NEWS · JULY 7, 2026

Collections in Delphi and Spring4D

By Dr. Holger Flick

Open almost any serious Delphi project and you'll find the same two friends everywhere: `TList<T>` and `TDictionary<T, V>`. They're good. They've earned their place. For a huge amount of everyday work, they are genuinely all you need.

But every so often you hit a shape of problem the runtime library doesn't have a container for — you want a key that maps to *many* values, or a dictionary you can look up from *either* side, or a queue that quietly drops its oldest entry when it fills, or you just want to write `people.Where(...).OrderBy(...)` the way you would in C# or LINQ and move on. That's the moment a lot of us reach for Spring4D — an open-source (Apache-2.0) base-class library by Stefan Glienke that has been quietly leveled-up since 2009, and whose `Spring.Collections.*` units are, in my honest opinion, some of the most enjoyable code to work with in the entire Delphi ecosystem.

I want to give you the tour I wish I'd had years ago: the breadth of what's in that toolbox, with real code — and then, just as honestly, where the *modern* Delphi RTL has caught up so well that you may not need the dependency at all. Both of those things are true at once, and that's the fun of it.

Everything below is grounded in the actual Spring4D source (the `Spring.Collections` unit and its siblings, copyright 2009–2024). Where I state a number or a claim, I link it. Where I'm giving you my opinion, I'll say so.

First, the thing that makes it click: everything is an interface

Before any specific collection, there's one design decision that colors the whole experience, so let me show it before I show anything else.

In Spring4D you don't create a *class* you have to `Free`. You ask a factory for an **interface**, and Delphi's reference counting cleans it up for you when it leaves scope.

```
uses
  Spring.Collections;

procedure Demo;
var
  names: IList<string>;      // an interface, not a class
begin
  names := TCollections.CreateList<string>;
  names.Add('Ada');
  names.Add('Grace');
  // no try/finally, no names.Free — the interface is ref-counted
end;
```

That `TCollections` factory (declared in `Spring.Collections.pas`) is the front door to the whole library:

`CreateList`, `CreateDictionary`, `CreateSet`, `CreateStack`, `CreateQueue`, `CreateDeque`, `CreateMultiMap`, and many more, each with overloads for comparers, initial values, and ownership.

Here's the mental model, because it's the difference that makes the rest feel effortless.

RTL collections are classes you own and free; Spring4D hands you ref-counted interfaces

Neither approach is "right" — they're different philosophies, and both have devoted fans. The RTL's explicit ownership is transparent and has zero reference-counting overhead. Spring4D's interface model removes a whole category of `try/finally` boilerplate and the leaks that come from forgetting it. I happen to love the second one; your codebase's existing conventions matter more than my preference here.

Now, a frank observation from years of working with Delphi teams: **many long-time Delphi developers have simply never made interfaces part of their daily toolkit.** That's not a criticism — Delphi's object model is perfectly productive without them, and a lot of excellent, shipping software was written by developers who reach for `TObject` descendants and `try/finally` by instinct and never needed more. But it does mean that Spring4D's interface-first design can feel unfamiliar at first, precisely because it leans on a language feature many of us skipped past. If that's you, the payoff of getting comfortable with interfaces reaches far beyond this one library — it's one of the highest-leverage skills in modern Delphi.

Coming soon: interfaces, properly

Interfaces deserve more than a paragraph, so I'm planning a dedicated post on them — what they are, why reference counting makes memory management easier rather than harder, and how a handful of patterns (like the one Spring4D uses here) can clean up code you've maintained for years. If interfaces have always felt like the part of Delphi you nodded past, that post is being written for you. Watch this space.

One nice touch: ownership of the *elements*, too

For object collections, Spring4D can free the items as well, not just the container. `TCollections.CreateObjectList<T>(ownsObjects: True)` gives you an `IList<T>` that frees each object when it's removed or when the list itself dies — mirroring the RTL's `TObjectList<T>`, but wrapped in the same ref-counted interface. Dictionaries take a `TDictionaryOwnerships` set (`doOwnsKeys`, `doOwnsValues`) so you can own keys, values, both, or neither.

LINQ in Delphi, for real

The feature that first sold me, and still delights me, is that `IEnumerable<T>` carries a full set of query operators — so you can express "what I want" instead of hand-rolling a loop.

Start with the everyday case: filter a list, sort it, and pull out one field. Reading top to bottom, it says exactly what it does.

```
uses
  Spring.Collections;

var
  people: IList<TPerson>;
  names: IEnumerable<string>;
begin
  // ... fill people ...

  // adults, sorted by name, projected down to just the name
  names :=
    TEnumerable.Select<TPerson, string>(
      TEnumerable.OrderBy<TPerson, string>(
        people.Where(function(const p: TPerson): Boolean
          begin Result := p.Age >= 18 end),
        function(const p: TPerson): string
          begin Result := p.Name end),
      function(const p: TPerson): string
        begin Result := p.Name end);
end;
```

Once that reads comfortably, the more powerful operators follow the same shape. Here's aggregation — grouping people by city so you can, say, count how many live in each:

```
var
  people: IList<TPerson>;
  byCity: IEnumerable<IGrouping<string, TPerson>>;
  group: IGrouping<string, TPerson>;
begin
  // ... fill people ...

  byCity :=
    TEnumerable.GroupBy<TPerson, string>(people,
      function(const p: TPerson): string
        begin Result := p.City end);

  for group in byCity do
    WriteLn(group.Key, ': ', group.Count);   // e.g. "Berlin: 12"
end;
```

The operators split across two homes, and it's worth knowing why, because it's a Delphi language fact rather than a Spring4D quirk. Methods that don't introduce a new type live directly on `IEnumerable<T>` — `Where`, `Take`, `Skip`, `Distinct`, `Concat`, `Union`, `First`, `Any`, `All`, `Min`, `Max`, `Sum`, `Aggregate`, `ToArray`, and more. Methods that *do* introduce a new result/key type (`Select<T, TResult>`, `GroupBy`, `OrderBy`, `Zip`, `SelectMany`)

can't be generic interface methods in Delphi, so they live as static methods on `TEnumerable`. That's exactly why `Where` reads as a fluent `people.Where(...)` above, while `OrderBy`, `Select`, and `GroupBy` are called as `TEnumerable.OrderBy(...)`. Once you know that split, the API stops surprising you.

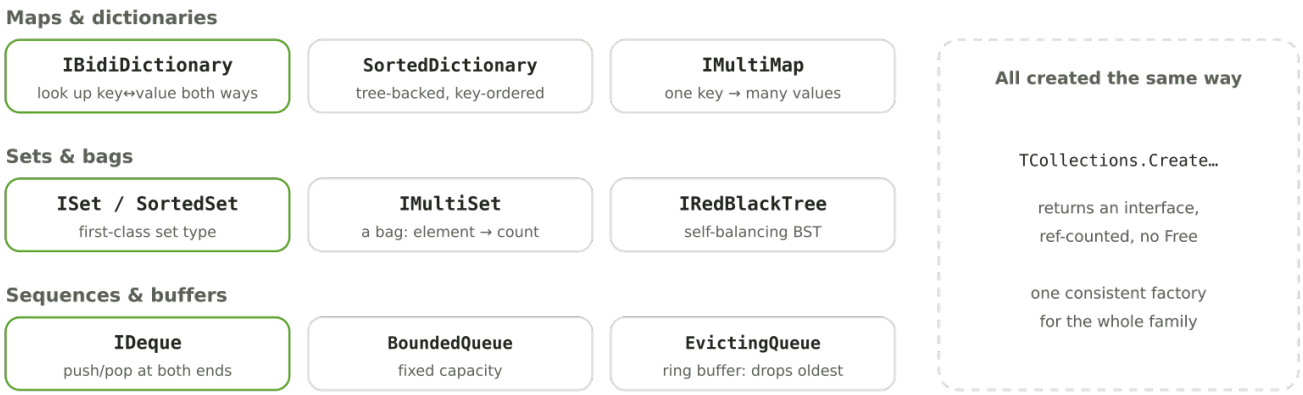
> **A small honesty note on the surface**
>
> Spring's `IEnumerable<T>` has `ToArray`, but there is no `ToList` — the materializers are `ToArray`, `TEnumerable.ToDictionary`, and `TEnumerable.ToLookup`. Its "except" operator is named `Exclude`, and ordering is `OrderBy`/`OrderByDescending`/`Ordered` (I didn't find a `ThenBy` in the source). None of this is a shortcoming — it's just the actual shape, and I'd rather you learn it from a factual tour than be surprised in the IDE.

> **Alternatives, in the spirit of a fair tour**
>
> Spring4D isn't the only way to get query-style helpers in Delphi — System.Generics.Collections.TEnumerable offers a handful of built-in ones. And if you ever find yourself wanting *heavy* set-based querying, that's often a signal the work belongs closer to your database's SQL engine. Spring4D's strength is that it's broad, fast, and all in one well-tested place.

The collections the RTL simply doesn't ship

Here's where the toolbox metaphor really lands. Beyond `IList` and `IDictionary`, Spring4D includes container shapes that have no direct equivalent in `System.Generics.Collections`. This isn't a knock on the RTL — it's a deliberately lean standard library — it's just genuinely more variety.

Let me lay the family out, because seeing them together is half the value.



Spring.Collections ships container shapes the RTL doesn't have out of the box

A few of these are worth a sentence each, because the *use case* is what makes them click:

- `IMultiMap` — a key that maps to a collection of values (`CreateMultiMap`, `CreateHashMultiMap`, `CreateTreeMultiMap`, and sorted variants). Perfect for "all orders for this customer" without you managing lists-of-lists by hand.
- `IBidiDictionary` — a bidirectional dictionary you can query from key *or* value. Handy for stable id ” name mappings where you need both directions.
- `IMultiSet` — a bag/counter that tracks how many times each element appears. A word-frequency counter becomes a two-liner.
- `IDeque` **and the bounded/evicting buffers** — double-ended queues and ring buffers built on a shared circular buffer. An `EvictingQueue` that keeps only the last N log lines is exactly the kind of thing you'd otherwise reinvent and get subtly wrong.
- `IRedBlackTree` **and sorted sets/dictionaries** — genuine self-balancing trees when you need ordered iteration with good insert/lookup behavior.

Here's the counter in practice — the kind of thing that's a real pleasure to *not* have to write by hand:

```
var
  wordCount: IMultiSet<string>;
  word: string;
begin
  wordCount := TCollections.CreateMultiSet<string>;
  for word in Tokenize(document) do
    wordCount.Add(word);
  // wordCount['the'] is now the number of times 'the' appeared
end;
```

Lifetime, ordering, and events you get for free

A few more conveniences round out why the toolbox feels cohesive rather than like a pile of parts, so let me group them.

First, **insertion order is preserved by default**: `TCollections.CreateDictionary<TKey, TValue>` actually returns an `IOrderedDictionary`, so iterating gives you back the order you inserted — something the RTL's classic `TDictionary` never promised.

Second, **change notifications are built in**. Collections expose an `OnChanged` event (dictionaries expose `OnKeyChanged` and `OnValueChanged`), reporting actions like `caAdded`, `caRemoved`, and `caReplaced`. There are even observable list factories (`CreateObservableList`) for UI-style binding scenarios.

```
var
  items: IList<string>;
begin
  items := TCollections.CreateList<string>;
  items.OnChanged.Add(
    procedure(Sender: TObject; const Item: string;
      Action: TCollectionChangedAction)
    begin
      if Action = caAdded then
        Log('added: ' + Item);
    end);
  items.Add('hello');   // fires the handler
end;
```

That's the whole pitch for the toolbox: broad, consistent, ref-counted, observable, and fast (more on speed next). Now let me be equally enthusiastic about the other side of the story.

And where the modern RTL is all you need

Here's the part I most want to be fair about, because it's a genuinely good-news story for every Delphi developer: **the standard library has come a long way, and staying current pays off in ways that aren't just visible components.**

We tend to judge a new Delphi release by what we can drop on a form. But a lot of the most valuable work over the last several releases has been *under the hood* — in the compiler and the RTL. The generic collections have been refined repeatedly: Delphi 10.3 Rio added `TryAdd` and measurable speedups to core operations like `Add` and `IndexOf`, and made `for-in` loops notably faster. More recently, Delphi 12 introduced `TOrderedDictionary<K,V>`, bringing insertion-ordered iteration right into `System.Generics.Collections` — a capability that used to be a reason to reach for a library. And Delphi 13 Florence (released September 2025) is the current, modern baseline this whole discussion sits on.

The honest summary is that the gap has been *closing*, steadily, release after release.

> **The upgrade angle, stated plainly**
> If you're on an older Delphi and weighing an upgrade, the RTL and compiler improvements are a real part of the value — not just the new visual bits. A modern `TDictionary` and a built-in `TOrderedDictionary` are the kind of quiet wins that make everyday code faster and simpler. Keeping current is one of the best "features" you can give your codebase.

So for a large share of everyday code — a `TList<T>` of records, a `TDictionary<Integer, TThing>` for lookups, an ordinary `for-in` — **the RTL is not a compromise. It's the right tool, with zero extra dependencies.** If that's the shape of your problem, you can absolutely skip the third-party library and lose nothing that matters.

Performance: what's actually known — fairly, from both sides

Performance deserves care, because it's the easiest place to overclaim, so let me stick to what's documented and label the rest.

The strongest published data point on the Spring4D side comes from its author, Stefan Glienke, describing the Spring4D 2.0 collections rewrite: the 1.x collections were, in his own candid words, slow — "dead slow, especially on Win32" — because nearly every method was virtual. Version 2.0 removed the virtual dispatch behind the interfaces and reworked the internals, and he reports **roughly a 4–7× improvement over the RTL for dictionary lookups** in his benchmark. That's a striking result, and it's the kind of engineering that earns a library its reputation.

Two pieces of honesty belong right next to that number, though.

> **Where a benchmark stops generalizing**
> A 4–7× dictionary-lookup result is a *specific* micro-benchmark (a particular key type, match rate, and Spring4D 2.0 vs. the RTL of that time). It's real, and it's impressive — but it is not a blanket "Spring4D is 5× faster at everything," and it shouldn't be read as one. Different operations, key types, and Delphi versions will land differently. **The only number that governs your decision is the one you measure on your own workload.** Both libraries are fast enough that for most code you will never feel the difference; when you're in a genuine hot path, profile it.

The fair, both-sides reading is this: Spring4D's collections are seriously well-optimized *and* the RTL has kept improving. Those two facts don't fight each other. If raw throughput on a specific structure is your bottleneck, Spring4D is very much worth benchmarking — and so is simply being on the latest Delphi.

So — which do you reach for?

Let me bring the tour back to a practical decision, because "it depends" is only useful if I tell you what it depends *on*. This next part is my opinion, formed from years of using both; treat it as a starting point, not a rule.

> **A fit-based rule of thumb (my take)**
> **Reach for Spring.Collections when the* shape *of your problem matches what it uniquely offers: you want LINQ-style querying inline; you need a multimap, bidirectional dictionary, multiset/bag, deque, or evicting ring buffer; you want ref-counted collections without* `try/finally`*; or you want change notifications. In those cases it saves you real code and real bugs. Reach for the RTL when your needs are well within its wheelhouse: ordinary lists and dictionaries, ordered iteration (you now have* `TOrderedDictionary`*), and you'd rather not add a dependency to the project. On a modern Delphi, that covers a* lot *of code, and it's a perfectly good answer. The deciding question isn't "which is better" — both are excellent. It's "does my problem need what only one of them has?"** If yes, that picks itself. If no, the one already in your project wins on simplicity.

There's no wrong choice here, and — importantly — no need to be dogmatic. Plenty of healthy codebases use the RTL by default and pull in Spring4D exactly where a multimap or an evicting buffer makes the code obviously better. Mixing them is fine.

Takeaways

Spring.Collections is a genuinely rich, well-engineered toolbox, and I still reach for it happily — but the modern Delphi RTL has closed enough of the gap that the right answer is finally, cleanly, *about fit*.

- **The RTL is excellent for the common case**, and it keeps getting better — `TryAdd`, faster generics, and Delphi 12's `TOrderedDictionary` are real wins. Staying current is one of the best upgrades you can make.
- **Spring4D shines where the RTL doesn't reach:** inline LINQ-style queries, multimaps, bidirectional and sorted dictionaries, multisets, deques, evicting ring buffers, ref-counted lifetime, and change events — all behind one consistent `TCollections` factory.
- **On performance, respect both:** Spring4D 2.0's rewrite is impressively fast (its author reports 4–7× on dictionary lookups), and the RTL is no slouch — but only *your* benchmark on *your* workload should decide a hot path.
- **And if the interface-first style is new to you, that's an opportunity, not a wall** — I'll have a dedicated post on interfaces soon, because they pay off far beyond this one library.

> Two great tools, one honest question: does your problem need what only one of them offers? If it does, that picks the tool. If it doesn't, keep it simple — and either way, Delphi comes out ahead.

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 —