NEWS · AUGUST 5, 2026

RTTI Part 3: Invoking Methods — and a Real Tool

By Dr. Holger Flick



We've come a long way. Part 1 explained what **RTTI** is — reflection, code that inspects types it didn't know at compile time — and introduced `TRttiContext` and `TValue`. Part 2 read an object's **properties, fields, and attributes**, and wrote values back by name with `GetValue`/`SetValue`. There's one verb left, and it's the most dramatic: **calling a method by its name, as a string.**

If reading a property by name felt like a small magic trick, invoking a method by name is the whole show. It means you can call `DoThing` on an object when the string `'DoThing'` only arrives at runtime — from a script, a config file, a message from a server, a plugin. It's how command dispatchers, scripting bridges, and test runners work.

Part 3 covers invocation — `TRttiMethod.Invoke`, passing arguments and reading return values as `TValue` — and then, to tie the whole series together, we'll build one small, genuinely useful tool with everything we've learned: a **generic object-to-CSV exporter** that works on any list of any class, using nothing but reflection.

Let's finish strong.

Calling a method by name

The setup mirrors Part 2 exactly: open a context, get the type, but this time ask for a **method** and `Invoke` it.

Suppose we have a class with a method that does something and returns a result:

```
type
  TCalculator = class
  public
    function Add(A, B: Integer): Integer;
    procedure Reset;
  end;
```

Here's how you call `Add(2, 3)` when `'Add'` is a runtime string — note that the arguments go in as an **array of** `TValue`, and the result comes back *as* a `TValue`:

```
uses
  System.Rtti;

var
  Ctx: TRttiContext;
  Calc: TCalculator;
  Method: TRttiMethod;
  Res: TValue;
begin
  Calc := TCalculator.Create;
  Ctx := TRttiContext.Create;
  try
    Method := Ctx.GetType(TCalculator).GetMethod('Add');
    Res := Method.Invoke(Calc, [2, 3]);      // call Add(2, 3)
    ShowMessage(Res.AsInteger.ToString);      // '5'
  finally
    Ctx.Free;
    Calc.Free;
  end;
end;
```

Look at the signature the RTL gives us: `Invoke(Instance: TObject; const Args: array of TValue): TValue`. Every argument is a `TValue` (the integers `2` and `3` box themselves via the implicit operators from Part 1), and the return value is a `TValue` you unwrap with `AsInteger`, `AsString`, or whichever accessor fits. For a procedure like `Reset` that returns nothing, you simply ignore the (empty) result.



Invoke: name + boxed args go in, a boxed result comes back

The diagram is the entire mechanism: a method name plus boxed arguments go in, the method runs on your live object, and a boxed result comes back. A command dispatcher is just this call with the name and args pulled from an incoming message.

> **Invoke also constructs objects and calls class methods**

> `Invoke` has overloads for a `TClass` and a `TValue` instance too — which means you can call **constructors** reflectively. Find the `Create` method via `GetMethod('Create')` and `Invoke` it on the *class* to build an instance when you only know the class at runtime. That's precisely how dependency-injection containers instantiate the types they're asked for. We won't build a full container here, but it's the same `Invoke` you just saw.

> **Invoke needs the right arguments — mismatches raise at runtime**
> The safety net the compiler normally gives you is gone here: if you pass the wrong number or types of arguments, you don't get a compile error — you get an `EInvocationError` (or a conversion error) *at runtime*.
>
> That's the fundamental trade of reflection: flexibility for compile-time checking. Guard invocations you build from external input, and lean on the RTL's typed helpers where you can. This is exactly why, as Part 1 stressed, you reach for `Invoke` only when you genuinely can't name the method in source.

The capstone: a generic CSV exporter

Let's put all three parts to work on something you might actually ship. The goal: **export any `TObjectList<T>` to CSV** — header row from the property names, one line per object — with a single routine that knows nothing about the classes it's handed. This is the kind of small win reflection is perfect for.

We'll even let a class *opt a property out* of the export with a custom attribute, tying in Part 2. First the attribute:

```
type
  // Mark a property to be skipped by the exporter (Part 2's attributes)
  NoExportAttribute = class(TCustomAttribute);

  TEmployee = class
  private
    FName: string;
    FSalary: Currency;
    FPasswordHash: string;
  public
    property Name: string read FName write FName;
    property Salary: Currency read FSalary write FSalary;
    [NoExport]                                  // never export this
    property PasswordHash: string read FPasswordHash write FPasswordHash;
  end;
```

Now the exporter. It walks the properties once to build the header (skipping anything marked `[NoExport]`), then walks each object's values to build the rows — using `GetProperties`, `GetValue`, `HasAttribute<T>`, and `TValue.ToString` from across the series:

```
uses
  System.Rtti, System.SysUtils, System.Classes, System.Generics.Collections;

function ExportToCsv<T: class>(const Items: TObjectList<T>): string;
var
  Ctx: TRttiContext;
  Typ: TRttiType;
  Props: TArray<TRttiProperty>;
  Prop: TRttiProperty;
  Item: T;
  Line: TArray<string>;
  SB: TStringBuilder;
```

```
  // include a property only if it's readable and not marked [NoExport]
  function Exported(const P: TRttiProperty): Boolean;
  begin
    Result := P.IsReadable and not P.HasAttribute<NoExportAttribute>;
  end;

begin
  Ctx := TRttiContext.Create;
  SB := TStringBuilder.Create;
  try
    Typ := Ctx.GetType(TClass(T));
    Props := Typ.GetProperties;

    // 1. Header row — property names
    Line := [];
    for Prop in Props do
      if Exported(Prop) then
        Line := Line + [Prop.Name];
    SB.AppendLine(string.Join(',', Line));

    // 2. One row per object — property values via reflection
    for Item in Items do
    begin
      Line := [];
      for Prop in Props do
        if Exported(Prop) then
          Line := Line + [Prop.GetValue(Item).ToString];
      SB.AppendLine(string.Join(',', Line));
    end;

    Result := SB.ToString;
  finally
    SB.Free;
    Ctx.Free;
  end;
end;
```

Feed it a list of employees and you get:

```
Name,Salary
Ada Lovelace,95000
Grace Hopper,102000
```

`PasswordHash` is absent — the `[NoExport]` attribute did its job — and, crucially, **this routine will export `TCustomer`, `TInvoice`, or any class you write next year, with zero changes.** That's the whole promise of reflection delivered in about forty lines: write the walker once, reap it forever. Here's the flow at a glance.



One generic walker: property names become the header, values become rows

The diagram is the tool in one line: a list of unknown objects goes in, the reflection walk turns property names into a header and property values into rows, and CSV comes out — no per-class code anywhere.

> **Make it production-ready before you ship it**
> To keep the example focused, it omits real-world niceties: **CSV escaping** (values containing commas, quotes, or newlines must be quoted per RFC 4180), culture-aware number/date formatting, and choosing properties vs. fields. Add those and it's genuinely shippable. The *reflection* part — the hard part — is already complete and correct; the rest is ordinary formatting.

The other side: is reflection the right call here?

Guarantee to the reader — even for a nice tool, ask whether reflection earns its place. For a CSV exporter that must handle *arbitrary* classes across your codebase, yes: the alternative is a hand-written exporter per class, which is exactly the boilerplate reflection exists to erase. But if you only ever export *one* known class, a plain hand-written function is simpler, faster, and safer — no `TValue` boxing, full compile-time checking. The rule from Part 1 holds all the way to the end: **reflect when you're writing code that must span many types you don't control; write direct code when you know the type.** The exporter qualifies because "any class" is in its very signature.

> **Alternatives worth knowing**
> You rarely need to hand-roll serialization from scratch. The RTL's `REST.Json` already does object”JSON via RTTI, and mature open-source libraries like Delphi-JSON / JsonDataObjects and ORMs such as mORMot lean heavily on reflection for serialization and mapping. Across stacks, .NET's `System.Text.Json` and Java's Jackson do the same job with the same reflective ideas. Reach for a proven library when one fits; build your own reflective tool when your need is specific — now you know exactly how they work inside.

Takeaways — the whole series

Three parts took us from "what is RTTI" to a working tool. The complete picture:

- **Invoke a method by name** with `TRttiMethod.Invoke(Instance, [args])` — arguments go in as `TValue`, the result comes back as `TValue`. The overloads even let you call constructors and class methods reflectively (the heart of DI containers).
- Reflection **loses compile-time checking**: wrong arguments raise `EInvocationError` at runtime. Guard invocations built from external input.
- The **capstone exporter** shows the payoff: `GetProperties` + `GetValue` + `HasAttribute<T>` produce a CSV writer that works on *any* class, written once — the essence of every serializer and mapper.
- The through-line of all three parts: **reflect when your code must span types you don't control; write direct code when you know the type.**

> Invocation completes the trio — read (`GetValue`), write (`SetValue`), and call (`Invoke`) any member by name — turning "types as data" into working tools like a class-agnostic CSV exporter in a few dozen lines.

That closes the RTTI series. If you want the foundations again, Part 1 is the "what and why" and Part 2 is properties and attributes. You now understand the machinery behind Delphi's serializers, ORMs, and DI containers — and you can build your own when the need is genuinely yours. Go reflect on something.

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

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 —