NEWS · AUGUST 4, 2026

RTTI Part 2: Reading Types, Properties, and Attributes

By Dr. Holger Flick

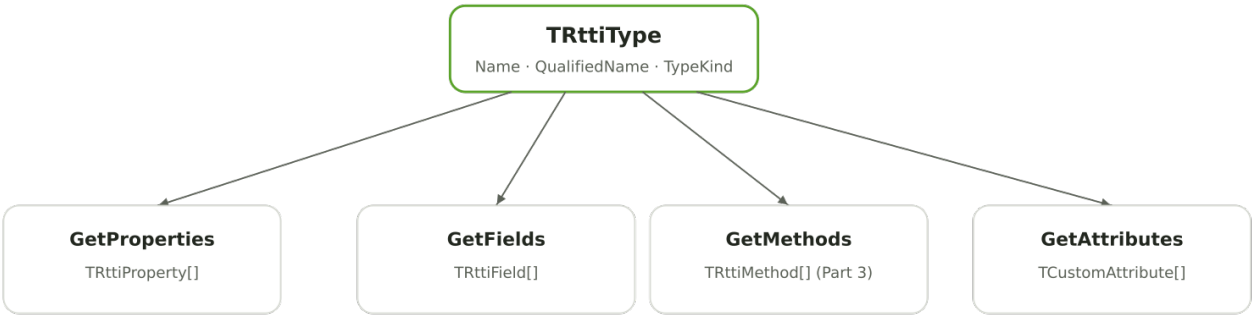
In Part 1 we established the idea: **RTTI is reflection** — code that inspects and manipulates types it didn't know at compile time — and it all starts from two records, `TRttiContext` (your session with the reflection system) and `TValue` (a box that holds a value of any type while remembering what that type is).

Now we make the type *talk*. This is the part where reflection stops being an abstract idea and becomes something you can watch work: you hand `System.Rtti` an object, and it tells you every property the object has, what each is named, what type it is — and then lets you read and write those properties by name, as a string. Do that once and the appeal is obvious; it's how a single routine can serialize, validate, or copy *any* class.

Part 2 covers reading structure: opening a `TRttiType`, walking its **properties** and **fields**, getting and setting their values through `TValue`, and then the feature that makes modern Delphi frameworks so clean — **custom attributes**, the little `[Bracketed]` annotations you can attach to a class and read back at runtime. Part 3 will use all of this to *invoke methods* and build a small, real tool. Let's read some types.

The map: what a `TRttiType` exposes

When you call `Ctx.GetType(...)` you get a `TRttiType`, and it's the hub for everything. Before the code, here's the shape of what it offers, so the method names that follow feel like a map rather than a list.



A TRttiType is the hub: from it you reach properties, fields, methods, and attributes

The diagram is your Part 2 table of contents: from one `TRttiType` you reach `GetProperties`, `GetFields`, `GetMethods`, and `GetAttributes` (plus the singular `GetProperty`/`GetField`/`GetMethod` for looking one up by name). We'll take properties and fields now, attributes next, and save methods for Part 3.

Walking an object's properties

Let's make it concrete with a small class and a routine that describes *any* instance of *any* class — the "hello world" of reflection. Suppose we have:

```
type
  TCustomer = class
  private
    FName: string;
    FAge: Integer;
  public
    property Name: string read FName write FName;
    property Age: Integer read FAge write FAge;
  end;
```

Here's a routine that prints every property's name, type, and current value — and note that it mentions `TCustomer` *nowhere*. It takes a plain `TObject`:

```
uses
  System.Rtti;

procedure Describe(const Obj: TObject);
var
  Ctx: TRttiContext;
  Typ: TRttiType;
  Prop: TRttiProperty;
begin
  Ctx := TRttiContext.Create;
  try
    Typ := Ctx.GetType(Obj.ClassType);
    for Prop in Typ.GetProperties do
      Writeln(Format('%s: %s = %s',
        [Prop.Name,
         Prop.PropertyType.Name,
         Prop.GetValue(Obj).ToString]));
  finally
    Ctx.Free;
  end;
end;
```

Call `Describe(SomeCustomer)` and you get:

```
Name: string = Ada Lovelace
Age: Integer = 36
```

Look at what happened. `GetProperties` returned a `TArray<TRttiProperty>`; each `TRttiProperty` knows its own `Name`, its `PropertyType` (itself a `TRttiType`), and — the key move — `GetValue(Obj)` reads that property *out of the live object* and hands it back as a `TValue`, whose `ToString` we print. This one routine now works for `TCustomer`, `TInvoice`, `TButton`, or a class that doesn't exist yet.

Reading *and writing* by name

Reflection reads and writes. The mirror image of `GetValue` is `SetValue`, and together they let you set a property when the property's name arrives as *data* — from a JSON key, a config entry, a database column. Here's a tiny generic setter:

```
procedure SetProp(const Obj: TObject; const PropName: string; const Value: TValue);
var
  Ctx: TRttiContext;
  Prop: TRttiProperty;
begin
  Ctx := TRttiContext.Create;
  try
    Prop := Ctx.GetType(Obj.ClassType).GetProperty(PropName);
    if (Prop <> nil) and Prop.IsWritable then
      Prop.SetValue(Obj, Value);
  finally
    Ctx.Free;
  end;
end;

// usage — the property name is a runtime string:
SetProp(Customer, 'Name', 'Grace Hopper');
SetProp(Customer, 'Age', 45);
```

`GetProperty(PropName)` looks up a single property by its string name (returning `nil` if there's none), `IsWritable` guards against read-only properties, and `SetValue` pokes the new value into the object. That `Value: TValue` parameter is why Part 1's box mattered: it lets one routine accept an integer here and a string there.

GetValue reads a live property into a TValue; SetValue writes one back

The diagram is the whole read/write cycle: `GetValue` pulls the property's current contents into a `TValue`; `SetValue` pushes a `TValue` back in. Every generic serializer and object-mapper is, at heart, a loop around these two calls.

Fields vs. properties — a quick, important distinction

You'll also see `GetFields`, and it's worth a sentence on when to use which, because they're not the same thing.

Properties are the public-facing surface (`Name`, `Age`) — often backed by getters/setters, and what you almost always want for serialization because they represent the object's logical state. **Fields** are the raw storage (`FName`, `FAge`) behind them. `GetFields` works identically — `Field.FieldType`, `Field.GetValue(Obj)`, `Field.SetValue(Obj, V)` — but reaches the underlying data directly. Prefer properties unless you specifically need the storage (some serializers deliberately read fields to bypass side-effecting setters). The API is symmetric, so knowing one gives you the other.

> **Visibility: what RTTI can see**
> Members carry a visibility — the RTL enumerates it as `mvPrivate`, `mvProtected`, `mvPublic`, `mvPublished`. By default the compiler emits full RTTI for **public and published** members of classes, which is what most reflection needs. If you need private/protected members reflected, or want to *reduce* emitted metadata for size, the `{$RTTI}` compiler directive controls exactly what's generated — see the RTTI directive docs. For everyday serialization of public/published properties, the defaults just work.

Attributes: annotations you can read at runtime

Here's the feature that elevates RTTI from "useful" to "elegant." A **custom attribute** is a small piece of metadata you attach to a class, property, field, or method in square brackets — and can read back through RTTI at runtime. If you've used a modern Delphi ORM or serializer and written something like `[Column('customer_name')]` above a property, you've used attributes; they're how declarative frameworks are built.

An attribute is just a class descending from `TCustomAttribute` (declared in `System.pas` as the base of them all). You define one with a constructor, then apply it in brackets:

```
type
  // 1. Define an attribute — an ordinary class with a constructor
  ColumnAttribute = class(TCustomAttribute)
  private
    FName: string;
  public
    constructor Create(const AName: string);
    property Name: string read FName;
  end;

  // 2. Apply it to properties, carrying extra metadata
  TCustomer = class
  private
    FName: string;
    FAge: Integer;
  public
    [Column('customer_name')]
    property Name: string read FName write FName;
    [Column('customer_age')]
    property Age: Integer read FAge write FAge;
  end;
```

Now the payoff — reading those attributes back to, say, discover each property's database column name without hardcoding a single mapping:

```
var
  Ctx: TRttiContext;
  Prop: TRttiProperty;
  Attr: TCustomAttribute;
begin
  Ctx := TRttiContext.Create;
  try
    for Prop in Ctx.GetType(TCustomer).GetProperties do
      for Attr in Prop.GetAttributes do
        if Attr is ColumnAttribute then
          Writeln(Format('%s -> column "%s"',
            [Prop.Name, ColumnAttribute(Attr).Name]));
  finally
    Ctx.Free;
  end;
end;
```

Output:

```
Name -> column "customer_name"
Age -> column "customer_age"
```

That's an ORM's mapping layer in miniature. The class *declares* its database columns right next to the properties, and a generic routine *reads* those declarations to build SQL — no separate mapping file, no per-class code.

> **The typed shortcut: GetAttribute<T> and HasAttribute<T>**
> Looping and type-testing with `is` works, but the RTL gives you cleaner generic helpers on every reflected object: `HasAttribute<ColumnAttribute>` returns a Boolean, and `GetAttribute<ColumnAttribute>` returns the attribute already cast to its type (or `nil`). So the inner check becomes: ``pascal var Col := Prop.GetAttribute<ColumnAttribute>; if Col <> nil then  Writeln(Prop.Name + ' -> ' + Col.Name); `` Cleaner, and no manual casting. Prefer these when you're after a specific attribute type.

The other side: attributes are metadata, not magic

Guarantee to the reader — attributes are wonderful, but it's worth being clear-eyed about what they are.

An attribute does *nothing on its own*. `[Column('customer_name')]` doesn't create a column or change how the property behaves; it's inert data until some code deliberately reads it via `GetAttributes` and acts. That's a feature — it keeps the annotation and the behavior decoupled — but it means the value of an attribute is entirely in the framework that interprets it. Define attributes when you're *building* such a framework (or configuring one that reads them). Don't scatter attributes that nothing reads; they're documentation with a runtime cost, not behavior.

Takeaways

Part 2 turned a `TRttiType` from a name into a fully explorable structure. What to carry into Part 3:

- From a `TRttiType`, `GetProperties`/`GetFields` (and singular `GetProperty`/`GetField`) enumerate an object's members; each knows its `Name` and type.
- `GetValue(Obj)` reads a member out of a live object as a `TValue`; `SetValue(Obj, V)` writes one back. This read/write pair is the core of every generic serializer and mapper.
- Prefer **properties** for logical state; reach for **fields** only when you need raw storage. Default RTTI covers public/published members; `{$RTTI}` tunes the rest.
- **Custom attributes** (`TCustomAttribute` subclasses) attach declarative metadata you read with `GetAttributes` or the typed `GetAttribute<T>` — the foundation of ORMs, validators, and serializers. They're inert until code reads them.

> Reflection reads and writes: `GetValue`/`SetValue` let one routine handle any object's properties by name, and custom attributes let a class *declare* how a framework should treat it.

Part 3 adds the last verb — **invoking methods** by name with `TRttiMethod.Invoke` — and ties the whole series together by building a small, genuinely useful reflection-powered tool. See you for the finale.

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