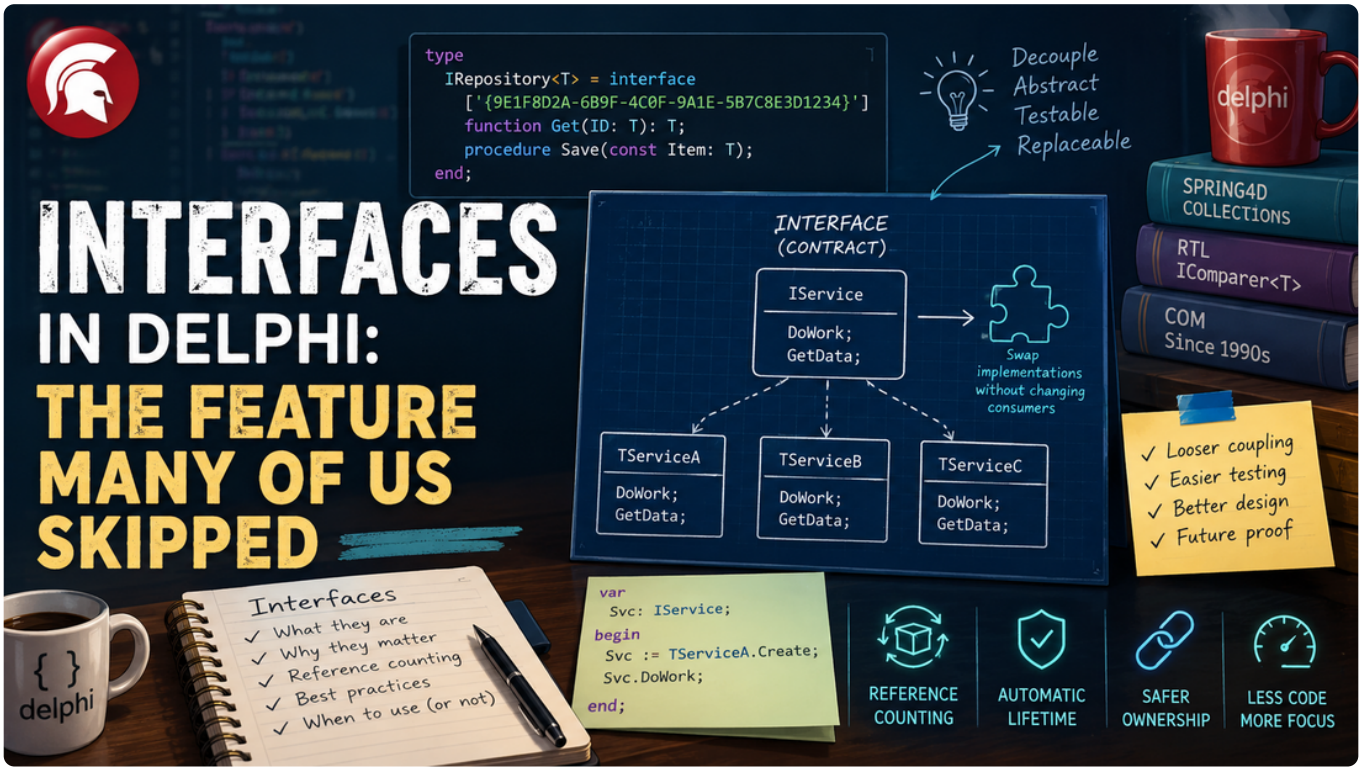
NEWS · JULY 9, 2026

Interfaces in Delphi: The Feature Many of Us Skipped

By Dr. Holger Flick



Here's something I've learned from years of working with Delphi teams: you can write excellent, shipping, money-making software for twenty years and never once declare an interface. Many of my customers have done exactly that. Classes, forms, `try/finally`, a well-organized data module — that toolkit carries you remarkably far, and there's nothing wrong with it.

But there's a curious gap. The libraries Delphi developers admire most lean on interfaces *heavily*. In my recent tour of Spring4D's collections, I verified something striking in the source: every single one of its 127 factory methods returns an interface, not a class. The RTL's own `IComparer<T>` is an interface. All of COM — the machinery behind decades of Windows automation — is interfaces. So the feature many of us nodded past is quietly load-bearing across the ecosystem.

I promised in that Spring4D post that interfaces would get a proper introduction of their own. This is it: what an interface actually is, why it's worth your time in 2026, how reference counting makes memory management *easier* rather than scarier — and, just as honestly, the sharp edges and the situations where a plain class is still the better call.

We'll start from what you already know and build up slowly, with runnable-sized examples at every step.

What you already do: classes, objects, and try/finally

Let's start on familiar ground, because interfaces make the most sense as a *contrast* to it.

In Delphi, an object created from a class is yours to destroy. The compiler doesn't watch its lifetime; you do. The idiom every Delphi developer has typed a thousand times:

```
var
  list: TStringList;
begin
  list := TStringList.Create;
  try
    list.Add('Hello');
    // ... work with list ...
  finally
    list.Free;
  end;
end;
```

This is *manual memory management*, and — said with genuine respect — it's a perfectly good model. It's explicit, it's predictable, there's no garbage collector pausing your app at odd moments, and ownership is always answerable: whoever created it (or whoever it was handed to) frees it. Delphi developers have built rock-solid software on this model for three decades.

Since Delphi 10.4 Sydney unified memory management across all platforms, this story is also beautifully consistent: object instances are managed manually *everywhere* — Windows, macOS, Linux, iOS, Android. No more separate ARC compiler on mobile with different rules. One language, one model.

> **The book to keep next to your keyboard**
> If you want the definitive treatment of how object and interface lifetimes really work in Delphi — `Free` vs. `Destroy` vs. `FreeAndNil`, exceptions during destruction, ownership patterns, reference cycles — Dalija Prasnikar's book Delphi Memory Management for Classic and ARC Compilers is, in my opinion, the single best reference the community has. Much of what this post introduces gently, her book covers exhaustively.

But notice one thing about the snippet above: the `try/finally` is *ceremony*. It says nothing about your problem — it exists purely so a forgotten `Free` doesn't become a leak. Hold that thought.

An interface is a contract — nothing more, nothing scarier

Here's the whole idea in one sentence: **a class says* how *something is done; an interface says only* what *can be done.**

An interface is a type that declares methods without implementing any of them. It's a promise — a contract — that some class, somewhere, will fulfill:

```
type
  ILogger = interface
    ['{8B3E4C21-9D5A-4F6E-B2C7-1A0D9E8F3B42}']
    procedure Log(const msg: string);
  end;
```

That's a complete, legal interface declaration (Object Interfaces in the Delphi docs)). No fields, no implementation, no constructor. Just: *anything claiming to be an* `ILogger` *can* `Log`*.*

> **What's that GUID for?**
> The string in square brackets is a GUID) — a globally unique identifier. Delphi uses it to identify interfaces at runtime, which is what makes `Supports()` and `QueryInterface` work (and it's required for COM). In the IDE, press **Ctrl+Shift+G** and Delphi generates one for you. You'll rarely think about it again — just always include one.

A class fulfills the contract by listing the interface and implementing its methods. The easiest way to start is to inherit from `TInterfacedObject`, which handles the plumbing for you:

```
type
  TFileLogger = class(TInterfacedObject, ILogger)
  private
    FFileName: string;
  public
    constructor Create(const fileName: string);
    procedure Log(const msg: string);
  end;

procedure TFileLogger.Log(const msg: string);
begin
  TFile.AppendAllText(FFileName, msg + sLineBreak);
end;
```

And using it looks like this — note what's *missing*:

```
var
  logger: ILogger;                      // the variable is the INTERFACE type
begin
  logger := TFileLogger.Create('app.log');
  logger.Log('Application started');
  // no try/finally, no logger.Free — see the next section
end;
```

The variable's type is `ILogger`, not `TFileLogger`. The calling code knows the *contract* and nothing else. That one shift unlocks both of the big advantages, so let's take them one at a time.

Advantage 1: swap the "how" without touching the callers

Because callers depend only on the contract, you can change the implementation behind it freely — and this is where interfaces start paying rent in real projects.

Picture three classes fulfilling the same contract:

```
type
  TConsoleLogger = class(TInterfacedObject, ILogger)
    procedure Log(const msg: string);    // writes to stdout
  end;

  TSilentLogger = class(TInterfacedObject, ILogger)
    procedure Log(const msg: string);    // does nothing — perfect for tests
  end;

  // ... and TFileLogger from above
```

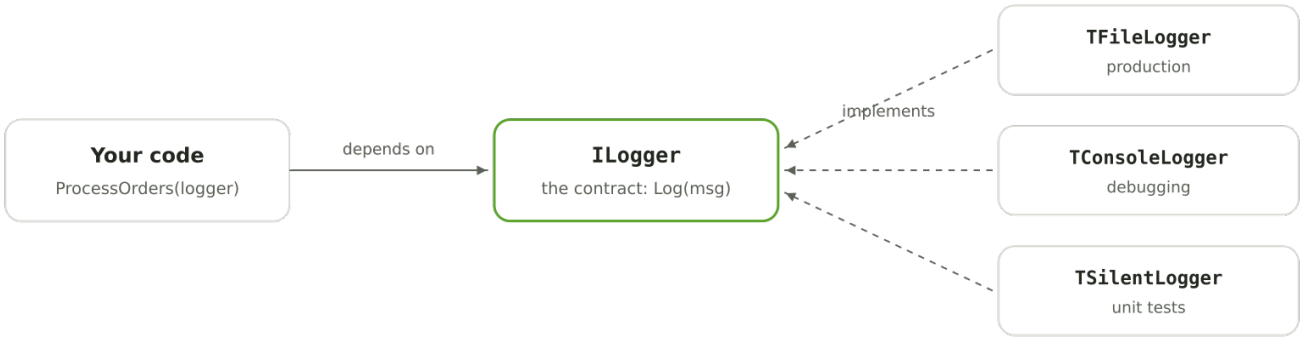
Any code written against `ILogger` works with all three, unchanged:

```
procedure ProcessOrders(const logger: ILogger);
begin
  logger.Log('Starting order run');
  // ... business logic ...
  logger.Log('Done');
end;

// production:
ProcessOrders(TFileLogger.Create('orders.log'));

// in a unit test — no file system touched:
ProcessOrders(TSilentLogger.Create);
```

Here's the relationship as a picture — the caller touches only the contract, never the concrete classes.



Callers depend on the contract; implementations can be swapped freely behind it

Take from the picture: the arrows from your code stop at the contract. The concrete classes on the right can be added, replaced, or rewritten without your calling code ever knowing — which is exactly what makes testing, logging swaps, and "we need a second backend" requests painless.

Notice this is **polymorphism without inheritance**. `TFileLogger` and `TSilentLogger` don't share an ancestor (beyond `TInterfacedObject`); they share a *promise*. You're no longer forced to design deep class hierarchies just to get substitutability — and a single class can implement several unrelated interfaces at once, something single inheritance can never give you.

> **This idea is bigger than Delphi**
> If you've touched other ecosystems, you've met this concept: C# and Java interfaces, TypeScript's `interface`, Go's implicit interfaces — all the same "contract vs. implementation" split. Learning it in Delphi is learning a portable design skill, not a vendor feature.

Advantage 2: the object frees itself

Here's the part that surprises people most, and it's the reason the earlier snippet had no `try/finally`: **interface references are counted, and when the count hits zero, the object destroys itself.**

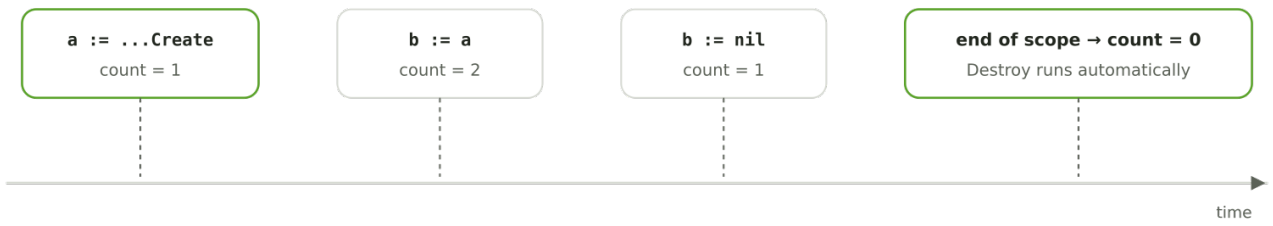
You already trust this mechanism every day, by the way — Delphi `string`s have been reference-counted forever, and that remained true on every platform after the 10.4 unification. Nobody writes `try/finally` around a string.

Interfaces extend that same courtesy to your own types.

The mechanics, step by step:

```
procedure Demo;
var
  a, b: ILogger;
begin
  a := TFileLogger.Create('app.log');   // reference count: 1
  b := a;                                // reference count: 2
  b := nil;                              // reference count: 1
end;                                     // 'a' goes out of scope ' count: 0
                                         // ' Destroy is called automatically
```

And as a timeline:



An interface-managed object lives exactly as long as someone still references it

Read it left to right: every assignment to an interface variable ticks the count up, every variable that lets go ticks it down, and the moment nobody holds a reference, the object cleans itself up — deterministically, right then, not "eventually" like a garbage collector. You keep Delphi's predictability and lose the ceremony.

For a fuller picture of how interface references) behave — including assignment-compatibility rules — the official docs are good, and Dalija's book (linked above) is the deep end.

> **The 2026 story is refreshingly simple**
> After the 10.4 unification, the rules are the same on every platform Delphi targets: **objects are freed manually; interfaces and strings free themselves by reference count.** Two lifetime models, cleanly

> separated, no per-platform exceptions. If you learned Delphi when mobile had its own ARC rules, it's simpler now than when you left off.

Where you've already met interfaces (maybe without noticing)

Interfaces aren't an exotic add-on to Delphi; they're woven through the ecosystem you already use, and it's worth seeing how mainstream they are.

- **The RTL itself.** Every time you pass a custom comparer to a sort, you're using `IComparer<T>` from `System.Generics.Defaults` — an interface.
- **COM and Windows automation.** Driving Excel or Word from Delphi, shell extensions, OLE — the entire COM model is built on interfaces. It's the historical reason Delphi's interface support is so mature.
- **Spring4D.** As covered in the collections tour, its entire public collection API is interfaces — `IList<T>`, `IDictionary<K,V>`, `IEnumerable<T>` — created via factories and freed by reference counting. Once the concepts in *this* post feel comfortable, that library reads naturally.

That's the practical motivation for learning this material: it's not a niche technique, it's the idiom the best tooling in the ecosystem already speaks.

The sharp edges — three rules that keep you safe

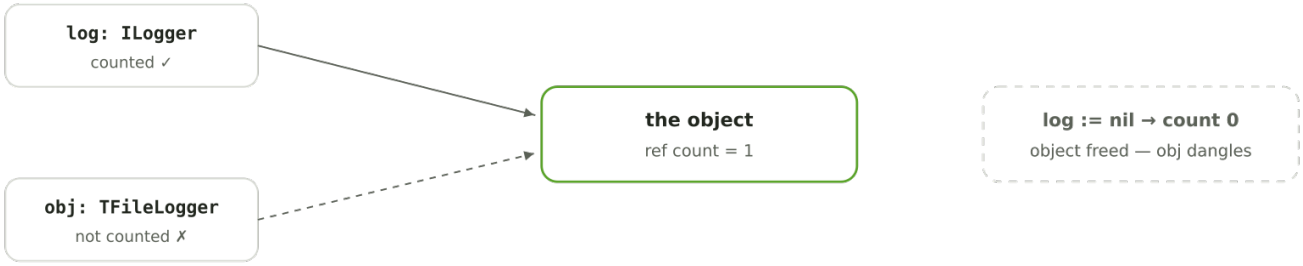
I'd be doing you a disservice to pretend interfaces have no gotchas. There are exactly three big ones, and all three are avoidable with simple habits. Learn these before you refactor anything real.

Rule 1: never hold the same object through both a class and an interface reference

This is the classic beginner accident, so let's look at it squarely:

```
var
  obj: TFileLogger;      // class reference — does NOT count
  log: ILogger;          // interface reference — counts
begin
  obj := TFileLogger.Create('app.log');
  log := obj;            // count: 1
  log := nil;            // count: 0 ' object is DESTROYED here
  obj.Log('crash');      // obj now points at freed memory!
end;
```

The object reference `obj` is invisible to the reference count. When the last *interface* reference lets go, the object dies — regardless of any class references still pointing at it.



A class reference doesn't count — when the counted references reach zero, it dangles

The takeaway from the picture: only the solid arrow keeps the object alive. The dashed one is just a raw pointer with no say in the matter.

> **One habit prevents this entirely**
> Once a class participates in reference counting (typically by descending from `TInterfacedObject`), **hold it through interface variables only, from the moment of creation**. Assign `TFileLogger.Create(...)` straight into an `ILogger` and never keep a `TFileLogger` variable around. One type per object, and this whole class of bug can't happen.

Rule 2: reference cycles need a `[weak]` link

If object A holds an interface reference to B, and B holds one back to A, their counts can never reach zero — both wait for the other forever, and you have a leak. The fix is to make one direction of the cycle *uncounted*.

Since [Delphi 10.1 Berlin, the classic compiler supports `[weak]` and `[unsafe]` attributes on interface references](https://blog.marcocantu.com/blog/2016-april-weak-unsafe-interface-references.html) exactly for this:

```
type
  TChild = class(TInterfacedObject, IChild)
  private
    [weak] FParent: IParent;   // doesn't count — and is set to nil
  end;                          // automatically if the parent dies first
```

A `[weak]` reference doesn't increase the count, and Delphi tracks it: if the target is destroyed, the weak reference becomes `nil` instead of dangling. (`[unsafe]` also skips counting but without the tracking — it's for rare, expert cases.) The rule of thumb: **the "owner" direction is strong, the "back-pointer" direction is `[weak]`.**

Parent'child strong, child'parent weak.

Rule 3: components play by their own (older) rules

`TComponent` descendants — forms, data modules, and everything you drop on them — implement interfaces *without* reference-counted lifetimes. Their lifetime belongs to the ownership model: the Owner frees them, exactly as it always has. That's a deliberate and sensible design — your form isn't going to vanish because an interface variable went out of scope. Just know that "interfaces = automatic lifetime" applies to `TInterfacedObject`-style classes, not to the VCL component world.

When *not* to use interfaces

Now the honest scoping, because a good tool recommendation always says where it stops. My take, from real projects; treat it as a starting point.

- **Components and anything with an Owner.** As per rule 3, the ownership model already manages those lifetimes elegantly. Interfaces add nothing there — don't fight the framework.
- **Working code with clear, local ownership.** A `TStringList` created and freed in the same method with `try/finally` is *fine*. There is zero need to retrofit interfaces onto code whose ownership story is already obvious. Refactor toward interfaces where you feel the pain (testing, swapping, unclear lifetimes) — not on principle.
- **Tiny data carriers.** For small value-like data, a `record` is often better than either a class or an interface — no heap allocation, no lifetime question at all.
- **The hottest of hot paths.** Reference counting does real (small) work — the count updates are thread-safe atomic operations, and interface method calls are always virtual-style dispatched. For the vast majority of code you will never notice. If you're in a genuinely performance-critical inner loop, measure before committing to either design — the profiler outranks any blog post, including this one.
- **A team that isn't on board yet.** Consistency within a codebase is worth more than any single technique. Introduce interfaces at natural seams — a new service, a testable boundary — rather than as a big-bang rewrite.

> **The fit-based rule of thumb (my take)**
> **Reach for an interface when code needs a* seam*: you want to swap implementations (tests, backends, platforms), you want callers decoupled from a concrete class, or you want shared, automatic lifetime for something passed around widely. Stay with a plain class when ownership is local and obvious**, when you're in `TComponent` territory, or when the object never crosses a boundary worth abstracting. Both are first-class Delphi. The skill is knowing which question you're answering.

Takeaways

Interfaces are not an advanced curiosity bolted onto Delphi — they're one half of how the language thinks about objects, and in 2026 the story is cleaner than it has ever been.

- **An interface is a contract**: it declares *what*, a class provides *how*. Callers that depend on contracts are easier to test, extend, and change.
- **Reference counting is deterministic convenience**: interface-held objects free themselves the instant the last reference lets go — the same mechanism you've trusted in strings all along, uniform across all platforms since Delphi 10.4.
- **Three habits keep you safe**: one reference type per object, `[weak]` on back-pointers, and remember components follow the ownership model instead.
- **Scope it honestly**: local `try/finally` code, components, records, and measured hot paths are all places where *not* using interfaces is the right engineering call.
-

The ecosystem rewards you: from `IComparer<T>` to COM to Spring4D's entirely interface-based collections, this is the idiom the best Delphi code already speaks. For the memory-management deep dive, keep Dalija Prasnikar's book within reach.

> A class says how. An interface says what. Learn to separate the two, and both your architecture and your memory management get simpler — one contract at a time.

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 —