REAL-WORLD DELPHI

TFile TDirectory TPath

CROSS-PLATFORM. ZERO HASSLE.  
Windows • macOS • Linux • iOS • Android

Hier ist eine Zeile Delphi, die fast jeder von uns schon einmal getippt hat:

```
if FileExists(FileName) then
// ...
```

Sie funktioniert. Sie funktioniert seit dem letzten Jahrhundert. Es ist nichts falsch daran. Aber irgendwann — still und leise, ohne eine Fanfare, die die meisten von uns erreicht hätte — bekam die RTL einen *zweiten* Weg, diese Frage zu stellen:

```
if TFile.Exists(FileName) then
// ...
```

Gleiche Antwort. Andere Unit. Und diese zweite Unit — `System.IOUtils` — ist eine der leisesten, aber nützlichsten Ecken der modernen Delphi-RTL, die sehr viele praktizierende Delphi-Entwickler schlicht nie geöffnet haben. Es gibt sie seit **Delphi 2010**. Sie liest eine ganze Textdatei in *einer Zeile*. Sie listet ein Verzeichnis mit *einem Aufruf* auf. Sie baut Pfade, die sich unter Windows, macOS, Linux, iOS und Android korrekt verhalten, ohne dass Sie einen einzigen Backslash anfassen. Und — ein Detail, das ich in dieser Serie ganz bewusst

betonen möchte — die Dinge, über die Sie sie aufrufen, sind **Records, keine Klassen**. Es gibt kein `Create`, kein `Free`, kein `try/finally`.

Dies ist ein zweiteiliges Kochbuch nach dem Motto „zeig mir den Code“. Jedes Rezept stellt den klassischen `SysUtils`-Weg, den Sie bereits kennen, direkt neben den `IOUtils`-Weg — damit Sie Aufgabe für Aufgabe entscheiden können, welcher besser passt. Teil 1 behandelt das Design (warum Records?) und **Dateien**: lesen, schreiben, kopieren, verschieben, löschen und inspizieren. [Teil 2](#) nimmt sich Verzeichnisse und Pfade vor. Am Ende dieses Teils werden Sie zu `TFile` greifen, ohne darüber nachzudenken.

## Zuerst das, was Ihnen niemand gesagt hat: Das sind Records

Öffnen Sie `System.IOUtils` und schauen Sie sich die drei Namen an, auf die es ankommt. So deklariert die RTL sie tatsächlich — kopiert aus dem Interface-Abschnitt von `System.IOUtils.pas` (© Embarcadero, ausgeliefert mit RAD Studio):

```
TDirectory = record
    // ...
end;

TPath = record
    // ...
end;

TFile = record
    // ...
end;
```

Nicht `class`, `record`. Dieses eine Schlüsselwort ändert alles daran, wie Sie sie verwenden — es lohnt sich also, kurz innezuhalten.

Ein [Delphi-Record](#) (Records) ist ein Wertetyp. Anders als eine Klasse wird er nie auf dem Heap alloziert, und Sie geben ihn nie frei. Und seit [Delphi 2009 Records die Fähigkeit gab, Methoden zu besitzen](#), kann ein Record eine ganze Werkzeugkiste von Funktionen mit sich tragen. `IOUtils` denkt diese Idee konsequent zu Ende: Jede einzelne Methode von `TFile`, `TDirectory` und `TPath` ist als `class ... static` deklariert — sie gehört also zum *Typ*, nicht zu irgendeiner Instanz. Hier ein repräsentativer Ausschnitt, wieder direkt aus dem Quelltext:

```
TFile = record
public
    class function Exists(const Path: string; FollowLink: Boolean = True): Boolean; inline; static;
    class function ReadAllText(const Path: string): string; overload; inline; static;
    class procedure WriteAllText(const Path, Contents: string); overload; static;
    class procedure Copy(const SourceFileName, DestFileName: string); overload; inline; static;
    // ... Dutzende weitere, alle class ... static
end;
```

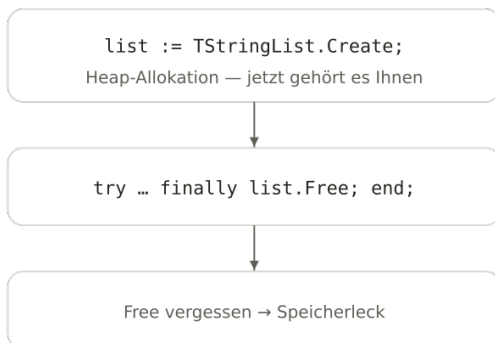
Die praktische Konsequenz: **Sie erzeugen niemals ein `TFile`**. Sie rufen Methoden *auf dem Typnamen selbst* auf, genau wie in einem Namespace freier Funktionen.

```
// Das tun Sie NICHT – es gibt nichts zu instanziiieren:
// var F := TFile.Create; // das ist etwas völlig anderes (siehe die Warnung unten)
// F.Exists(...);

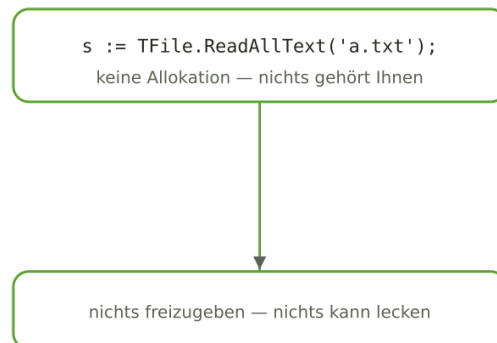
// Sie rufen einfach über den Typ auf:
if TFile.Exists('data.json') then
  Writeln('gefunden');
```

Lassen Sie mich den Kontrast zeichnen, der es klick machen lässt — denn genau darin liegt der Grund, warum dieses Design so angenehm zu benutzen ist.

#### Eine Klasse — ein Objekt mit Lebensdauer



#### Ein IOUtils-Record — ein Namespace aus Funktionen



Eine Klasse, die man instanziiert und freigeben muss; ein IOUtils-Record, den man einfach aufruft

Die linke Spalte ist der vertraute Objekt-Tanz: allozieren, mit `try/finally` absichern, freigeben. Die rechte Spalte ist das, was `IOUtils` Ihnen für die häufigen Fälle bietet — ein Aufruf, der einen Wert zurückgibt und nichts hinterlässt, das Sie aufräumen müssten. Das ist der ergonomische Gewinn des Record-Designs, und deshalb liest sich der Rest dieses Kochbuchs so sauber.

#### Eine wirklich verwirrende Ausnahme: `TFile.Create`

`TFile` hat tatsächlich eine Methode namens `Create` — aber sie erzeugt **kein** `TFile`. Sie ist eine Factory, die einen brandneuen, leeren `TFileStream` auf der Festplatte zurückgibt (`class function Create(const Path: string): TFileStream; static;`). Dieser zurückgegebene Stream **ist** ein Objekt, das Ihnen gehört und das Sie per `Free` freigeben müssen. Die Regel gilt also weiterhin — Sie instanziiieren den Record nie — aber lassen Sie sich vom Namen nicht täuschen: `TFile.Create('new.bin')` erzeugt eine Datei, keinen Datei-Helfer.

## Die ehrliche Version der Schlagzeile

Ich habe mit `FileExists` vs. `TFile.Exists` eröffnet, und ich schulde Ihnen die Wahrheit darüber, was unter der Haube wirklich passiert — denn die flix-Hausregel lautet: Wir übertreiben nicht. Wenn Sie die Implementierung von `TFile.Exists` im RTL-Quelltext lesen, ist dies der gesamte Rumpf:

```
class function TFile.Exists(const Path: string; FollowLink: Boolean = True): Boolean;
begin
  Result := FileExists(Path, FollowLink);
end;
```

Sie ruft `FileExists` auf. `TFile.Exists` ist keine klügere, schnellere oder mächtigere Existenzprüfung — sie ist eine dünne, einheitliche Fassade über derselben RTL-Primitive. Wenn Sie also nur prüfen, ob eine Datei existiert, ist `FileExists` völlig in Ordnung, und es gibt keinen Grund, sich rückständig zu fühlen, weil Sie es benutzen.

Der Wert von `IOUtils` liegt nicht darin, dass irgendeine einzelne Methode ihren `SysUtils`-Verwandten schlägt. Er liegt in der **Kohärenz**: eine Unit, eine Namenskonvention, ein mentales Modell, das Dateien, Verzeichnisse und Pfade auf jeder Plattform abdeckt, die Delphi unterstützt — bewusst nach dem Vorbild von .NETs `System.IO` gestaltet, damit sich die Formen vertraut anfühlen. Das ist der eigentliche Grund, sie zu lernen. Jetzt lösen wir diesen Anspruch ein — Rezept für Rezept.

## Rezept: eine ganze Textdatei lesen

Das ist das Rezept, das Leute überzeugt. Eine Textdatei auf klassische Art zu lesen bedeutet: einen Container wählen, ihn erzeugen, hineinladen und wieder freigeben:

```
var
  Lines: TStringList;
  Text: string;
begin
  Lines := TStringList.Create;
  try
    Lines.LoadFromFile('config.ini');
    Text := Lines.Text;
  finally
    Lines.Free;
  end;
end;
```

Der `IOUtils`-Weg ist ein einziger Ausdruck, der den Inhalt der Datei als String zurückgibt — kein Container, kein `try/finally`:

```
var
  Text: string;
begin
  Text := TFile.ReadAllText('config.ini');
end;
```

`ReadAllText` hat eine Überladung, die ein `TEncoding` entgegennimmt, falls Sie eines erzwingen müssen; ohne Encoding-Argument erkennt sie eine Byte-Order-Mark und fällt sonst auf den Standard zurück. Zwei Geschwistermethoden runden die Lese-Familie ab:

- `TFile.ReadAllLines('log.txt')` liefert ein `TArray<string>` — ein Element pro Zeile, ideal für `for ... in`.
- `TFile.ReadAllBytes('image.png')` liefert ein `TBytes` — die ganze Datei als rohe Bytes, perfekt für binäre Inhalte.

### Wann Sie **\*nicht\*** alles auf einmal lesen sollten

`ReadAllText/ReadAllBytes` laden die gesamte Datei in den Speicher. Das ist genau das, was Sie für eine Konfigurationsdatei oder ein kleines Dokument wollen — und genau das, was Sie für ein mehrere Gigabyte großes Log *nicht* wollen. Für große oder streamende Lesevorgänge liefert `TFile.OpenText` einen

`TStreamReader` und `TFile.OpenRead` einen `TFileStream` — beides Objekte, die Ihnen gehören und die Sie freigeben, sodass Sie die Datei stück- oder zeilenweise verarbeiten können.

## Rezept: eine Textdatei schreiben, anhängen und erzeugen

Schreiben spiegelt Lesen. Der klassische Ansatz leiht sich wieder eine `TStringList`:

```
var
  Lines: TStringList;
begin
  Lines := TStringList.Create;
  try
    Lines.Text := 'Hallo, Welt!';
    Lines.SaveToFile('greeting.txt');
  finally
    Lines.Free;
  end;
end;
```

Mit `IOUtils` ist es ein Aufruf, und die Datei wird für Sie erzeugt (oder überschrieben):

```
TFile.WriteAllText('greeting.txt', 'Hello, world!');
```

Die gesamte Schreib-/Anhänge-Familie ist symmetrisch zur Lese-Familie — und genau das ist der Punkt: Kennen Sie eine, kennen Sie alle:

```
TFile.WriteAllText('out.txt', SomeString);           // erzeugen/überschreiben aus einem String
TFile.WriteAllLines('out.txt', ArrayOfStrings);      // erzeugen/überschreiben aus TArray<string>
TFile.WriteAllBytes('out.bin', SomeBytes);           // erzeugen/überschreiben aus TBytes
TFile.AppendAllText('log.txt', 'another line'#13#10); // ans Ende anhängen, erzeugt bei Bedarf
```

Wenn Sie Text lieber selbst hinausstreamen möchten: `TFile.CreateText` liefert einen frischen `TStreamWriter` (ein Objekt, das Sie freigeben), und `TFile.AppendText` liefert einen, der am Ende einer bestehenden Datei positioniert ist.

## Rezept: eine Datei kopieren, verschieben und löschen

Hier sind die `IOUtils`-Namen schlicht klarer als die `SysUtils`-Pendants — und Klarheit ist etwas wert, wenn ein Kollege Ihren Code in sechs Monaten liest. Vergleichen Sie:

| Aufgabe                  | SysUtils                                                                    | IOUtils                             |
|--------------------------|-----------------------------------------------------------------------------|-------------------------------------|
| Datei kopieren           | <code>CopyFile(PChar(Src), PChar(Dst), False)</code> ( <i>Windows-API</i> ) | <code>TFile.Copy(Src, Dst)</code>   |
| Verschieben / umbenennen | <code>RenameFile(Src, Dst)</code>                                           | <code>TFile.Move(Src, Dst)</code>   |
| Löschen                  | <code>DeleteFile(FileName)</code>                                           | <code>TFile.Delete(FileName)</code> |

Die `IOUtils`-Varianten lesen sich wie der Satz, den Sie laut aussprechen würden. `TFile.Copy` hat eine Überladung mit einem `Overwrite: Boolean`-Flag; standardmäßig weigert sie sich, ein bestehendes Ziel zu überschreiben — ein sichererer Standard als das rohe `Windows-CopyFile`. Beachten Sie allerdings einen bewussten Unterschied im Temperament.

### Sie schlagen laut fehl — und meistens ist das genau richtig

`SysUtils.DeleteFile` gibt ein `Boolean` zurück, das Sie stillschweigend ignorieren können; `TFile.Delete` löst eine Exception aus, wenn es die Aufgabe nicht erledigen kann. Das ist kein Fehler, sondern eine

Philosophie: `IOUtils` folgt dem .NET-Modell des *Werfens bei Fehlern* statt der Rückgabe eines Statuscodes — ein Problem tritt also sofort zutage, statt sich hinter einem ungeprüften Ergebnis zu verstecken. Umgeben Sie Dateioperationen mit `try/except`, wo Fehlschläge zu erwarten sind (eine gesperrte Datei, ein fehlender Pfad), und lassen Sie die Exception propagieren, wo das nicht der Fall ist.

## Rezept: eine Datei inspizieren — Größe, Zeitstempel, Attribute

Hier erspart Ihnen `IOUtils` tatsächlich frickeligen Plattform-Code. Die Größe einer Datei auf die alte Art zu ermitteln heißt, entweder einen Stream zu öffnen oder Windows-APIs aufzurufen; `TFile.GetSize` liefert direkt ein `Int64`:

```
var
  Bytes: Int64;
begin
  Bytes := TFile.GetSize('bigfile.dat');  // -1, wenn die Datei nicht gelesen werden kann
end;
```

Zeitstempel kommen als passendes Set von Gettern und Settern, jeweils in einer Lokalzeit- und einer UTC-Variante — kein `FileAge`, kein `FindFirst`-Record zum Auseinandernehmen:

```
Created  := TFile.GetCreationTime('report.pdf');
Written  := TFile.GetLastWriteTime('report.pdf');
Accessed := TFile.GetLastAccessTime('report.pdf');
// UTC-Geschwister: GetCreationTimeUtc, GetLastWriteTimeUtc, GetLastAccessTimeUtc
// und die passenden SetXxx- / SetXxxUtc-Prozeduren
```

Attribute kommen als `TFileAttributes`-Set zurück, was deutlich angenehmer ist als das Bit-Gefummel an einem Integer voller Flags:

```
if TFileAttribute.faReadOnly in TFile.GetAttributes('locked.txt') then
  Writeln('schreibgeschützt');
```

Hier gibt es eine plattformübergreifende Feinheit, die man kennen sollte — und sie ist ein gutes Beispiel dafür, wie das Design der ganzen Unit durchscheint.

### **`TFileAttribute` ist absichtlich plattformspezifisch**

Das `TFileAttribute`-Enum ist **nicht auf jedem OS dieselbe Menge** — die RTL deklariert zwei verschiedene Versionen. Unter Windows bekommen Sie die vertrauten `faReadOnly`, `faHidden`, `faSystem`, `faArchive`, ...; unter POSIX (macOS, Linux, Mobile) stattdessen Unix-artige Einträge wie `faOwnerRead`, `faGroupWrite`, `faOthersExecute`, `faSymLink`, .... Der Typ ist im Quelltext sogar als `platform` markiert, sodass der Compiler einen Hinweis gibt, wenn Sie ihn in plattformübergreifendem Code verwenden. Das ist ehrliches Engineering: Die RTL tut nicht so, als wären Windows-ACLs und Unix-Berechtigungsbits dasselbe. Wenn Ihr Code überall kompilieren muss, kapseln Sie die Attributlogik hinter `{IFDEF MSWINDOWS}` / `{IFDEF POSIX}` und verwenden Sie die Einträge, die es auf der jeweiligen Seite gibt.

## Rezept: eine Datei als Stream öffnen, auf die moderne Art

Manchmal brauchen Sie doch einen echten Stream — um ihn einem Parser, einem JSON-Reader oder einer Netzwerkkomponente zu übergeben. `TFile` bietet dafür saubere Factories, und dies ist die eine Stelle, an der die Objektregeln zurückkehren — seien wir also explizit, was die Besitzverhältnisse angeht.

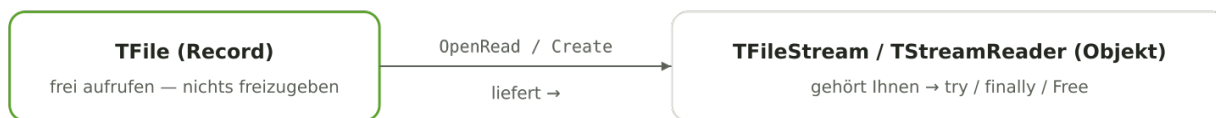


```

var
  Stream: TFileStream;
begin
  Stream := TFile.OpenRead('data.bin');  // liefert ein Objekt, das IHNEN gehört
  try
    // ... aus Stream lesen ...
  finally
    Stream.Free;                        // ab jetzt Ihre Verantwortung
  end;
end;

```

`TFile.OpenRead`, `TFile.OpenWrite` und das flexible `TFile.Open(Path, Mode, Access, Share)` liefern alle einen `TFileStream`, den Sie freigeben müssen — denn ein Stream hat, anders als der Record, tatsächlich eine Lebensdauer. Das mentale Modell bleibt konsistent mit allem in diesem Blog: *Records sind werttypisierte Helfer, die Sie nie freigeben; die Objekte, die sie zurückgeben, gehören Ihnen*. Wenn Sie die vertiefte Behandlung von Objekt- und Stream-Besitz möchten, gehen [mein Interfaces-Beitrag](#) und Dalija Prasnikars [Delphi Memory Management](#) beide weiter, als ich es hier kann.



Die Faustregel: Den Record rufen Sie frei auf; die Objekte, die er zurückgibt, geben Sie frei

Das Bild ist die gesamte Besitzgeschichte auf einen Blick: Alles links vom Pfeil ist „aufrufen und vergessen“; alles rechts davon müssen Sie selbst aufräumen.

## Ein faires Wort zu den Alternativen

Weil dies ein Delphi-zentrierter Beitrag ist, verlangen die Hausregeln, dass ich die weitere Landschaft ehrlich benenne. Wenn Sie über Ökosysteme hinweg arbeiten, wird sich die Form von `IOUtils` aus gutem Grund vertraut anfühlen: Sie wurde nach dem Vorbild von .NETs `System.IO` gestaltet, dessen Klassen `File`, `Directory` und `Path` sie fast Methode für Methode spiegelt. Node.js-Entwickler greifen zu `node:fs` (`fs.readFileSync`, `fs.promises`), Python-Entwickler zum eleganten objektorientierten `pathlib` und zum altherwürdigen Gespann `os/shutil`. Jedes davon ist in seinem eigenen Zuhause exzellent. Was `IOUtils` dem Delphi-Entwickler gibt, ist dieselbe Ein-Zeilen-Bequemlichkeit *nativ kompiliert*, in der Sprache, die Sie ohnehin ausliefern — keine

Runtime, kein Interpreter, dasselbe Binary auf fünf Plattformen. Greifen Sie zu dem Werkzeug, das dort lebt, wo Ihr Code lebt; und wenn Ihr Code Delphi ist, dann ist dies das Werkzeug, das die ganze Zeit in Ihrer RTL versteckt war.

## Wo Teil 1 Sie zurücklässt

Wir wollten zeigen, dass modernes Delphi für die Dateiarbeit ein saubereres, plattformübergreifendes Vokabular hat, als viele von uns je gelernt haben — und ehrlich sein, wie weit dieser Anspruch tatsächlich reicht. Das hier hält der Prüfung stand:

-

`System.IOUtils` gibt Ihnen `TFile`, `TDirectory` und `TPath` — **Records, keine Klassen**. Sie rufen Methoden auf dem Typnamen auf; Sie erzeugen (`Create`) und befreien (`Free`) sie nie. (Die eine Falle: `TFile.Create` gibt einen *Stream* zurück, der Ihnen sehr wohl gehört.)

- Für Dateien verwandelt `TFile` mehrzeilige Zeremonien in einzelne Ausdrücke: `ReadAllText`, `WriteAllText`, `Copy`, `Move`, `Delete`, `GetSize`, `GetLastWriteTime`, `GetAttributes`.
- Die Bequemlichkeit ist real, aber die Magie liegt in der *Kohärenz*, nicht in roher Kraft — `TFile.Exists` ruft buchstäblich `FileExists` auf. Nutzen Sie `IOUtils` für eine einheitliche, plattformübergreifende API, nicht weil die alten Funktionen kaputt wären.
- Wenn eine Methode einen `TFileStream` oder `TStreamReader` zurückgibt, ist dieses Objekt Ihres — und Sie geben es frei. Den Record nicht.

Modernes Delphi hat `FileExists` nicht ersetzt. Es hat `FileExists` eine ganze Familie gegeben — eine kohärente, plattformübergreifende Record-API für alles, was Sie mit dem Dateisystem tun.

Teil 2 verfolgt denselben rezeptgetriebenen Ansatz für **Verzeichnisse** (`TDirectory` — anlegen, auflisten, durchlaufen, Bäume kopieren) und **Pfade** (`TPath` — zusammensetzen, zerlegen, und die wirklich unverzichtbaren plattformübergreifenden Ordner-Speicherorte wie `GetDocumentsPath`) und schließt mit einer vollständigen, einseitigen Referenz zu jedem Record und jeder Methode der Unit. Wenn Sie Dateien schreiben, werden Sie auch die dazugehörigen Verzeichnis- und Pfad-Werkzeuge haben wollen. Wir sehen uns dort.

#### Die Serie: das IOUtils-Kochbuch

Zwei Teile, Rezept für Rezept: 1. [Dateien: lesen, schreiben, kopieren, inspizieren](#) — Sie sind hier 2.

[Verzeichnisse, Pfade und die vollständige Referenz](#) — Ordner auflisten, Pfade bauen und jede Methode an einem Ort



# 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 —](#)