NEWS · AUGUST 3, 2026

What Is RTTI? Delphi's Reflection, Part 1

By Dr. Holger Flick

Here's a question that sounds impossible at first: **can a program inspect its own code while it's running?**

Can it look at an object it's never seen before, ask "what properties do you have, what are they called, what types are they," and then read and write them — all without a single line written specifically for that object's class?

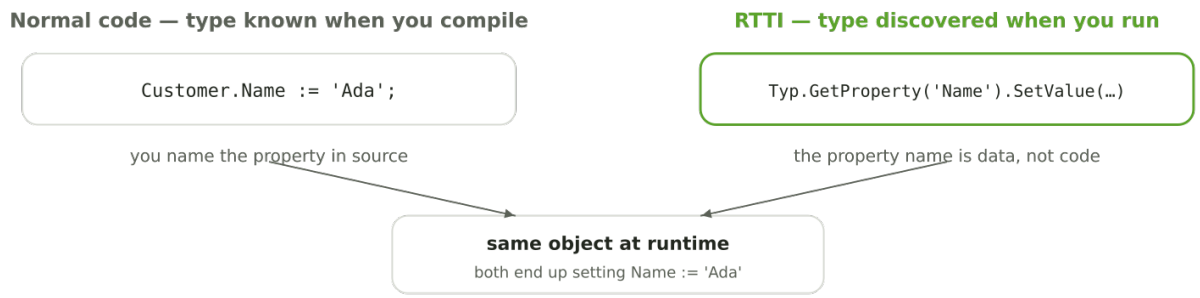
The answer in modern Delphi is yes, and the machinery that makes it possible is **RTTI — Run-Time Type Information**. It's the quiet engine behind an enormous amount of the tooling you already use: the way the IDE's Object Inspector lists a component's published properties, the way FireDAC and ORMs map object fields to database columns, the way a JSON serializer turns any object into text and back. None of those tools were written knowing about *your* classes — they discover them at runtime, through RTTI.

Most Delphi developers have benefited from RTTI for years without ever calling it directly. That's a bit of a shame, because the modern `System.Rtti` unit — introduced back in Delphi 2010 — makes reflection genuinely approachable, and once it clicks it unlocks a whole category of elegant, generic solutions. This three-part series is a gentle, honest tour. **Part 1 answers the "what" and "why"**: what RTTI actually is, the two records everything starts from (`TRttiContext` and `TValue`), and — importantly — when *not* to reach for it. Part 2 reads types, properties, and attributes; Part 3 invokes methods and builds a small, genuinely useful tool. Let's start with the idea.

Reflection, in one sentence

The umbrella concept has a name you'll see across many languages: **reflection** — a program's ability to examine and manipulate its own structure at runtime. Java has it, C# has it, Python has it. In Delphi, the feature is called RTTI, and `System.Rtti` is the modern, object-oriented API for using it.

Normally your code is *specific*: to read a `TCustomer`'s `Name`, you write `Customer.Name`, and the compiler bakes in exactly where that field lives. RTTI flips this around. It lets you write code that works with a type it knows *nothing* about until the program runs — asking the type to describe itself, then acting on the answer.



Normal code knows the type at compile time; RTTI discovers it at run time

The diagram captures the essential difference. On the left, the property name `Name` is *compiled in* — change your mind and you edit source and recompile. On the right, `'Name'` is a **string** — data your program can receive from a config file, a JSON key, or a database column, and act on without ever having heard of `TCustomer`. That indirection is the entire superpower.

Why would I ever want this?

The honest question. Reflection has a cost (we'll be candid about it below), so it should earn its place. The tasks where it earns it overwhelmingly share one shape: **you're writing code that must work uniformly across many types you don't control.** A few concrete examples, each a real thing people build with `System.Rtti`:

- **Serialization** — turning *any* object into JSON/XML and back, by walking its properties. The RTL's own `REST.Json` does exactly this.
- **Object–relational mapping** — reading which fields a class has to build `INSERT`/`SELECT` statements automatically.
- **Dependency injection & IoC containers** — inspecting a class's constructor to figure out what to supply.
- **Validation & UI generation** — reading properties (and *attributes* on them) to auto-build a form or check a rule.
- **Copying / comparing objects generically** — a "clone any object" routine that doesn't care what the object is.

The common thread: without RTTI, each of these forces you to hand-write per-class boilerplate — a `ToJSON` method on every single class, forever. With RTTI, you write the walker *once*.

The starting point: `TRttiContext`

Everything in `System.Rtti` begins with a single record: `TRttiContext`. Think of it as your session with the reflection system — you open one, ask it questions, and close it. It's declared in the source as a record with a `Create` and a `Free`:

```
uses
  System.Rtti;

var
  Ctx: TRttiContext;
  Typ: TRttiType;
begin
  Ctx := TRttiContext.Create;
  try
    Typ := Ctx.GetType(TButton);      // reflect the TButton class
    ShowMessage(Typ.QualifiedName);    // 'Vcl.StdCtrls.TButton'
  finally
    Ctx.Free;
  end;
end;
```

A few things are worth calling out right away, because the lifetime rules here are a little unusual and worth getting right from the first example.

> **TRttiContext is a record, but you still Create/Free it — here's why**
>
> `TRttiContext` is a **record**, so the pattern from our IOUtils and regex posts (records you never free) doesn't apply here. The reason is that the context holds an internal reference that keeps a *pool of `TRtti` objects alive and cached**. Create/Free manage that pool. You *can* omit them — the context uses a reference-counted token internally, so it's not a hard leak — but the idiomatic, predictable form is Create in a try, Free in the finally. The objects it hands you (TRttiType, TRttiProperty`, …) are **owned by the context** — you never free those yourself.

The context's most-used method is `GetType`, which takes either a class reference or a type-info pointer and returns a `TRttiType` — the runtime description of that type. That `TRttiType` is the gateway to everything in Part 2: its properties, methods, fields, and attributes. `GetType` has a partner, `GetTypes`, that returns *every* type the program has RTTI for, and `FindType` looks one up by its qualified-name string.

The universal value: `TValue`

There's a second record you must meet before anything else makes sense, because it's everywhere in the API: `TValue`. Here's the problem it solves. If you read a property whose type you don't know at compile time, what does the read *return*? An `Integer`? A `string`? A `TObject`? You can't know until runtime — so RTTI needs a container that can hold **a value of any type**, while remembering what that type actually is. That container is `TValue`.

TValue is a typed box: it holds a value of any type and remembers which type

The picture is the concept: whatever you put in — an integer, a string, an object reference — comes out as a `TValue` that *knows its own type*, and you extract it with a typed accessor. You put values in with `TValue.From<T>` (or a simple assignment, thanks to implicit operators the source declares for common types), and you take them out with `AsInteger`, `AsString`, `AsObject`, the generic `AsType<T>`, or the safe `ToString`:

```
var
  V: TValue;
begin
  V := TValue.From<Integer>(42);   // box an Integer
  // ... or just:  V := 42;        // implicit operator does the same

  if V.IsEmpty then Exit;
  ShowMessage(V.AsString);          // '42'  (ToString-style conversion)
  ShowMessage(V.AsInteger.ToString);// '42'  (typed extraction)
end;
```

`TValue` is what lets a single serialization routine handle a property whether it's a number, a name, or a nested object — it's the "any type" glue that makes generic code possible.

> **TValue is also useful on its own**
> You don't need the whole reflection apparatus to appreciate `TValue`. It's a handy, type-safe alternative to `Variant` whenever you need a variable that can hold different types — and unlike `Variant`, it can hold *any* type including objects and records, and it remembers the exact type rather than coercing. If you've ever reached for `Variant` to pass "some value of some type" around, `TValue` is often the cleaner modern choice.

The other side: when *not* to use RTTI

Guarantee to the reader — reflection is powerful, and precisely because of that it's easy to over-apply. Let me scope it honestly.

RTTI trades a little runtime cost and some compile-time safety for enormous flexibility. That trade is a *bargain* when you're writing a generic framework component (a serializer, a mapper) used across many types. It's a *bad deal* when you're reaching for it to do something you could do directly.

> **Where reflection is the wrong tool**
> - **You know the type at compile time.** If you can write `Customer.Name`, write `Customer.Name` — it's faster, safer, and clearer than `GetProperty('Name')`. RTTI is for when you *can't* name the member in source. - **Hot**

> **inner loops.** Reflective access involves lookups and boxing into `TValue`; it's fine for wiring things up, but don't reflect inside a tight per-pixel or per-row loop if a direct call would do. Reflect once, cache the result. - **Binary size / linking.** RTTI generation adds metadata to your executable, and the compiler can only strip what it's sure is unused. For most desktop and server apps this is a non-issue; on tightly size-constrained targets it's worth knowing the `{$RTTI}` directive exists to tune what's emitted (a Part 2 topic). None of this means "avoid RTTI." It means *use it where its flexibility pays for itself* — framework-level, cross-type code — and use direct member access everywhere else.

> **Alternatives, briefly**
> Reflection is a cross-ecosystem idea, so if your work spans stacks: .NET's `System.Reflection` and Java's `java.lang.reflect` solve the same problems with very similar shapes (a "type" object exposing properties/methods/attributes). Delphi's `System.Rtti` will feel immediately familiar if you've used either. The concepts transfer; only the names change.

What Part 1 established

We set out to answer *what RTTI is and why you'd want it*, honestly scoped. The foundation to carry into Part 2:

- **RTTI (Run-Time Type Information)** is Delphi's **reflection**: code that inspects and manipulates types it didn't know at compile time. It powers serializers, ORMs, DI containers, and the Object Inspector itself.
- Everything starts with the `TRttiContext` record — `Create` it, ask it for a `TRttiType` via `GetType`, `Free` it when done. The `TRttiType` objects it returns are **owned by the context**.
- `TValue` is the universal typed container that lets generic code hold and pass a value of *any* type while remembering what that type is.
- Reflect where flexibility pays — cross-type framework code. Use direct member access when you already know the type. Cache reflected lookups; don't reflect in hot loops.

> RTTI lets your program treat *types themselves* as data — discovering an object's properties and methods at runtime and acting on them, without a line of code written for that specific class.

With the "what" and "why" in hand, Part 2 gets concrete: opening a `TRttiType`, walking its properties and fields, reading and writing values through them, and reading the **custom attributes** that make modern Delphi frameworks so declarative. That's where reflection starts to feel like magic. See you there.

> **The series: Delphi's Reflection (RTTI)**
> Three parts: 1. What is RTTI? **— you are here** 2. Reading types, properties, and attributes 3. Invoking methods — and a real tool

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 —