Properties (and Why F-Fields Are Evil)

> *A property looks like a public field. It is not. The difference between the two is the difference between a class that survives ten years of changes and a class that becomes a museum piece six months after release.*

Pre-requisites: Chapter 11 (Classes and Objects), Chapter 12 (Inheritance and Polymorphism).
What you will learn: the property contract; the five forms of property; what `default`, `stored`, and `nodefault` tell the streaming system; indexed and array properties; class properties; and the rules the rest of this book leans on - never expose F-fields, every property has read and write, no `Exit`, no `Inc` on a property, never reuse a Delphi-defined property name.
Estimated reading time: 35 minutes.

Why this chapter exists

The single most common refactoring failure in long-lived Object Pascal codebases is the field that became a property and broke a hundred caller sites. A class ships with a public `Count` field. Two years later somebody needs the count to be derived from a list, or to validate a maximum, or to fire an event when it changes. The class wants a getter and a setter, and the only way to give it one is to rename the field, which means every line in every other unit that reads or writes `Count` stops compiling.

Properties are the answer to that problem. The good news is that they are easy. The bad news is that almost every novice Object Pascal codebase fails to use them correctly, and almost every senior Object Pascal codebase has at least one place where someone reached past a property to touch the underlying field and broke an invariant the property was holding up.

This chapter explains what a property actually is, shows the five forms, presents the rules, and gives you a real-world example - a configurable rate limiter - that exercises every form.

13.1 What a property actually is

A property is a published *name* that the compiler maps to a getter, a setter, or both. The getter and setter can be fields (the simple case) or methods (the not-simple case that earns its keep within minutes). When you write `MyObject.Count := 5;`, the compiler looks at the property declaration to decide whether that statement assigns to a field, calls a method, or is a compile error.

Listing 13.1 shows the simplest possible property. It does almost nothing.

```
type
    TUser = class
    strict private
        FUsername: string;
    public
        property Username: string read FUsername write FUsername;
    end;
```

Listing 13.1 - A property that reads and writes the same field. Form 1.

Reading `SomeUser.Username` compiles to a load of `SomeUser.FUsername`. Writing `SomeUser.Username := 'alice'` compiles to a store into `SomeUser.FUsername`. There is no method call in either direction. The generated machine code is identical to what you would get from a public field. So why bother?

Because tomorrow you may need the property to do something. Listing 13.2 is the same class six months later. The public surface has not changed. Every caller still says `U.Username := 'alice'`. But every assignment now passes through a setter that can validate, log, fire an event, or refuse the assignment.

```
type
    TUser = class
    strict private
        FUsername: string;
        procedure SetUsername(const AValue: string);
    public
```

```
        property Username: string read FUsername write SetUsername;
    end;

procedure TUser.SetUsername(const AValue: string);
BEGIN
    TRY
        IF Trim(AValue) = '' THEN
            RAISE EArgumentException.Create('Username cannot be blank.');
        FUsername := AValue;
    EXCEPT
        RAISE;
    END;
END;
```

Listing 13.2 - The same class with a setter that validates. Form 2. No caller of TUser had to change.

This is the contract a property gives you. The public name is stable. The implementation behind the name is yours to change. A field gives you neither - the field is the implementation. Change the implementation, every caller breaks.

Warning - never use a Delphi-defined property name. The first draft of this class called the property `Name`. It was wrong. `Name` is the property the streaming system uses to identify a component on a form, declared on `TComponent`. A class that declares its own `Name` shadows the inherited one and breaks the streaming system the moment the class is dropped on a form. Even outside `TComponent`, names that collide with framework identifiers - `Name`, `Tag`, `Owner`, `Parent`, `Caption`, `Action`, `Hint` - confuse readers and create future-version risk. Pick a domain-specific name (`Username`, `ProductName`, `RegionName`) every time.

13.2 The five forms of property

Object Pascal properties come in five forms, ranked by how much work the compiler is doing on your behalf.

Form	read	write	What it does
1	field	field	Identical to a public field at the machine-code level. Use this when you want the option to add behaviour later but need

			none today.
2	field	method	The reader is a load. The writer goes through a setter. Use this when you need to validate or react to writes but reads are unconstrained.
3	method	field	The reader goes through a getter. Use this when the value is derived from other state.
4	method	method	Both go through methods. Use this when both directions need behaviour.
5	method or field	(none)	Read-only property. Or write-only with directions reversed - rare.

Form 5 is a Delphi language feature. The codebase convention used in this book restricts when you should reach for it. The convention - introduced as Rule 13.4 in section 13.11 below - is that every property has a `read` and a `write` directive. When the value is purely derived from other state and a setter would be meaningless, you write a public method instead of a read-only property. Section 13.3 shows the language form 5 for completeness; the worked example in section 13.9 demonstrates the convention.

> **Tip.** A getter that simply returns the field and a setter that simply assigns the field are both candidates for the `inline` directive. The compiler then emits the same code as a direct field access while leaving you the freedom to add behaviour later.

13.3 Read-only, write-only, and the index trick

Drop the `write` clause and the property is read-only at the language level. The compiler refuses any assignment to it.

```
type
    TInvoice = class
    strict private
        FLines: TList;
    public
```

```
        function Total: Currency;
    end;
```

Listing 13.3 - Pure-derivation read access. The codebase convention prefers a public method here over a read-only property because the value cannot be set.

The `index` directive lets one method back many properties. Useful when several properties differ only in which slot they touch.

```
type
    TVector3 = class
    strict private
        FValues: array[0..2] of Double;
        function GetCoord(AIndex: Integer): Double;
        procedure SetCoord(AIndex: Integer; const AValue: Double);
    public
        property X: Double index 0 read GetCoord write SetCoord;
        property Y: Double index 1 read GetCoord write SetCoord;
        property Z: Double index 2 read GetCoord write SetCoord;
    end;
```

Listing 13.4 - Three properties, one getter, one setter. The index value is passed to the method.

13.4 Default, stored, nodefault - what published properties tell the streaming system

When a property is in the `published` section of a class that descends from `TPersistent`, the streaming system can read it from a .dfm at design time and write it back when the form is saved. Three directives shape that round trip.

- **`default V`** - tells the streamer that if the property's runtime value equals `V`, the .dfm does not need to record it. This keeps form files small and diffable.
- **`nodefault`** - tells the streamer to always record the value, even when it matches the type's natural default. Use this when there is no sensible default.
- **`stored M`** - calls the boolean method `M` at streaming time. The streamer records the property only when `M` returns true.

> **Warning.** The `default` directive in a property declaration has nothing to do with the `default` keyword on an array property in section 13.6. Same word, different feature, different scope.

13.5 Indexed properties

An indexed property is a property whose name takes a parameter list, much like a method. The most common form is the integer index used to address into an internal collection.

```
type
    TStringRing = class
    strict private
        FItems: array of string;
        function GetItem(AIndex: Integer): string;
        procedure SetItem(AIndex: Integer; const AValue: string);
    public
        property Items[AIndex: Integer]: string read GetItem write SetItem;
        function Count: Integer;
    end;
```

Listing 13.5 - An indexed property over a dynamic array. `Count` is a method because its value is purely derived from the array length.

13.6 Array properties and the `default` array property

An array property may be marked as the *default* array property of its class. The class is then allowed exactly one default array property. When you write `Ring[3]` the compiler reads it as `Ring.Items[3]` through the default array property.

```
type
    TStringRing = class
    strict private
        /// ... as before ...
    public
        property Items[AIndex: Integer]: string read GetItem write SetItem; de
    end;
```

```
/// usage:
procedure TestStringRing;
var
    Ring: TStringRing;
BEGIN
    TRY
        Ring := TStringRing.Create;
        TRY
            Ring[0] := 'first';
            WriteLn(Ring[0]);
        FINALLY
            Ring.Free;
        END;
    EXCEPT
        RAISE;
    END;
END;
```

Listing 13.6 - A default array property. The class can be indexed directly without naming the property.

13.7 Class properties

A class property belongs to the class itself, not to an instance.

```
type
    TLogger = class
    strict private
        class var FMinLevel: Integer;
        class function GetMinLevel: Integer; static;
        class procedure SetMinLevel(const AValue: Integer); static;
    public
        class property MinLevel: Integer read GetMinLevel write SetMinLevel;
    end;

/// usage:
TLogger.MinLevel := 2;
```

Listing 13.7 - A class property. Getter and setter are `static`; they receive no `Self`. Declaration is on a single line.

13.8 The rule: never expose the F-field

> **RULE 13.1**
>
> **Never expose an F-field outside the class that owns it. Always go through a property.**
>
> **Why.** A field has no behaviour. The day you need behaviour, you have to rename the field, which breaks every caller. A property is a stable name that maps to whatever implementation is behind it.

The corollary applies inside the class too. Even within a class's own methods, you read and write through the property name, not through the F-field. Three places legitimately touch an F-field directly: the property declaration itself (`read FX write FX`), the getter and setter implementations, and the constructor and destructor. Everywhere else - business-logic methods, helper methods, internal calculations - you go through the property name.

> **Warning.** The Object Inspector reads `published` properties via RTTI. If a class has a published property and a public F-field with the same name (worse: differing types), the streaming system records the property and ignores the field. The class then loads with values for properties only. Never declare a field and a property that read or write each other circularly.

13.9 Real-world worked example - a configurable rate limiter

> **Real-world worked example.** Build `TRateLimiter`, a class that allows up to *N* operations per *T* seconds and returns false from `TryAcquire` when the limit is exceeded. The class will exercise every property form covered in this chapter and will obey every codebase rule introduced in this book so far.

Listing 13.8 shows the unit and class declaration. Listing 13.9 shows the implementation. Note the unit-name pattern `Obj.<Descriptor>.<Function>`, the unit-header block, the named constants for defaults instead of magic numbers, the field declarations before any

methods within each visibility section, and getters and setters at the bottom of the implementation.

```
/// Obj.Sample.RateLimiter
/// Author: book companion
/// Created: 02/05/2026
/// Modified: 02/05/2026
/// Version: 1.0
/// Notes: Demonstrates every property form covered in Chapter 13.
/// Usage: drop a TRateLimiter on a form, set MaxPerWindow and WindowSeconds,

unit Obj.Sample.RateLimiter;

interface

uses
    System.Classes, System.SysUtils, System.DateUtils, System.Generics.Collect

const
    DEFAULT_MAX_PER_WINDOW = 100;
    DEFAULT_WINDOW_SECONDS = 60;

type
    TRateLimiter = class(TComponent)
    strict private
        FMaxPerWindow: Integer;
        FWindowSeconds: Integer;
        FAllowed: Int64;
        FRefused: Int64;
        FActive: Boolean;
        FTimestamps: TList<TDateTime>;
        procedure SetMaxPerWindow(AValue: Integer);
        procedure SetWindowSeconds(AValue: Integer);
        procedure SetAllowed(AValue: Int64);
        procedure SetRefused(AValue: Int64);
        property Timestamps: TList<TDateTime> read FTimestamps write FTimestam
    public
        constructor Create(AOwner: TComponent); override;
        destructor Destroy; override;
        function TryAcquire: Boolean;
        function RecentCount: Integer;
        function RecentDelay(AIndex: Integer): Integer;
        property Allowed: Int64 read FAllowed write SetAllowed;
        property Refused: Int64 read FRefused write SetRefused;
    published
```

```
        property MaxPerWindow: Integer read FMaxPerWindow write SetMaxPerWindo
        property WindowSeconds: Integer read FWindowSeconds write SetWindowSec
        property Active: Boolean read FActive write FActive default True;
    end;

implementation
```

Listing 13.8 - The TRateLimiter declaration. Every property has a read and a write directive. RecentCount and RecentDelay are public methods (form-5 read-only would have violated the codebase convention). Field declarations come before method declarations within each visibility section.

```
constructor TRateLimiter.Create(AOwner: TComponent);
BEGIN
    TRY
        inherited Create(AOwner);
        FTimestamps := TList<TDateTime>.Create;
        MaxPerWindow := DEFAULT_MAX_PER_WINDOW;
        WindowSeconds := DEFAULT_WINDOW_SECONDS;
        FActive := True;
    EXCEPT
        RAISE;
    END;
END;

destructor TRateLimiter.Destroy;
BEGIN
    TRY
        FreeAndNil(FTimestamps);
    EXCEPT
        /// destruction never raises – swallow deliberately
    END;
    inherited Destroy;
END;

function TRateLimiter.TryAcquire: Boolean;
var
    Cutoff: TDateTime;
BEGIN
    Result := False;
    TRY
        IF NOT Active THEN
            BEGIN
                Allowed := Allowed + 1;
                Result := True;
```

```
            END
        ELSE
            BEGIN
                Cutoff := IncSecond(Now, -WindowSeconds);
                WHILE (Timestamps.Count > 0) AND (Timestamps[0] < Cutoff) DO
                    Timestamps.Delete(0);
                IF Timestamps.Count >= MaxPerWindow THEN
                    Refused := Refused + 1
                ELSE
                    BEGIN
                        Timestamps.Add(Now);
                        Allowed := Allowed + 1;
                        Result := True;
                    END;
            END;
    EXCEPT
        ON E: Exception DO
            BEGIN
                /// fail closed - on any internal error, refuse the operation
                Refused := Refused + 1;
                Result := False;
            END;
    END;
END;

function TRateLimiter.RecentCount: Integer;
BEGIN
    TRY
        Result := Timestamps.Count;
    EXCEPT
        RAISE;
    END;
END;

function TRateLimiter.RecentDelay(AIndex: Integer): Integer;
BEGIN
    TRY
        IF (AIndex < 0) OR (AIndex >= Timestamps.Count - 1) THEN
            RAISE EArgumentOutOfRangeException.Create('RecentDelay index out o
        Result := MilliSecondsBetween(Timestamps[AIndex + 1], Timestamps[AInde
    EXCEPT
        RAISE;
    END;
END;

/// --- setters at the end of the unit, per codebase convention ---
```

```
procedure TRateLimiter.SetMaxPerWindow(AValue: Integer);
BEGIN
    TRY
        IF AValue < 1 THEN
            RAISE EArgumentException.CreateFmt('MaxPerWindow must be at least
        FMaxPerWindow := AValue;
    EXCEPT
        RAISE;
    END;
END;

procedure TRateLimiter.SetWindowSeconds(AValue: Integer);
BEGIN
    TRY
        IF AValue < 1 THEN
            RAISE EArgumentException.CreateFmt('WindowSeconds must be at least
        FWindowSeconds := AValue;
    EXCEPT
        RAISE;
    END;
END;

procedure TRateLimiter.SetAllowed(AValue: Int64);
BEGIN
    TRY
        IF AValue < 0 THEN
            RAISE EArgumentException.Create('Allowed cannot be negative.');
        FAllowed := AValue;
    EXCEPT
        RAISE;
    END;
END;

procedure TRateLimiter.SetRefused(AValue: Int64);
BEGIN
    TRY
        IF AValue < 0 THEN
            RAISE EArgumentException.Create('Refused cannot be negative.');
        FRefused := AValue;
    EXCEPT
        RAISE;
    END;
END;
```

```
end.
```

Listing 13.9 - Implementation of TRateLimiter. Constructor and destructor first, public methods next, all setters at the end. Every method body has a try/except wrapper. No `Exit`, no `Inc`. Business logic reads through property names (Active, Timestamps, MaxPerWindow, WindowSeconds, Allowed, Refused). Counters increment by assignment: `Allowed := Allowed + 1`. Control-flow keywords (BEGIN, END, TRY, EXCEPT, IF, THEN, ELSE, WHILE, DO, RAISE, ON, NOT, AND, OR) in uppercase per the codebase convention. Comments start with `///`.

Compile this unit, drop a `TRateLimiter` on a form, set its `MaxPerWindow` in the Object Inspector. Now try to set it to `0`. The setter raises, the Object Inspector flashes red, the value reverts. The *property contract* is doing its job.

13.10 Pitfalls

1. **The setter that loops.** A setter that assigns to the property instead of the field recurses forever. `SetUsername` writes `FUsername := AValue`, never `Username := AValue`. The compiler will not save you - the recursive call type-checks fine.
2. **Reaching around the property.** A method body that reads or writes the F-field directly bypasses every behaviour the property's getter or setter was meant to enforce. Even within the class that owns the field, go through the property name. The only exceptions are the constructor, the destructor, and the getter / setter implementations themselves.
3. **Using `Inc()` or `Dec()` on a property.** Both intrinsics take a var parameter, and a property is not a variable. The compiler refuses. Worse, if you reach for the F-field instead so you can write `Inc(FAllowed)`, you have just bypassed the property and broken Rule 13.1. Use the assignment form: `Allowed := Allowed + 1`.
4. **Using `Exit` to leave a function.** `Exit` is forbidden in this book's codebase. It obscures control flow and skips work the function intended to do at its bottom. Restructure with a result variable assigned at the top, then nested if/else, then a single fall-through to `END`.
5. **Naming a property after a Delphi-defined identifier.** `Name`, `Tag`, `Owner`, `Parent`, `Caption`, `Action`, `Hint` are properties or fields the framework already uses. Pick a domain-specific name every time.

6. **Declaring a field after a method in the same visibility section.** Fields must come before methods and properties within each visibility scope. The compiler refuses the inverse order with an obscure message; reorder so all F-vars sit at the top of their section.
7. **Forgetting the `///` comment style.** The codebase uses `///` for every comment, not `//`. Mixed styles confuse parsers and tooling.
8. **Hard-coded magic numbers as defaults.** A `default 100` on a published property is a maintenance trap. Define a named constant (`DEFAULT_MAX_PER_WINDOW = 100`) and reference the constant in the directive and in the constructor.
9. **Adding a getter or setter that does nothing but touch the F-field.** If the body is just `Result := FX;` or `FX := AValue;`, drop the method and use the field directly in the property declaration: `read FX write FX`.
10. **Putting getters and setters in the middle of the unit.** The codebase convention places all getters and setters at the END of the implementation. Constructors, destructors, and public methods come first.
11. **The getter that allocates without freeing.** A getter that lazily creates a sub-object on first read needs to keep a field reference and free that reference in the destructor. The destructor must be ready for the field being nil.
12. **The published property that uses a complex setter at design time.** The Object Inspector calls the setter while the form is loading. A setter that touches another component the form has not yet created crashes the IDE. Defend with a `csLoading in ComponentState` guard.
13. **The default-array-property collision.** A class can have only one default array property. Inheriting from a class that already has one and declaring another is a compile error.
14. **The visibility mismatch.** A property in `public` whose setter is in `private` still lets the public outside world write to the property. The visibility rule is on the property name, not on the methods it calls.
15. **The F-field accidentally promoted to `public`.** Always put fields in `strict private`. Promote to `private` only with documented reason.

13.11 Rules introduced in this chapter

RULE 13.1

Never expose an F-field outside the class that owns it. Always go through a property.

Why. Properties are a stable public surface; fields are an implementation detail. Exposing the field welds the implementation to every caller.

RULE 13.2

Inside a class's own methods, read and write through the property name. The F-field is touched only in the property declaration, the getter / setter, the constructor, and the destructor.

Why. The getter sometimes does work the field does not - lazy initialisation, derived state, instrumentation. Bypassing the getter from inside the class skips that work.

RULE 13.3

Put fields in `strict private`. Promote to `private` only with a documented reason.

Why. `strict private` is invisible even to other code in the same unit. The default of "even my own unit cannot see it" forces every read and write through the public surface.

RULE 13.4

Every property has a `read` and a `write`. A purely-derived value with no meaningful setter is exposed as a public method, not as a read-only property.

Why. Read-only properties create an asymmetry the rest of the toolchain assumes does not exist - design-time editors, form streaming, RTTI marshallers all expect read and write together. A method clearly signals "computed, no setter possible" and avoids the asymmetry.

RULE 13.5

Do not add a getter or setter that only touches the F-field. Use direct field reference in the property declaration: `read FX write FX`.

Why. A method body of one line that does no work is noise. The property declaration is the seam; do not duplicate it.

RULE 13.6

Never use `Inc()`, `Dec()`, or `Exit`. Increment counters by assignment (`X := X + 1`). Restructure early-exit functions with a result variable and nested if/else.

Why. `Inc` on a property is illegal anyway and `Inc` on an F-field bypasses the property. `Exit` obscures control flow and skips work the function intended to do at its bottom.

RULE 13.7

Every published property declares its `default` or is marked `nodefault`. The default value comes from a named constant, not a magic number.

Why. The streaming system uses the default to decide whether to write the property to the .dfm. Named constants make the default value findable, changeable, and documented.

Summary

- A property is a public name that the compiler maps to a getter, a setter, or both.
- Five forms exist - the codebase convention restricts form 5 (read-only) to a public method when the value is purely derived.
- The `index` directive lets one method back many properties.
- The `default`, `stored`, and `nodefault` directives shape how the streaming system serialises a published property.
- Indexed and array properties give you collection-style access. The default array property lets you index the class itself.
- Class properties belong to the class, not the instance.
- Never expose F-fields. Every property has a read and a write. No `Exit`. No `Inc` on a property. No Delphi-reserved property names. Constants for default values. Comments start with `///`. Field declarations before methods. Getters and setters at the END of the unit.

Exercises

1. **Easy.** Convert `TBookshelf` from Chapter 11 so all field access is via properties, every property has read and write, and pure-derived values become methods. The public surface must not change.
2. **Easy.** Add a `SetCount` setter to `TStringRing` from Listing 13.5 that resizes the underlying dynamic array, validating the new count is at least 0.
3. **Medium.** Add an `OnRefused` event to `TRateLimiter` that fires every time `TryAcquire` refuses an operation. The event must be a published property of type `TNotifyEvent` editable in the Object Inspector.
4. **Medium.** Write a `TBoundedQueue<T>` generic class with a default array property indexer, a published `MaxItems` with a validating setter, public methods `Count`, `Enqueued`, and `Dropped` for derived values, and full obedience to every rule introduced in this chapter.
5. **Hard.** Build a `TConfig` class whose properties are populated from environment variables on first read, with a unit-tested behaviour that reading the same property a second time does not re-read the environment.
6. **Hard.** Take the `TRateLimiter` from Listing 13.9 and add full DUnitX tests covering every property form, every validation rule, the `Active = False` bypass behaviour, and the cutoff-roll behaviour at the end of a window.

Solutions are in the companion repository under `chapters/13-properties/exercises/`.

Further reading

- Embarcadero DocWiki, "Properties (Delphi)" - `https://docwiki.embarcadero.com/RADStudio/en/Properties_(Delphi)`.
- This book, Chapter 17 (VCL Windows Applications), section 17.3 - design-time properties in published collections.

- - -