

Delphi

THREAD SAFETY PATTERNS

*Dalija Prasnikar
Neven Prasnikar Jr.*

Dalija Prasnikar

Neven Prasnikar Jr.

Delphi Thread Safety Patterns

Copyright ©2022 Dalija Prasnikar & Neven Prasnikar Jr.

First Edition

Contents

Introduction	11
Acknowledgements	12
Code	12
Important Note	12
Part 1. Thread Safety	17
1 Language and general thread safety	17
1.1 Thread safety categorization	20
1.2 General thread safety facts	23
2 Proving thread safety	25
3 Examples	29
3.1 Data type thread safety	29
3.1.1 Example 1. Thread-unsafe class with read/write property	29
3.1.2 Example 2. Thread-safe class with read-only property	30
3.1.3 Example 3. Thread-safe class with read/write property	31
3.1.4 Example 4. Thread-unsafe class with read-only property and mutating method	32
3.1.5 Example 5. Thread-safe record with read-only property	33
3.1.6 Example 6. Thread-safe class with read-only object property	34
3.2 Thread-safe data types used in an unsafe manner	38

3.2.1	Example 7. Thread-safe integer list	39
3.2.2	Example 8. Thread-safe usage of thread-safe data type	41
3.2.3	Example 9. Thread-safe usage of thread-safe data type	42
3.2.4	Example 10. Thread-unsafe usage of thread-safe data type	44
3.3	Code thread safety	47
3.3.1	Example 11. Thread-safe data handover between threads	48
3.3.2	Example 12. Thread-unsafe data handover between threads	51
Part 2. The Core Run-Time Library		55
4 Global state		55
5 Floating-point control register		57
5.1	FPCR-related thread-unsafe API	62
5.1.1	Posix platforms	62
5.1.2	Windows platforms	63
5.2	A thread-safe design for FPCR management	65
6 FormatSettings and formatting routines		69
6.1	SysLocale	71
6.2	Creating custom thread-safe format settings	73
6.2.1	Multiple, non-writeable typed constants	76
6.2.2	Multiple, unprotected variables, initialized only once	76
6.2.3	Using private instances	77
6.2.4	TFormatSettings.Invariant	80
6.2.5	Factory functions	80
6.2.6	Immutable types	81
6.2.7	Complete functionality wrapper	82
6.2.8	Wrapper with locks	85

7 Global functions and procedures	89
7.1 Parameters	89
7.2 FreeAndNil	92
7.3 Class methods	96
7.4 Other global functions	96
8 Class fields, singletons, and default instances	99
8.1 Class constructors and initialization section	101
8.2 Thread-safe lazy initialization	103
8.3 Writeable instances	108
9 Core classes	111
9.1 Custom classes	112
10 Threads	117
10.1 FreeOnTerminate	119
10.2 Starting a thread	126
10.3 Order of destruction	128
10.4 Custom thread data	130
11 Streams	133
12 Collections	139
12.1 Making a thread-safe variant of a thread-unsafe collection	144
12.2 Thread-safe iteration	147
12.2.1 Iterating method with procedural parameter	149
12.2.2 Thread-safe enumerator	150
12.2.3 Exposing internal collection	152
12.3 Partial locking	155
12.4 False sense of security	160
12.5 Wrappers vs inheritance	164
12.6 Immutable collections	169

13 Parallel collection processing	171
13.1 Independent processing of individual collection items	172
13.1.1 Single-threaded processing	172
13.1.2 Multiple threads	172
13.1.3 Multiple tasks	173
13.1.4 TParallel.For	174
13.1.5 Batch processing with multiple threads	175
13.2 Dependent processing of individual collection items	176
13.2.1 Single-threaded processing	176
13.2.2 TParallel.For	177
14 Components	183
14.1 Using components in background threads	185
14.2 Component streaming	188
15 RTTI	193
Part 3. Core Frameworks	201
16 Serialization	201
16.1 Protecting the data	203
16.1.1 Unshared data	204
16.1.2 Read-only data	206
16.1.3 Mutable data protected by a synchronization mechanism	207
16.2 Protecting the serializer	217
16.2.1 Function, procedure or class method	217
16.2.2 Stateless or immutable serializer instance	217
16.2.3 Stateful serializer instance	218
16.3 Net encoding classes	218
16.4 JSON	221
16.5 XML	223
16.5.1 Guide for thread-safe XML serialization	224

17 System.Net	227
18 Asynchronous Programming Library	235
18.1 Wrong level of abstraction	237
18.2 Complicated exception handling	239
18.3 Difficult to debug	239
18.4 Tight coupling between the UI and asynchronous code	243
18.5 Not suitable for running multiple operations	243
19 Indy	249
20 REST	253
21 Regular expressions	257
Part 4. Visual Frameworks - VCL and FMX	263
22 Visual framework components	263
23 LiveBindings	265
24 VCL and FMX controls	267
25 Interactions with OS APIs and frameworks	269
25.1 Window allocation and de-allocation	270
25.2 Windows messaging	272
25.3 Timers	274
Part 5. Graphics and Image Processing	279
26 Graphics and image processing	279
26.1 Graphics platforms and frameworks	281
27 Resource consumption	283

28 Common graphics types and API	289
29 VCL graphics types and API	291
29.1 Fonts, pens, and brushes	292
29.2 Canvas	293
29.3 Graphics	295
29.3.1 TBitmap	297
29.3.2 TIcon	298
29.3.3 TMetafile	298
29.3.4 TWICImage	298
29.3.5 TJPEGImage	298
29.3.6 TPngImage	299
29.3.7 TGIFImage	299
29.4 Picture	300
29.5 Vcl.GraphUtil	300
29.6 VCL image collections	300
30 VCL graphics example	303
31 FMX graphics types and API	311
Part 6. Custom Frameworks	315
32 Writing custom frameworks	315
33 Logging	317
33.1 Logging levels	319
33.2 Cleanup on shutdown	321
33.3 Loggers	324
33.3.1 System Logger	324
33.3.2 File logger	327
33.4 Logging extensions	329

33.4.1 Thread identification	331
33.4.2 Timestamp	335
33.4.3 Logging level	335
33.4.4 Combining extensions	336
33.5 Final thread safety check	338
34 Cancellation tokens	341
34.1 Cancellations without tokens	342
34.2 Cancellations with a token	345
34.3 Tokens, tasks, and async results	348
34.4 Non-cancellable tokens	351
35 Event bus	355
35.1 Event bus building blocks	357
35.1.1 Events and categories	357
35.1.2 Event delivery options	363
35.1.3 Publishing	365
35.1.4 Subscribers	366
35.1.5 Topics or channels	369
35.2 Event bus implementation	370
35.3 Back-pressure	378
36 Measuring performance	381
36.1 Profiling	382
36.2 Benchmarking	382
Appendix	395
References	395
Quality Portal Reports	399

Introduction

While the thread safety of a particular piece of code depends on the surrounding context and how it is used, some data types are inherently unsafe, and for some of them, thread safety will depend on the use case and the specific code. Unfortunately, when you look at some class or type declaration or API alone, there is very little information there that will tell you whether instances of that type can be safely used in multiple threads, or under which conditions. The proper place to learn about thread safety is the documentation. However, most documentation (unless you are dealing with *multithreading* libraries, where multithreading is the main feature) will not explicitly give you that information.

Occasionally, the documentation will mention that a particular feature is not thread-safe, or will tell you that a feature can be used in background threads, but for the most part, you will have to figure out thread safety on your own.

One reason for this is that thread safety depends on the context. The number of features that are absolutely unsafe or are absolutely safe is very small. It is also assumed that if you are familiar with multithreading, you will be able to recognize some patterns on your own, and that you don't need explicit guidance for every use case. For the rest, properly documenting thread-safe usage might involve writing a general thread safety tutorial for every class.

Another problem is that some libraries, like visual frameworks, are thread-unsafe by definition, while only some of their bits and pieces can be used in background threads under specific conditions.

Besides reading documentation, the only other thing you can do to determine thread safety is to read the code of the implementation. And that can be a daunting task. Experience helps, but when you don't have much experience, this may be a problem.

This book is a sequel to the *Delphi Event-based and Asynchronous Programming* book, which provides an introduction to asynchronous programming and multithreading, as well as explaining concepts of thread safety and the general problems and solutions one may face.

This book goes one step further: It converts that knowledge into practice, and gives an overview of core Delphi frameworks and commonly used features from a thread safety perspective. You will find examples of how to use particular classes in a thread-safe manner and how to perform some common tasks, following already-established thread safety patterns.

Besides elaborating on the thread safety of particular parts of Delphi frameworks, the book explores the thread safety of alternate solutions, along with general coding examples. Every explanation about why some code is thread-safe or not, also serves as an example of thread (un)safety patterns and helps in recognizing thread-unsafe code as well as establishing a working set of thread safety patterns that can later be applied in custom code.

Those examples, covering the most commonly used parts of Delphi frameworks, will also serve as *learn by example* pointers for determining the thread safety of other parts of the frameworks, and even of 3rd-party libraries that are not specifically covered in this book. Delphi frameworks are huge, and this book is not meant to be a *complete reference guide*, where the thread safety of every available variable, class and function is examined.

There are also implementations and examples of some commonly used concepts in asynchronous programming that are not part of the core Delphi frameworks, but can be indispensable when writing multithreaded code.

Acknowledgements

Huge thanks to William Meyer for providing the technical review, David Heffernan for reviewing the *Floating-point control register* chapter and suggesting solutions for FPCR thread safety issues, and Sandra Prasnikar for drawing illustrations.

Code

It is hard to learn from code you cannot easily follow and reason about. Because of that, the clarity and simplicity of the presented code examples will take precedence over some small (or less small) performance gains and optimizations that would make the code harder to read. That also includes using common RTL classes in places where some more specialized, hand-crafted, classes could offer better performance in a given block of code.

Once you are familiar with the concept and code flow, it is easy to replace some building blocks with other more performant ones.

Code examples are available for download from GitHub:

<https://github.com/dalijap/code-delphi-thread-safety>

Important Note

Languages, compilers, frameworks, and code in general, change with time. Changes can always introduce bugs and issues, or they can fix them. Thread safety is an extremely fragile concept and can easily be broken by subtle bugs or changes in behavior. Because multithreading bugs

are quite often subtle, it may take some time, sometimes even years, to discover and fix them. Some undiscovered bugs are lurking even now.

So while the information presented in this book regarding the thread safety of some particular code should be correct at the time of writing, it may have been broken in the past, and it is always possible it will be broken in the future.

Some more important and more visible (read: known) issues are covered, and you will only need to verify that the Delphi version you are currently using works correctly. In reality, though, there are no absolute guarantees that you will not encounter some issue that was not covered, or not even detected so far.

Part 1. Thread Safety

Chapter 1

Language and general thread safety

Before tackling the Delphi RTL and other frameworks, let's start with a brief overview of the language itself, as well as the definition of thread safety. This has been covered in more depth in my prior book: *Delphi Event-based and Asynchronous Programming*.

The definition from Wikipedia https://en.wikipedia.org/wiki/Thread_safety says:

Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction.

The term *unintended interaction (race condition)* is rather broad and open for interpretation. What can be considered an unintended interaction in one case, may be perfectly acceptable in another. Thread safety is context-dependent and without knowing the exact code and intended functionality, it may be almost impossible to say whether a given block of code is thread-safe or not.

However, when we break the code down to the tiniest level, there are some constructs that are thread-safe and some that are inherently not thread-safe. While thread-safe code pieces may not be enough to give thread safety in some broader contexts, those basic thread-safe bits and pieces will be the basic building blocks for achieving thread safety in more complex code.

Recognizing code that is not thread-safe is also an important part of the process. In many places, when you remove obviously unsafe code, the rest will (most likely) be thread-safe. Since what is *obvious* depends on the developer and their knowledge, the more you know, the less likely it is that you will write unsafe code.

In that light, examples of both thread-safe and thread-unsafe code are equally important for learning about and achieving thread safety.

The definition of thread safety talks about code thread safety. After all, data is just a passive subject that does nothing on its own. But in object-oriented programming, the data is combined with the code that operates on that data. Besides classes, record types in Delphi can also contain methods. Because some types can combine data and code, it makes sense to talk about the thread safety of the types and eventually about the thread safety of the data itself. Just because some data types don't contain code doesn't mean that instances of such types cannot still be modified by code. Whether they can actually be modified also determines whether such standalone data is thread-safe or not.

Because we can independently assess the thread safety of data types that will be used in a yet-unspecified code context, as well as the thread safety of a context where all the relevant code has already been written, the definitions of *thread-safe* and *thread-unsafe* are highly dependent on the surrounding context. It covers a fairly wide range from *this code has a bug and needs to be fixed*, *this class needs to be protected if used in threads*, to *this code cannot be fixed at all and needs complete rework because it is built upon faulty premises*.

When we are assessing the thread safety of data types, or assessing code on a more abstract level in order to define the acceptable uses of data types or code, then *thread-safe* and *thread-unsafe* only define whether the data or code in question will require additional synchronization mechanisms and thread coordination when used in real-life, multithreaded scenarios. A thread-safe data type is a type where no outside thread can ever access its instances in an inconsistent state.

We are not determining final correctness, because there is no complete code context that would allow us to judge the final correctness of the code. In other words, being thread-unsafe in that context doesn't imply existence of bugs, it is merely a warning and a usage instruction. In a broader context, using a thread-safe type also does not mean that code using it will be automatically thread-safe—it is possible to use a thread-safe data type in a thread-unsafe manner, writing code that will suffer from concurrency issues.

On the other hand, if we are examining actual code where the code context is fully defined, and all the code we are assessing at that point is fully written, then *thread-safe* and *thread-unsafe* have completely different meanings. In such an exact context, thread-safe means that the code has no concurrency related issues, while thread-unsafe means that that particular code **contains bugs** that need to be fixed.

“Fully-written” does not mean that you should not assess, nor cannot assess, the thread safety of some code while you are in process of writing it. In other words, it does not mean that you need to spend six months writing some application code before you can determine how thread-safe it is. It means that every line of code that you add has the potential to break the already-established thread safety of your codebase and that you will need to reevaluate its thread safety after every change, no matter how insignificant that change might seem.

Just as distinguishing between thread-safe and thread-unsafe code gets easier with experience and accumulated knowledge, how fast you can reevaluate the thread safety of newly added code will also grow with experience.

If multiple threads will only read data, regardless of the type, such access will always be thread-safe. Unintended interaction is impossible for data that is only being read. If even **one** of the threads can modify (write) the data, then all unprotected access (reading or writing) to such data is thread-unsafe. It is really important to remember that even one thread writing will also ruin thread safety for the threads and code that are just reading that data.

Immutable data is therefore thread-safe by definition. No matter how many threads use it, they can only read, never write. For mutable data, there is an important difference between “a thread **can** modify” and “a thread **will** modify the data”.

If code in any thread **will** modify the data, then the data is obviously not thread-safe. But what about situations where threads **could** modify the data, but they don’t and just use it for reading?

Such data will be generally considered as thread-unsafe, especially when it is accessible from a broad context, because thread safety depends on the developers’ discipline to ensure there is no code that will modify the data and break thread safety. If no code breaks the thread safety of such unprotected data, that code will also run correctly. In other words, it will be thread-safe. See: *FormatSettings and formatting routines*

This actually illustrates one of the major problems with defining thread safety. On an abstract level, mutable data is thread-unsafe, even when it’s only being read. But at the same time, if nothing is writing it, it’s technically thread-safe in that concrete context. On a concrete level, it might be considered unsafe, or it might be considered safe.

The difference between the two is that in the former case, you’re either examining a broader or more abstract context—for instance, abstract thread safety is a good thing to be mindful of when considering new libraries and frameworks—or are capable of making it safer in the concrete context you’re examining it in. In the latter, you may not be able to make it any safer than it is, or it might require writing excessively slow and/or convoluted code to be fully safe.

Another difference when talking about general thread safety is that we are talking about unknown, possibly even unwritten code, and we need to cover all scenarios. So even the possibility of unintended interaction for code that may not even exist is enough to call something thread-unsafe, while in real life you will often rely on thread safety achieved solely on the basis of not having code that will change some unprotected data.

Sometimes, you need to say that something is not thread-safe because there is already a general misunderstanding going around about what thread safety in that particular case means, and you need to break the habit of writing the wrong code, even though the thread unsafety of the code in question sits in another castle. See: *FreeAndNil*

So in many places, writing thread-safe code is a lot like following a *Pirate code*, but if you are not careful about the surrounding context and the actual code involved, the consequences can be just as deadly.

The Pirate code is more what you’d call “guidelines” than actual rules.

If possible, always write code in such a way that the thread safety of the data cannot be easily broken, but if needed, you can accomplish thread safety by following strict discipline and carefully avoiding traps and actual thread-unsafe code.

Delphi types fall into two categories, with slightly different thread safety considerations: value types and reference types.

When it comes to value types, data is stored in a single location, the variable memory, and we only need to worry about safely accessing that single location. Reference types have data stored in two locations: the variable memory that stores the reference itself, and the associated (referenced) memory allocated on the heap.

Even though reference type data is split across two locations, thread-safe access always considers those two locations independently. If you are accessing a variable, reading or writing a reference, you need to consider the thread safety of that reference alone. If you are accessing the data the reference points to, after you have established that accessing the reference itself is thread-safe, you need to separately consider the thread safety of that data.

1.1 Thread safety categorization

Instances of data types (usually classes) can also be categorized by thread safety levels:

- Thread-safe types that can be simultaneously used from multiple threads
- Thread-unsafe types that cannot be simultaneously used from multiple threads
 - types that don't have any additional restrictions besides simultaneous access
 - types that can only be constructed and used in background threads if certain conditions are met
 - types that have thread-dependent initialization/deinitialization and must be constructed and destroyed inside the thread they will be used from
- Thread-unsafe types that can only be used from the context of the main thread

When talking about the thread safety of types, we cannot talk about the thread safety of interfaces and their declarations, because they are a mere API description, and thread safety ultimately depends on each particular implementation of the interface. In other words, you can easily have both thread-safe and thread-unsafe classes implementing the same interface.

The above categorization can help when determining the appropriate usage of a particular instance, or some data type in general. I will use that categorization, combined with color coding, throughout the book to avoid repeated explanations what each of those means.

Thread-safe data type

Fully thread-safe types are types that either don't hold any data (records and classes without fields) or types that encapsulate various mechanisms to protect their data integrity when an instance can be simultaneously used from multiple threads.

Examples:

- synchronization objects (types) like `System.SyncObjs.TCriticalSection`
- `System.Generics.Collections.TThreadList<T>`

Thread-unsafe data type (Single-thread-safe) - some conditions may apply

Thread-unsafe types are types that can be used in background threads but don't have thread safety built in by default. Their usage is safe only as long as code in multiple threads cannot simultaneously modify their data, or they can only be used in background threads under certain other conditions. I will also refer to such types as *single-thread-safe* types.

In other words, they can be safely used only if they are confined to a single thread, or never accessed from multiple threads simultaneously, or if simultaneous access is protected by some additional, external mechanism.

Examples:

- `System.Classes TStringList`
- `System.Generics.Collections.TList<T>`
- `System.Messaging.TMessageManager`

Thread-unsafe data type - main thread only

Thread-unsafe types and instances that can only be used in the context of the main thread. Functions and other code that directly manipulates such thread-unsafe types and instances can also only be used in the context of the main thread.

Examples:

- `FormatSettings`
- `System.Messaging.TMessageManager.DefaultManager`
- visual controls in VCL and FMX

Standalone functions, procedures, class or record methods can also be categorized in a similar manner, based solely on their inner code, as the thread safety of passed parameters is a separate concern:

Thread-safe function, procedure, or method

Thread-safe functions are those that don't access any unprotected shared data. When calling such functions, you still need to consider the thread safety of the passed parameters, regardless of their type or parameter modifiers.

Examples:

- `System.SysUtils.IntToStr`

Thread-unsafe function, procedure, or method

Functions in this category are basically thread-safe based on their inner code, but this kind of categorization will be used for functions that are commonly used in scenarios where additional caution is necessary because of their manipulation of passed parameters, or where calling them thread-safe might inadvertently imply that the thread safety of their parameters is not relevant.

Examples:

- `System.SysUtils.FreeAndNil`

Thread-unsafe function, procedure, or method - main thread only

Functions that directly access unprotected shared data, and can only be used in the context of the main thread.

Examples:

- `System.SysUtils.Format` - overload without `TFormatSettings` parameter

Complete code examples, where the whole context is clearly defined, have only two categories:

Thread-safe - correct code

Thread-safe, correct code that functions according to the specifications, without any concurrency bugs.

Thread-unsafe - incorrect or dangerous code

Thread-unsafe, incorrect code with concurrency bugs. Also includes functions that have negative impact on thread safety of the whole application, regardless of thread on which they run, and should (must) be avoided.

1.2 General thread safety facts

- only reading data is always thread-safe
- if even one thread writes the data, then any access, even reading, is unsafe
- the thread safety of value type instances depends on access to the variable itself
- the thread safety of reference types instances depends on the thread safety of both the reference variables and the associated data, which are independent from each other
- memory allocation/deallocation depends on the thread safety of the memory manager, and the default manager, FASTMM, is thread-safe
- initiating construction of objects—calling a type's constructor—is thread-safe
- initiating destruction—explicitly calling the destructor `Destroy`, `Free` or similar procedures—is not thread-safe (you need to guarantee that it will be only called once)
- once initiated, the object construction and destruction process is thread-safe if none of the methods invoked in the process use unprotected (unsafe) shared data:
 - `NewInstance`
 - `FreeInstance`
 - constructors
 - destructors
 - `AfterConstruction`
 - `BeforeDestruction`
- additionally, for reference-counted classes, thread-safe construction and destruction also depends on:
 - `_AddRef`
 - `_Release`
- the default implementations of all of the above methods in `TObject`, `TInterfacedObject` and `TThread` are thread-safe
- `TComponent` and its descendant classes are generally not thread-safe
- generally, assignment is not a thread-safe operation
- assignment of naturally aligned value types up to pointer size is atomic
- atomicity does not imply thread safety, it just guarantees data consistency - no tearing
- assignment of reference types is not thread-safe
- weak references are not thread-safe—taking a strong reference from weak one
- reference counting in managed types guarantees that clearing (assigning `nil`, going out of scope) and initiating the destruction of such instances, where each thread already holds a strong reference, will be thread-safe

Note: `TObject`, `TInterfacedObject` and `TThread` are explicitly mentioned as being thread-safe, because they are the most commonly used base classes. Since their construction/destruction implementations are thread-safe, determining whether the construction/destruction of their descendant classes is thread-safe depends solely upon the overrides of the methods involved in the process. That makes it easier to inspect the class hierarchy. If all ancestors of a class are thread-safe, then that class will also be thread-safe if it overrides the construction/destruction methods in a thread-safe manner (or not at all).

Chapter 2

Proving thread safety

Thread safety cannot be proven by testing. The most you can achieve by testing is to prove that some code is *not* thread-safe. Applying more extensive testing and stressing the code can help expose some concurrency issues in a given block of code, but again: If your code does not break under stress, that only means it is *probably* thread-safe. You cannot say it is 100% thread-safe.

Thread safety is an absolute category, and either some code is thread-safe, or it is not. Running some tests and determining that the code is probably thread-safe is not good enough. Ignoring thread safety issues not only has a negative impact on the application's stability, but also on the consistency of the user's data. Since threading issues are also highly dependent on timing, code that passes tests on one machine can easily be broken all the time on another.

The only way to prove some code is thread-safe is to use logic. You need to inspect all the data and code involved and determine their thread safety by logical arguments. You need to know whether and when multiple threads can access any particular data simultaneously and whether such access is thread-safe or not. You must also ensure that the data will always be in a consistent state, and that the code functions according to the specification—that there are no other logical errors. And even when you prove with arguments that code is thread-safe, there is always a chance that you have missed some subtle issue that makes that code unsafe.

Proving the thread safety of some code starts by proving the thread safety of the smallest pieces of code, and working your way up from there. If your small building blocks are thread-safe, and you don't make any thread safety mistakes when combining them together, then the combined code will also be thread-safe.

Regardless of how big the code is, the steps needed to prove thread safety are the same.

Steps:

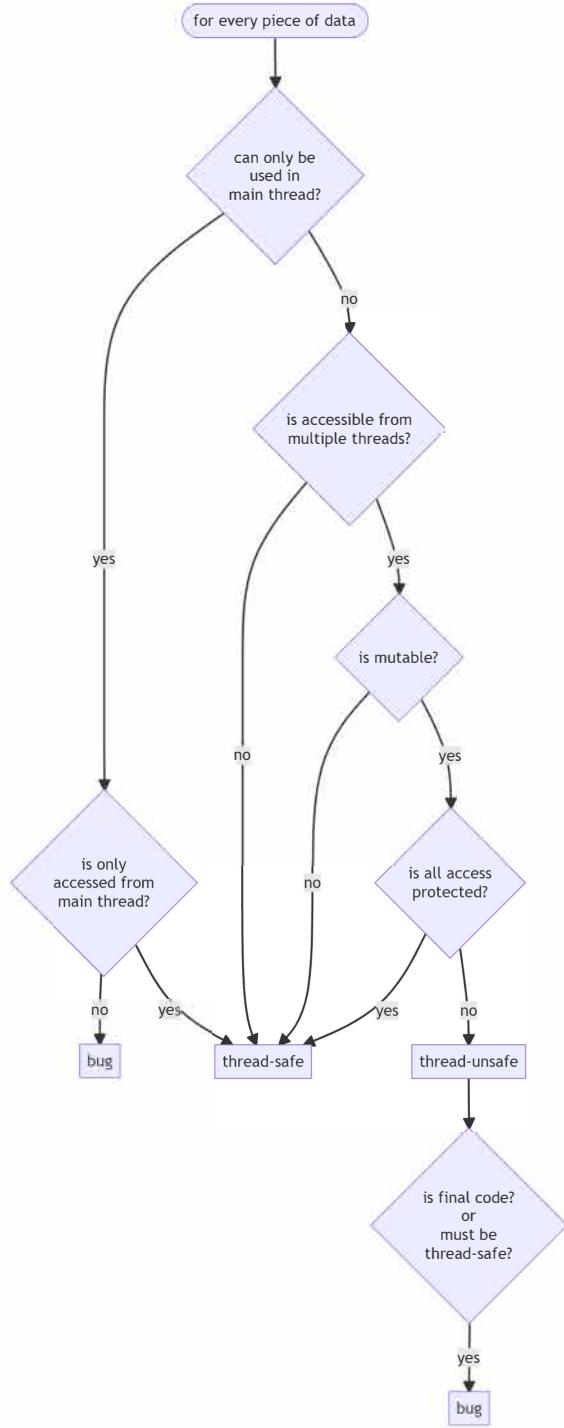
- identify all the data used
- for every piece of data, determine:
 - if the data can only be used in the main thread context, is all access synchronized with the main thread?
 - is the data accessible from multiple threads?
 - if accessible, is it mutable?
 - if mutable, is it protected with some synchronization mechanism?
 - if the data requires a synchronization mechanism, is all access—read or write—protected?
- inspect whether the code as a whole meets the specifications—pay special attention to multiple operations on data that must be executed without interruptions from other threads in order to provide data consistency and desired behavior

If thread safety or functionality is broken in any of the above steps, the code is not thread-safe.

If thread safety arguments are about a specific data type, then being thread-unsafe is not a bug. Rather, it is a categorization telling us that such data either requires additional protection when used in multithreaded scenarios, or is meant to be used only from the context of the main thread.

If the type is meant to be thread-safe (if thread safety is part of the specification), or arguments are being made about some *final*, actual code, then being thread-unsafe represents a bug that requires fixing. Accessing data from a background thread, when it is meant to be used only in the context of the main thread, is also a bug.

Note: In the context of thread safety arguments, the term *final code* means a snapshot of the working code (application) at some point in time, not code that will never change in the future. If any part of that particular code involving data handling changes later on, it needs to be reevaluated again. It is an iterative process, and as the code changes, so does the surrounding context, and thread-safety always depends on the complete context.



Whatever you do, you can never be 100% sure that some code is thread-safe. This is important to remember, because no matter how experienced you are, and how much you know about writing thread-safe code, you always need to be prepared for the possibility that code you thought is thread-safe, is actually not thread-safe at all. This is what ultimately makes multithreading hard: The inability to absolutely trust and prove the thread safety of some code.

In very simple scenarios, the thread safety of some code can be easily reasoned out, and you can usually claim that such simple code is 100% thread-safe. But, real applications are anything but simple, and no matter how sure you are, you should always expect the unexpected.

The ability to make *more correct* thread safety arguments in simple scenarios is crucial for achieving thread safety on a larger scale. Encapsulation and layering is the only viable approach. The more confined your data is, and the less shared data you have lying around in the open where any code can mess with it at any time, the safer your application will be. The larger the context, the harder it is to determine whether all thread safety requirements are actually met.

Chapter 3

Examples

3.1 Data type thread safety

The following examples show the process of determining the thread safety of some data type. In this process, thread safety arguments will strictly concern the data type itself and its public API, not the thread safety of a particular variable of that type. In other words, we will treat such an unspecified variable (value or reference) as read-only, and examine whether we can break the thread safety of such data when simultaneously used in multiple threads.

3.1.1 Example 1. Thread-unsafe class with read/write property

Thread-unsafe data type

```
type
  TNumber = class
  private
    FValue: Int64;
  public
    constructor Create(AValue: Int64);
    property Value: Int64 read FValue write FValue;
  end;

constructor TNumber.Create(AValue: Int64);
begin
  FValue := AValue;
end;
```

Arguments:

- publicly accessible data: the **Value** property, backed by the **FValue** field
- the **Value** property has read and write access to the **FValue** field, and is not protected from simultaneous access by multiple threads

Conclusion:

- the **TNumber** class is not thread-safe, because multiple threads can simultaneously access and modify its data in a thread-unsafe manner

3.1.2 Example 2. Thread-safe class with read-only property

Thread-safe data type

```
type
  TFoo = class
  private
    FValue: Int64;
  public
    constructor Create(AValue: Int64);
    property Value: Int64 read FValue;
  end;

constructor TFoo.Create(AValue: Int64);
begin
  FValue := AValue;
end;
```

Arguments:

- publicly accessible data: the **Value** property, backed by the **FValue** field
- **FValue** can only be set (mutated) by the constructor
- the **Value** property only has read access to the immutable **FValue** field, and cannot be used to modify the contents of the **FValue** field

Conclusion:

- the **TFoo** class is thread-safe because once created, the contents of the instance cannot be mutated, and immutable data is thread safe

3.1.3 Example 3. Thread-safe class with read/write property

Thread-safe data type

```
type
  TSafeFoo = class
private
  FLock: TCriticalSection;
  FValue: Int64;
  procedure SetValue(AValue: Int64);
  function GetValue: Int64;
public
  constructor Create(AValue: Int64);
  destructor Destroy; override;
  property Value: Int64 read GetValue write SetValue;
end;

constructor TSafeFoo.Create(AValue: Int64);
begin
  FLock := TCriticalSection.Create;
  FValue := AValue;
end;

destructor TSafeFoo.Destroy;
begin
  FLock.Free;
  inherited;
end;

procedure TSafeFoo.SetValue(AValue: Int64);
begin
  FLock.Enter;
  try
    FValue := AValue;
  finally
    FLock.Leave;
  end;
end;

function TSafeFoo.GetValue: Int64;
begin
  FLock.Enter;
  try
```

```
    Result := FValue;
  finally
    FLock.Leave;
  end;
end;
```

Arguments:

- publicly accessible data: the `Value` property, backed by the `FValue` field accessed by the `SetValue` and `GetValue` methods
- `FValue` can be set by the constructor and `SetValue` method, and read by `GetValue` method
- read and write access to `FValue` within the `SetValue` and `GetValue` methods is protected with a lock stored in the `FLock` field

Conclusion:

- the `TSafeFoo` class is thread-safe, because all access to its data is protected by a lock, and multiple threads can only access the data one thread at the time

Thread safety in this example does not imply anything more than that the stored value will be in a consistent state—there will be no data tearing. It does not mean that multiple operations performed from one thread will always see the same value, if there is another thread running and modifying the data.

Following the same principle, we can also protect the data consistency in a class that holds multiple values.

3.1.4 Example 4. Thread-unsafe class with read-only property and mutating method

Thread-unsafe data type

```
type
  TBaz = class
  private
    FValue: Int64;
  public
    constructor Create(AValue: Int64);
    procedure Test(AValue: Int64);
    property Value: Int64 read FValue;
  end;
```

```

constructor TBaz.Create(AValue: Int64);
begin
    FValue := AValue;
end;

procedure TBaz.Test(AValue: Int64);
begin
    if FValue > AValue then
        FValue := AValue;
end;

```

Arguments:

- publicly accessible data: the **Value** property, backed by the **FValue** field
- **FValue** can be set (mutated) by the constructor and the **Test** procedure
- the **Value** property only has read access to the immutable **FValue** field, and cannot be used to modify the contents of the **FValue** field
- the **Test** procedure can modify **FValue** in a thread-unsafe manner

Conclusion:

- the **TBaz** class is not thread-safe, because multiple threads can simultaneously access and modify its data in a thread-unsafe manner

3.1.5 Example 5. Thread-safe record with read-only property**Thread-safe data type**

```

type
    TFooRec = record
    private
        FValue: Int64;
    public
        constructor Create(AValue: Int64);
        property Value: Int64 read FValue;
    end;

constructor TFooRec.Create(AValue: Int64);
begin
    FValue := AValue;
end;

```

Arguments:

- publicly accessible data: the **Value** property, backed by the **FValue** field
- **FValue** can only be set (mutated) by the constructor
- the **Value** property only has read access to the immutable **FValue** field, and cannot be used to modify the contents of the **FValue** field

Conclusion:

- the **TFooRec** record is thread-safe because once created, the contents of the instance cannot be mutated, and immutable data is thread-safe

3.1.6 Example 6. Thread-safe class with read-only object property

Thread-safe data type - “kind of”

```

type
  TFooBar = class
    private
      FValue: TFoo;
    public
      constructor Create(AValue: Int64);
      property Value: TFoo read FValue;
    end;

constructor TFooBar.Create(AValue: Int64);
begin
  FValue := TFoo.Create(AValue);
end;

```

Arguments:

- publicly accessible data: the **Value** property, backed by the **FValue** field
- **FValue** can only be set (mutated) by the constructor
- the **Value** property only has read access to the immutable **FValue** field, and cannot be used to modify the contents of the **FValue** field

The above arguments for the **TFooBar** class have a flaw. In other words, **TFooBar**'s thread safety is not foolproof.

Unlike the scenario demonstrated by the **TFoo** class, **TFooBar**'s **Value** property is actually an object reference, and the only thread-safe part here is the reference itself. The **TFoo** class is

immutable, so instance pointed by **FValue** is also immutable. But, **FValue** is an object, and all objects in Delphi have some rather *nasty* methods available, namely **Destroy** and **Free**. We can easily call those on any object reference, and we can easily kill the **FValue** object instance from one thread while another thread is still using it. Not very thread-safe, is it?

Running the following code will show that, at some point, we will be accessing an invalid object from the first thread.

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  FooBar: TFooBar;
begin
  // the FooBar object will leak,
  // but that fact is not relevant for proving unsafety
  FooBar := TFooBar.Create(5);
  TThread.CreateAnonymousThread(
    procedure
    var
      i: Integer;
    begin
      for i := 0 to 100 do
        NxLog.D(FooBar.Value.Value.ToString);
    end).Start;
  TThread.CreateAnonymousThread(
    procedure
    begin
      FooBar.Value.Free;
      NxLog.D('FooBar destroyed');
    end).Start;
end;
```

```
NXDEBUG BT 7544 - 5
NXDEBUG BT 7544 - 5
NXDEBUG BT 7544 - 5
NXDEBUG BT 5652 - FooBar destroyed
NXDEBUG BT 7544 - 5
NXDEBUG BT 7544 - 5
NXDEBUG BT 7544 - 5
```

This code does not crash, because the destroyed object's memory is still intact, but that is just because we were lucky, not because the code is correct.

Correct arguments:

- publicly accessible data: the **Value** property, backed by the **FValue** field

- `FValue` can only be set (mutated) by the constructor
- the `Value` property only has read access to the immutable `FValue` field, and cannot be used to modify the contents of the `FValue` field
- the `Value` property is an object reference, and we can call `Destroy` or `Free` on that reference, which destroys the object -> even though the object itself and its reference are not mutable, the property is not fully thread-safe as long as we can destroy it

Conclusion:

- the `TFooBar` class is not thread-safe because we can destroy the inner object

But this is not the end of the discussion, because things are way more complicated.

First, let's remove the weak point in `TFooBar`'s thread safety—the exposed object property:

```

type
  TFooBarSafe = class
  strict private
    FValue: TFoo;
    function GetValue: Int64;
  public
    constructor Create(AValue: Int64);
    destructor Destroy; override;
    property Value: Int64 read GetValue;
  end;

constructor TFooBarSafe.Create(AValue: Int64);
begin
  FValue := TFoo.Create(AValue);
end;

destructor TFooBarSafe.Destroy;
begin
  FValue.Free;
  inherited;
end;

function TFooBarSafe.GetValue: Int64;
begin
  Result := FValue.Value;
end;

```

By fully encapsulating the inner `TFoo` instance, we have removed the possibility of the code in outer context destroying it. Or have we?

All we need to know in order to break in, is the declaration of the **TFooBarSafe** class. This gives us access to its private fields, and the ability to do whatever we want in a completely thread-unsafe manner:

```
type
  TFooBarHack = class
  public
    FValue: TFoo;
  end;

var
  FooBar: TFooBar;
begin
  FooBar := TFooBar.Create(5);
  TFooBarHack(FooBar).FValue.Free;
end;
```

Now what?

This is a good place to be reminded once again that there is no absolute safety: neither for thread safety, nor anything else. Delphi is a rather safe language, and it will prevent you from casually shooting yourself in the foot in many places, but this safety net only goes so far. There is always a way to make a mess, and eventually developers have to take some responsibility for the code they write.

With similar code, we can also hack into the **TFoo** class, and we can just as easily break into **TFooRec**. If safety is relative, how should we define the **TFooBar** class? As safe or unsafe?

The only way to break **TFooBar**'s safety without explicitly hacking it is to call **Value.Free**. Normally, developers don't go around and call **Free** on random references. The odds that someone will make such a mistake are rather low. Also, in a real-life situation, the **FooBar** instance would eventually be destroyed. That would cause a double free of the inner object instance, causing a crash. And even if someone forgot to free **FooBar**, then that instance would leak, and such a leak would show if you turned on memory leak checking in FASTMM. Which is generally a really good practice.

When all is considered, we can give the green light to **TFooBar**'s thread safety, and say it is a thread-safe class. However, if we were dealing with a class that exposes some otherwise mutable inner object, then we might cast a different judgement. But it all depends on the actual class and the code behind it.

Encapsulating the functionality of the inner object in the case of **TFooBar** is rather simple. Doing so with a more complex class could require a lot of boilerplate code. There is always some tradeoff. In a real-life situation, you have to decide whether some class deserves to be as foolproof as possible, or if you should leave some loopholes open because closing them is not a viable option, while still considering the class in question as thread-safe if used properly—that is, as outlined in your documentation.

Conclusion:

- the `TFooBar` class is thread-safe because its thread safety can be broken only by uncommonly used code

Whether you would categorize `TFooBar` as thread-safe or thread-unsafe is pretty much opinion-based. Since breaking thread safety requires writing fairly uncommon and incorrect code regardless of whether threading is involved, considering this class thread-safe is the more correct option.

From the previous examples, it is rather obvious that the thread safety categorization of data types creates something more like guidelines than rules. No matter how thread-safe a type is, it is always possible to break that safety with some bad code.

Eventually, context is all that matters, and only actual complete code can be fully judged. But some types will always give more protection than others. It is a matter of statistics: If you are using a type with many loopholes, it is more likely that you will fall into one, than with a type that has fewer. But it is also a matter of camouflage. If the loophole is obvious, if it requires unusual or generally poor code, then the odds of anyone writing such code are rather small. On the other hand, if writing regular, day-to-day code, that does not smell from a mile away, is all it takes to stumble upon a loophole, then it is hard to say that such a type is thread-safe. Or, at least, it must have huge warning signs attached, clearly stating what kind of code will open up concurrency issues.

3.2 Thread-safe data types used in an unsafe manner

Even if some data type is thread-safe, it can always be used in a thread-unsafe manner.

Sometimes, thread safety will be broken because the data type has some loopholes, or the code using the data type is applying various hacks.

But sometimes, thread safety will not be broken because the data type itself has loopholes, or the code using it is hacky, but because it is used to achieve some functionality that is not part of the type's initial design and specification. When some data type is used in some code, then the thread safety of that code also depends on the design and specification for that particular piece of code—on its expected behavior and functionality.

In other words, exactly the same code can be thread-safe in one scenario and thread-unsafe in another, depending on the expected functionality.

A common form of thread safety issues while using thread-safe data types arises when combining multiple methods on data to achieve a particular functionality. Even when each method used is thread-safe, their combination will no longer be thread-safe, unless it is additionally protected in the context where they are combined.

3.2.1 Example 7. Thread-safe integer list

Thread-safe data type

```
type
  TIntegerList = class
protected
  FLock: TCriticalSection;
  FItems: array of Integer;
public
  constructor Create;
  destructor Destroy; override;
  procedure Add(Value: Integer);
  function Exists(Value: Integer): Boolean;
  procedure Iterate(const AProc: TProc<Integer>);
end;

constructor TIntegerList.Create;
begin
  FLock := TCriticalSection.Create;
end;

destructor TIntegerList.Destroy;
begin
  FLock.Free;
  inherited;
end;

procedure TIntegerList.Add(Value: Integer);
begin
  FLock.Enter;
  try
    SetLength(FItems, Length(FItems) + 1);
    FItems[High(FItems)] := Value;
  finally
    FLock.Leave;
  end;
end;

function TIntegerList.Exists(Value: Integer): Boolean;
var
  X: Integer;
begin
```

```
Result := False;
FLock.Enter;
try
  for X in FItems do
    if X = Value then
      begin
        Result := True;
        break;
      end;
  finally
    FLock.Leave;
  end;
end;

procedure TIntegerList.Iterate(const AProc: TProc<Integer>);
var
  X: Integer;
begin
  FLock.Enter;
  try
    for X in FItems do
      AProc(X);
  finally
    FLock.Leave;
  end;
end;
```

Arguments:

- **FItems** is a private field and is not directly accessible
- all methods that access **FItems** use a locking mechanism through the private **FLock** field to protect **FItems** from simultaneous access from multiple threads

Conclusion:

- the **TIntegerList** class is thread-safe

3.2.2 Example 8. Thread-safe usage of thread-safe data type

Thread-safe usage of thread-safe data type

Code specification:

An integer list needs to be populated with unique numbers in the range of **0** to **20** on one thread. Another thread needs to repeatedly iterate through the list and log the numbers added so far, until all numbers are added to the list.

```
var
  List: TIntegerList;
begin
  List := TIntegerList.Create;
  TThread.CreateAnonymousThread(
    procedure
    var
      i: Integer;
    begin
      for i := 0 to 20 do
        begin
          Sleep(2);
          List.Add(i);
        end;
    end).Start;

  TThread.CreateAnonymousThread(
    procedure
    begin
      repeat
        List.Iterate(
          procedure(Value: Integer)
          begin
            NxLog.D(Value.ToString);
          end);
        Sleep(2);
        // check if last number is added to the list
        if List.Exists(20) then
          Break;
      until False;
      List.Free;
    end).Start;
end;
```

Arguments:

- `List` is a local variable, assigned only once, before secondary threads have access to it
- adding numbers to the list in the first thread is done by the thread-safe `Add` method
- the repeat loop in second thread uses the thread-safe `Iterate` method and cannot interfere with adding numbers in first thread
- the repeat loop will exit only when the last number is added to the list - that check is done through the thread-safe `Exists` method
- there is a possibility that the last number(s) in the list will be added between the `Iterate` and `if List.Exists()` lines, but our code specification does not require that all added numbers must be written into log at some point, so the code will run correctly as specified
- `List` is released in the second thread, but only after all numbers are added to the `List`, and after that point, no code in the first thread will use `List`

Conclusion:

- the example code is thread-safe and functional according to the specification

3.2.3 Example 9. Thread-safe usage of thread-safe data type

Thread-safe usage of thread-safe data type

Code specification:

An integer list needs to be populated with unique numbers in the range of `0` to `20` in one thread. Another thread needs to repeatedly iterate through the list and log the numbers added so far, until all numbers are added to the list. All numbers must be logged at least once.

```
var
  List: TIntegerList;
begin
  List := TIntegerList.Create;
  TThread.CreateAnonymousThread(
    procedure
    var
      i: Integer;
    begin
      for i := 0 to 20 do
        begin
          Sleep(2);
          List.Add(i);
        end;
    end).Start;
```

```
TThread.CreateAnonymousThread(
  procedure
  begin
    while not List.Exists(20) do
      begin
        List.Iterate(
          procedure(Value: Integer)
          begin
            NxLog.D(Value.ToString);
          end);
        Sleep(2);
      end;
    List.Iterate(
      procedure(Value: Integer)
      begin
        NxLog.D(Value.ToString);
      end);
    List.Free;
  end).Start;
end;
```

Arguments:

- data shared between threads: **List**
- **List** is a local variable, assigned only once, before secondary threads have access to it
- adding numbers to the list in first thread is done by the thread-safe **Add** method
- the while loop in the second thread uses the thread-safe **Iterate** method and cannot interfere with adding numbers in the first thread
- the while loop in the second thread will exit only when the last number is added to the list - that check is done through the thread-safe **Exists** method
- since we need to log all numbers at least once, there is an additional **Iterate** call after the while loop
- **List** is released in the second thread, but only after all numbers are added to the **List**, and after that point, no code in the first thread will use **List**

Conclusion:

- the example code is thread-safe and functional according to the specification

3.2.4 Example 10. Thread-unsafe usage of thread-safe data type

Thread-unsafe usage of thread-safe data type

Code specification:

An integer list needs to be populated with unique numbers in the range of **0** to **20** using multiple threads. After all numbers are added, the contents of the list should be logged.

```
procedure Numbers;
var
  List: TIntegerList;
begin
  List := TIntegerList.Create;
  TThread.CreateAnonymousThread(
    procedure
    var
      i: Integer;
    begin
      Sleep(1);
      for i := 0 to 20 do
        begin
          if not List.Exists(i) then
            List.Add(i);
        end;
    end).Start;

  TThread.CreateAnonymousThread(
    procedure
    var
      i: integer;
    begin
      Sleep(1);
      repeat
        for i := 0 to 19 do
          begin
            if not List.Exists(i) then
              List.Add(i);
            end;
        // check if last number is added to the list
        if List.Exists(20) then
          Break;
      until False;
      i := 0;
```

```

List.Iterate(
  procedure(Value: Integer)
  begin
    Inc(i);
    NxLog.D(Value.ToString());
  end);
  if i > 21 then
    NxLog.D('Duplicates');
  List.Free;
end).Start;
end;

```

Arguments:

- Two threads are adding numbers to `List`. The code used for adding unique numbers uses the thread-safe `Exists` and `Add` methods. But it is possible that both threads pass the `if not List.Exists(i)` check for the same value of `i`, and add the same number to the list twice.

Conclusion:

- The code in this example is not thread-safe, because the threads can interfere in such a manner that the code will not run according to the specification. It is possible that the list will contain duplicate numbers, while the specification requires that the numbers in the list must be unique.

Thread-safe, correct code

```

procedure Numbers;
var
  List: TIntegerList;
  Lock: TCriticalSection;
begin
  Lock := TCriticalSection.Create;
  List := TIntegerList.Create;
  TThread.CreateAnonymousThread(
    procedure
    var
      i: Integer;
    begin
      Sleep(1);
      for i := 0 to 20 do
        begin

```

```
Lock.Enter;
try
  if not List.Exists(i) then
    List.Add(i);
finally
  Lock.Leave;
end;
end;
end).Start;

TThread.CreateAnonymousThread(
procedure
var
  i: integer;
begin
  Sleep(1);
  repeat
    for i := 0 to 19 do
      begin
        Lock.Enter;
        try
          if not List.Exists(i) then
            List.Add(i);
        finally
          Lock.Leave;
        end;
      end;
    // check if last number is added to the list
    if List.Exists(20) then
      Break;
  until False;
  i := 0;
  List.Iterate(
    procedure(Value: Integer)
    begin
      Inc(i);
      NxLog.D(Value.ToString());
    end);
  if i > 21 then
    NxLog.D('Duplicates');
  List.Free;
  Lock.Free;
end).Start;
end;
```

Arguments:

- Data shared between threads: **List**, **Lock**
- **List** is a local variable, assigned only once, before secondary threads have access to it
- **Lock** is a local variable, assigned only once, before secondary threads have access to it
- **Lock** is used as a synchronization mechanism inside the loops in both the first and second thread, and protects the code that checks whether the number exists before adding it to the list
- **List** and **Lock** are released in the second thread, but only after the last number has been added to the list, and at that point, the first thread no longer uses **List** nor **Lock**. For this argument, it is critical that the loop in the second thread only goes to **19**, otherwise it would be possible (even though chances are slim) that the second thread would be the one that will add the last number **20**, and could release **List** and **Lock** while the first thread is still in the loop where **List** and **Lock** are used. This is the kind of subtle logical problem that can cause rare bugs in multithreaded code.

Conclusion:

- the code in this example is thread-safe, and runs according to the specification.

3.3 Code thread safety

Examining the thread safety of some parts of code is quite similar to examining the thread safety of data types. Just because the data and code is not organized in a class or record, you are still judging the thread safety of data accessible through some particular code.

The main difference is that the context of the code you need to explore and take into account will commonly be much broader than the context you will consider when assessing data type thread safety. While the process of determining thread safety in all contexts is the same, the broader context makes it harder to do it accurately.

Reducing shared data, and narrowing the context can significantly simplify determining the thread safety of some code. After all, what is a class, if not encapsulated data with associated code? How thread-safe a class is also depends on how well its data is encapsulated and protected. Just because some data does not necessarily belong to a class does not mean it cannot be confined to a much narrower scope.

The *Thread-safe data types used in an unsafe manner* subchapter already covered the process of making code safety arguments on a smaller scale, namely in the scope of a procedure.

There is an additional concern when considering the thread safety of procedures and methods that has not been covered in the previous examples—the thread safety of passed parameters. The thread safety of parameters is a separate concern from the thread safety of the code within the procedures or methods. When assessing the thread safety of individual procedures and methods, parameters belong to an external, unspecified context, and we can only judge the

safety of the known context within. Similarly to determining the thread-safety of data types, we will assume that the passed parameters are read-only in the outside context.

If they are not thread-safe in the broader context, then that thread-unsafe will also bleed into the code we are judging. More about parameters can be found in the *FreeAndNil* chapter.

3.3.1 Example 11. Thread-safe data handover between threads

One of the common thread-safety coding patterns includes constructing an otherwise single thread-safe instance and handing it over to another thread. This kind of usage is thread-safe without the need for any additional protection mechanisms because threads involved don't simultaneously access the data. There is some discipline involved when writing such code, because the developer needs to pay attention and not use the data after the handover, but this is not any harder to achieve than not using an object instance after it has been released.

Thread-safe

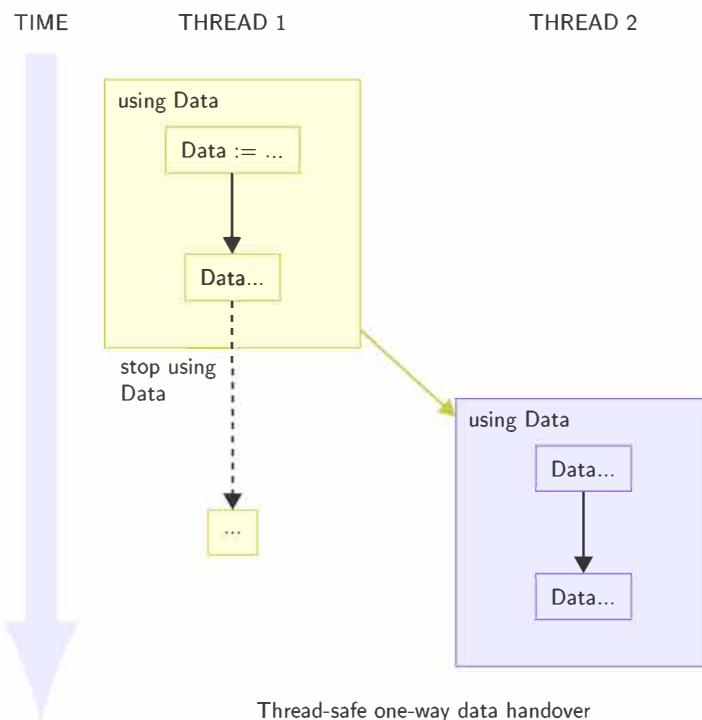
```
var
  sl: TStringList;
begin
  sl := TStringList.Create;
  try
    sl.Add('abc');
    sl.Add('123');
    TThread.CreateAnonymousThread(
      procedure
        var
          s: string;
        begin
          try
            sl.Add('456');
            for s in sl do
              NxLog.D(s);
          finally
            sl.Free;
          end;
        end).Start;
  except
    sl.Free;
    raise;
  end;
end;
```

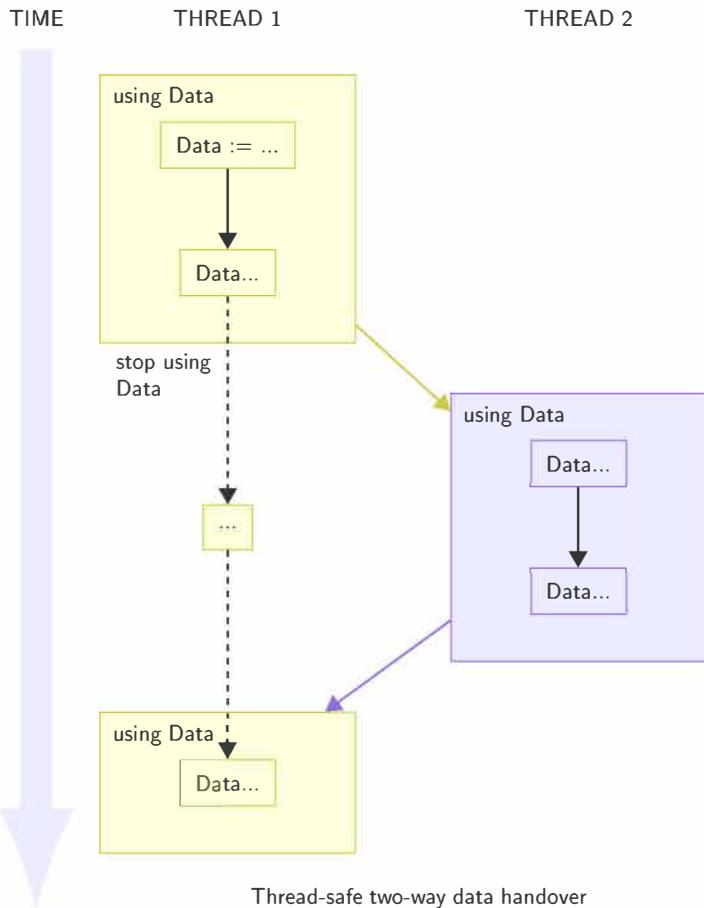
Arguments:

- data shared between threads: `sl`
- `sl` is constructed and used in original thread before another (anonymous) thread that uses it is started
- `sl` usage in original thread is protected by `try...except` block and if any exception happens in that part of the code, code in the `except` block will run and `sl` instance memory will be released and exception re-raised for further handling—we don't have to re-raise exception here and we can also completely handle it there
- if there are no exceptions raised in that part of the code that means secondary thread has been successfully started and at that point, while secondary thread uses captured `sl` instance original thread is no longer using it
- after code in secondary thread is finished, `sl` memory will be released
- to prevent memory leaks in case of any exception in secondary thread, that code is protected with `try...finally` block

Conclusion:

- the code in this example is thread-safe





Safety of the unprotected data handover does not depend on how the handover is done: via anonymous method capture, data passed as a parameter..., and what kind of threads are involved in the process: anonymous threads, custom threads, tasks.... The only thing that matters is that data must not be touched in the original thread after it is handed out to another thread and while that thread is using the data.

If a secondary thread can notify the original thread it is no longer working with the data, for instance in thread **OnTerminate** event, if thread has been waited for, or with any other similar mechanism, the original thread can safely use the data again from that point on. If particular code does not use any such notification mechanism and cannot determine when secondary thread finished using the data, then the original thread cannot start using the data again.

3.3.2 Example 12. Thread-unsafe data handover between threads

This example shows incorrect, thread-unsafe data handover. Data is being used after the handover to another thread and this opens possibility for unintended interaction.

Thread-unsafe - incorrect code

```
var
  sl: TStringList;
begin
  sl := TStringList.Create;
  try
    sl.Add('abc');
    sl.Add('123');
    TThread.CreateAnonymousThread(
      procedure
        var
          s: string;
        begin
          try
            sl.Add('456');
            for s in sl do
              NxLog.D(s);
          finally
            sl.Free;
          end;
        end).Start;
    // any usage of sl in this block is not thread-safe
    // ---- BEGIN ----
    sl.Add('777');
    // ---- END ----
  except
    sl.Free;
    raise;
  end;
end;
```

Arguments:

- data shared between threads: `sl`
- adding string `777` to the `sl` happens after anonymous thread has started running and it opens possibility that two threads simultaneously use `sl` instance and `TStringList` is not thread-safe type that allows unprotected, simultaneous use of the same instance

Conclusion:

- the code in this example is not thread-safe

Part 2. The Core Run-Time Library

Chapter 4

Global state

Mutable shared data (state) is the enemy of thread safety. Any data that can be changed while being accessed by multiple threads is not thread-safe, unless explicitly protected by some general synchronization mechanism. And global state clearly wins the unsafety contest. So when determining the thread safety of the Delphi RTL, the first thing to look for would be unprotected, mutable global state.

The first one that comes to mind, one that has already caused numerous issues, is the global `FormatSettings` variable, previously divided into several global variables. But before we dive into the thread safety of `FormatSettings` and various related formatting routines, there are more devious and dangerous globals that unfortunately affect the thread safety of all floating-point operations, and also have direct impact on some formatting routines that deal with floating-point numbers.

The thread-unsafe manner in which the Delphi RTL handles floating-point control registers is not well known, and as such, is quite often ignored. If the issue does not happen often enough, or it is hard to reproduce—and by their very nature, threading issues often present themselves as spurious errors in seemingly unrelated code—when the issue does happen, finding the root cause can be hard.

Concurrency issues happen when two threads simultaneously access unprotected data, and often when some data is categorized as thread-unsafe, it is just another way of saying that that data should be only used from the context of the main thread. In other words, the specification for that piece of data only allows usage from the main thread. Usually, the solution is to either change your code to completely avoid access from background threads, or run such pieces of code in the context of the main thread using the `TThread.Synchronize` or `TThread.Queue` methods.

There are other situations where the issue is not caused by *wrong* usage from external code, but by a bug in the framework, where something that should be thread-safe is not. Such issues may be hard or impossible to avoid without fixing the original bug.

In the context of the RTL, the `FormatSettings` issue belongs to the former, and the floating-point control register issue to the latter category.

Chapter 5

Floating-point control register

A floating-point unit is a part of the CPU (or an additional unit) dedicated to operating on floating-point numbers, and as such, it consists of floating-point data registers, a floating-point status register, and a floating-point control register—or FPCR. The FPCR defines floating-point exception masks, precision, and rounding modes.

Since the FPCR is part of the FPU, during the thread context switch, the current value stored in the FPCR is preserved on the OS level—just like the rest of the CPU and FPU state—and restored when the thread continues, meaning that different threads can operate with different FPCR settings.

In Delphi, when an application is started, as well as when a new thread starts, the `_FPUInit` procedure from the `System` unit is called to clear FPU status and initialize the FPU control register to the default value stored in several global variables across platforms and CPU architectures: `DefaultFPSCR`, `Default8087CW`, and `DefaultMXCSR`.

If you need to change the default value of the FPCR, changing the global variable is not enough. The desired value needs to be applied to the floating-point control register in the FPU, too. There are several global procedures for doing that, and all of them eventually call one of the following procedures, depending on the platform and CPU architecture:

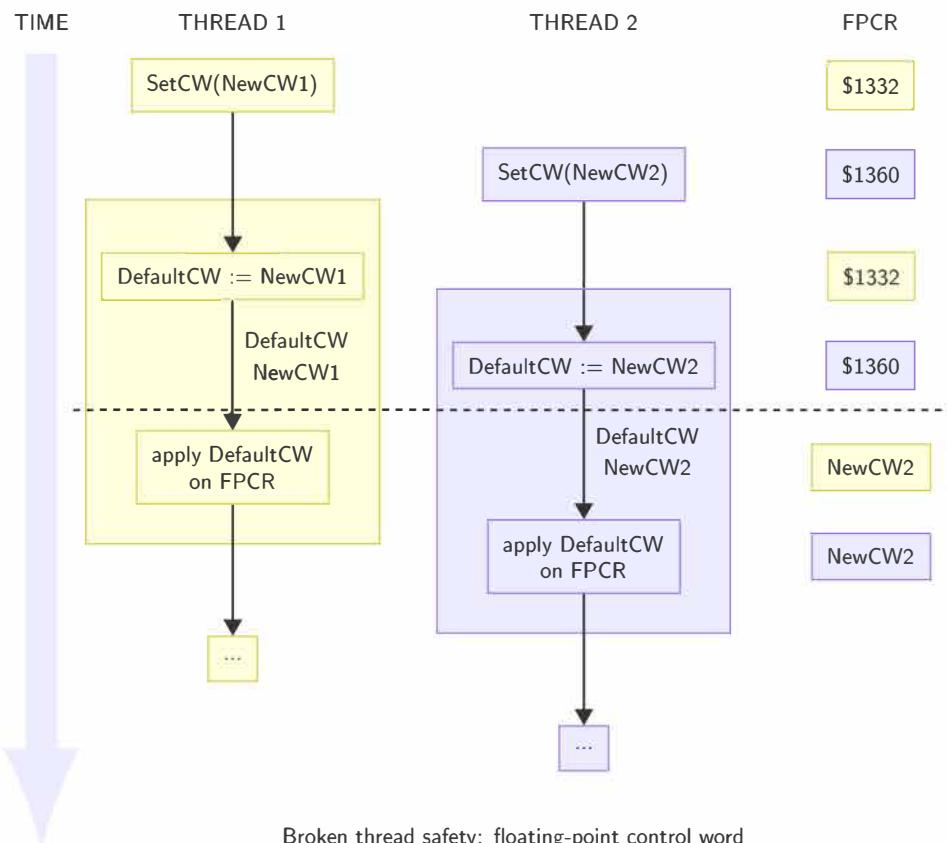
- `Set8087CW` - sets the control register for the Intel x87 FPU
- `SetMXCSR` - sets the MXCSR control register for the Intel SSE
- `SetFPSCR` - sets the control register for the ARM

Because different code is involved on different platforms, the behavior differs from platform to platform, and so do the issues. Unfortunately, some of those `SetXXX` procedures are not thread-safe. That affects all code that calls them, but it also impacts code using the default values in any other way, including the `_FPUInit` procedure that is called as part of every thread initialization, and in various procedures for exception support implemented in the `System` unit.

The following is a code snippet from the `Set8087CW` procedure. The other unsafe procedures follow the same pattern: setting the default global variable first, and then applying the value of the default variable to the floating-point control register on the FPU.

```
procedure Set8087CW(NewCW: Word);
begin
  Default8087CW := NewCW;
  asm
    FNCLEX          // Clear all floating-point exception flags
    FLDCW Default8087CW // Load (apply) x87 FPU control word
  end;
end;
```

And the following diagram shows how above code works in practice, and how two threads changing FPCR at the same time can interfere with each other.



But the problem is not merely in setting the new FPU value. Part of it also comes from the fact that calling any of those thread-unsafe procedures to temporarily change a thread's FPCR value also changes the default value, impacting the behavior of threads spawned in the meantime.

There are a number of procedures and functions across the RTL that call the thread-unsafe **SetXXX** FPCR procedures, and any code using them is also thread-unsafe, but before we start exploring the impact of this thread-unsafe FPCR handling and possible solutions, let's prove with a small example that changing FPCR values in one thread can actually *leak* and cause unintended changes to that FPCR value in another thread. **SetExceptionMask** is used in this particular example, but you can use a similar approach to test other thread-unsafe functions.

SetExceptionMask masks specific exceptions for floating point operations and changes how particular code behaves. For instance if you mask the division-by-zero exception, then such exceptions will not be raised at runtime when division by zero is encountered. On the other hand, if division by zero is unmasked, then division by zero will cause an exception at runtime.

SetExceptionMask is a rather simple procedure, that reads the current state of the FPCR, applies the new exception mask, and calls the appropriate **SetXXX** FPCR procedure.

This example clearly shows what kind of unexpected behavior you can get in your code. It can either cause unexpected exceptions—when your code masks particular exceptions and another thread unmasks those—or you can get bad data when you were expecting an exception, but none was raised.

The first thread masks only precision exceptions, and if you divide by zero, the normal code flow would be raising an exception and landing in the **except** part of the **try...except** block. The second thread will mask divide-by-zero exceptions, and the expected behavior is that it will never raise an exception in such case and will never end up in the **except** part of the **try...except** handler.

Thread-unsafe - broken code

```
procedure BrokenExceptionMask;
var
  OK: Boolean;
begin
  OK := True;

  TThread.CreateAnonymousThread(
    procedure
    var
      i: integer;
      f, d: Double;
      m: TArithmeticExceptionMask;
    begin
      i := 0;
```

```
d := 0;
while OK do
begin
  Inc(i);
  m := SetExceptionMask([TAриthmeticException.exPrecision]);
  try
    f := 10 / d;
    // if it ends here - broken
    OK := False;
    Writeln('BROKEN FLOAT 1 ' + i.ToString());
  except
  end;
end;
end)
.Start;

TThread.CreateAnonymousThread(
procedure
var
  i: integer;
  f, d: Double;
  m: TAриthmeticExceptionMask;
begin
  i := 0;
  d := 0;
  while OK do
  begin
    Inc(i);
    m := SetExceptionMask([TAриthmeticException.exZeroDivide]);
    try
      f := 10 / d;
    except
      // if it ends here - broken
      OK := False;
      Writeln('BROKEN FLOAT 2 ' + i.ToString());
    end;
  end;
end)
.Start;

while OK do;
end;
```

While the previous example shows an unintended interaction between two background threads, similar issues can happen with code running in the main thread and a single background thread.

Which thread will get broken, and at which iteration, is random and can change between runs.

```
procedure BrokenExceptionMaskMain;
var
  OK: Boolean;
  i: integer;
  f, d: Double;
begin
  OK := True;
  i := 0;
  d := 0;

  TThread.CreateAnonymousThread(
    procedure
    var
      i: integer;
      f, d: Double;
    begin
      i := 0;
      d := 0;
      while OK do
        try
          Inc(i);
          SetExceptionMask([TArithmeticException.exPrecision]);
          f := 10 / d;
          // if it ends here - broken
          OK := False;
          Writeln('BROKEN FLOAT 1 ' + i.ToString());
        except
        end;
      end)
    .Start;

  while OK do
    try
      Inc(i);
      SetExceptionMask([TArithmeticException.exZeroDivide]);
      f := 10 / d;
    except
      // if it ends here - broken
      OK := False;
      Writeln('BROKEN FLOAT MAIN ' + i.ToString());
    end;
  end;
```

5.1 FPCR-related thread-unsafe API

Because different platforms have slightly different ways of handling the FPCR, they don't all exhibit the issue nor exhibit the issue in the same way. The oldest well-known report of the issue was filed by David Heffernan in Quality Central in 2012, as report #106943 covering the Windows 32-bit and Windows 64-bit platforms.

Obviously the issue is not new, and it exists in older Delphi versions (can be found in Delphi 7) and is present in the most widely used platforms, so one would expect applications to be falling apart left and right because of this issue.

One of the important factors in this issue is whether calls to various the `SetXXX` procedures that happen at runtime actually change the existing default FPCR value—if the passed parameter is the same as the default value, then nothing bad will happen since assignment to those global integers is atomic, and if there is no change in value, then there is no negative impact on behavior.

If the application uses the same FPCR settings across all threads all the time, or if it modifies the defaults before additional threads start running, then there is no problem. Because of that, not all applications are equally affected by this issue, depending on the code they use. And even if the application has code that can cause a problem, we need to keep in mind this is a concurrency issue, and it is not enough just to have some code that calls problematic procedures—you also need to have multiple threads running that will interleave in a very specific, extremely short timeframe.

If the issue does not occur often enough in the affected applications, it is quite possible that the issue was never even recognized as such and its origin found. Maybe there was an occasional glitch here and there, that was attributed to cosmic rays, some butterfly flapping its wings, or some other random event.

Thread-unsafe base FPCR API

- `System.Default8087CW`, `System.DefaultMXCSR`, `System.DefaultFPSCR`
- `System.Set8087CW`, `System.SetMXCSR`

5.1.1 Posix platforms

As previously mentioned, different platforms have different code and are affected by this issue in different ways. While directly modifying global variables can have an impact on all platforms, Posix has significant differences, as the only unsafe part are global variables that are directly exposed.

All Posix platforms use LIBM—the standard C library of basic mathematical functions—which is thread-safe. Posix platforms running on ARM processors use `SetFPSCR`, but this procedure is thread-safe and it doesn't change the default global variable, it only applies the new value to

the floating-point control register. So you can only get in trouble if you are manually modifying the value of the global `DefaultFPSCR` variable.

The only exceptions are the 32-bit iOS simulator and 32-bit macOS, which are using the thread-unsafe `Set8087CW` procedure, but those platforms have been deprecated for quite some time now.

5.1.2 Windows platforms

When it comes to Windows platforms, the core FPCR API is thread-unsafe on both 32-bit and 64-bit Windows. Any code calling any of those procedures or accessing global variables is also thread-unsafe. When it comes to such code in Delphi frameworks, there are some differences between 32-bit and 64-bit Windows platforms, because the platform implementations differ, and not all code handles the FPCR in thread-unsafe way.

Even though none of the listed APIs are thread-safe, some of them are *less* broken and will run correctly if application uses a single FPCR global value during runtime—if it doesn’t change `Default8087CW` or `DefaultMXCSR` after background threads start running. On the other hand, some of the functionality that is used to explicitly set the FPCR flags, and the API that temporarily changes FPCR values in a thread-unsafe way, are completely broken, and if called in an application while multiple threads are running, it can cause issues.

Thread-unsafe API that can run correctly if FPCR globals don’t change

- System
 - `Reset8087CW`, `ResetMXCSR`
 - `_FPUIInit`
 - `ThreadWrapper`
 - `_HandleAnyException`, `_HandleAutoException`, `_ExceptionHandler`, ...
- Windows 32-bit
 - `Winapi.Windows: CreateWindowXXX`

Thread-unsafe API that changes FPCR in a thread-unsafe manner

- System
 - `FSetRound`
- Other
 - `System.Math: SetExceptionMask`, `SetRoundMode`, `SetPrecisionMode`

- Windows 32-bit
 - `System.Win.ComObj`: `CreateComObject`, `CreateRemoteComObject`, `CreateOleObject`
- Windows 64-bit:
 - `System.SysUtils`: `InternalTextToExtended`, `InternalTextToCurrency` - pure Pascal variants, `TextToFloat`, `StrToFloat`, `StrToCurr`, `StrToBool`...

There are also some APIs on Windows platforms that are not thread-safe in their pure Pascal variants, but by default, the RTL uses thread-safe assembly code. Some of those are: `_Trunc`, `Int`, `Frac`, `SafeLoadLibrary`.

Note: The list of broken APIs due to FPCR handling unsafety depends on platforms and Delphi versions. It is possible that some of the listed functions are or will be fixed independently from the complete fix for this problem. Because the issue is so widespread, it is also impossible to list all APIs that might at some point call some of the thread-unsafe code, so you should treat the above list as non-exhaustive.

As you can see, the list of code impacted by the FPCR issue is huge. It is almost impossible to avoid the issue, especially if you are integrating with libraries that require different FPCR settings than Delphi defaults; working with various graphic frameworks like Direct2D, GDI+ or DirectX; or interacting with COM on 32-bit Windows or doing basically anything on 64-bit Windows that involves converting strings to floats, like serialization and databases.

When some code is not thread-safe, there are several options for fixing such code:

- run particular code only in the context of the main thread
 - not viable, because changing the FPCR needs to be applied on the FPU, which will be preserved during the context switch, any thread-safety-breaking API must be executed in the context of the thread that requires changing the FPCR
- protect access to shared data
 - blocking literally all threads while some thread is calling the affected FPCR API—defies the purpose of using multiple threads
- avoid calling thread-unsafe code
 - good luck with that—especially on 64-bit Windows—the issue is more constrained on the 32-bit Windows platform, and avoiding APIs that change the FPCR at runtime might be possible for some applications
- fix the thread-unsafe code at the origin

To be honest, I explicitly wrote the first three options not to show them as potential solutions, but to emphasize that the last one is the only viable solution for this issue. It simply requires fixing this issue at its origin, which unfortunately means patching the RTL.

5.2 A thread-safe design for FPCR management

In his QC report, David Heffernan provided design principles for FPCR management that would avoid the FPCR thread safety issue:

1. The user should be able to control the FPCR that is applied to a newly created thread. The Delphi RTL would perform the assignment of the FPCR to a value specified by the user.
2. RTL functions that modify the FPCR should be threadsafe and only have influence on the behaviour of the thread that calls the function.
3. The only RTL functions that result in a modification of the FPCR (i.e. value on return differs from value on entry) are those that are explicitly intended to perform the task of modifying the FPCR. In the current RTL that is functions like `Set8087CW`, `SetMXCSR`, `SetRoundMode`, `SetPrecisionMode` etc.
4. The FPCR should not be modified when an exception is raised.

If the RTL was modified to follow these principles, then the FPCR would be in the full and complete control of the programmer. Each thread would be isolated from other threads, and the only way for the FPCR to be modified would be if the programmer explicitly modified it. The FPCR for each thread would have a well-defined value when the thread execution passed into the control of the programmer.

These principles could be followed by making the following changes to the RTL:

- Retain the global variables `Default8087CW` and `DefaultMXCSR`, but change the way they are used. They would be used solely to initialise the FPCR in the top-level Delphi thread procedure, i.e. `ThreadWrapper`.
- Change `Set8087CW` and `SetMXCSR` to set the FPCR to whatever value was passed to those functions and do nothing more. Stop modifying `Default8087CW/DefaultMXCSR`.
- Modify the exception handling functions `_HandleAnyException`, `_HandleOnException`, `_HandleAutoException`, `_ExceptionHandler` and `_DelphiExceptionHandler`. These should no longer modify the FPCR. Obviously these routines need to clear the 8087 data register and clear all FP error flags. But they should preserve the value of the FPCRs.
- The previous bullet point would require a review of all RTL functions that temporarily modify the FPCR values. Because the exception handling functions would no longer reset the FPCR to a default value, any RTL functions that temporarily modify the FPCR need to ensure that they restore the FPCR to the value on entry even in case an exception is raised.

If we ignore the fact that fixing the issue requires recompiling the RTL, a fix following the above principles is not that complicated, though it may require extensive sweeps through the RTL code to find all code that requires fixing. The most significant changes are confined to the **System** unit, and once you fix that, most of the other code should be automatically fixed, too.

The core of the FPCR issue is in the **Set8087CW** and **SetMXCSR** procedures, and they need to be rewritten to stop modifying global values. Please note that the following examples are the simplest code that solves the issue, and that using some more specific compiler directives and code paths may require slightly different code:

```
procedure Set8087CW(NewCW: Word);
var
  CW: Word;
asm
  MOV CW, AX
  FNCLEX // don't raise pending exceptions enabled by the new flags
  FLDCW CW
end;
```

```
procedure SetMXCSR(NewMXCSR: LongWord);
var
  MXCSR: LongWord;
asm
  AND EAX, $FFC0 // Remove flag bits
  MOV MXCSR, EAX
  LDMXCSR MXCSR
end;
```

Once you have fixed the **Set8087CW** and **SetMXCSR** functions, all other code that calls them will automatically be thread-safe, as it will no longer be able to impact other currently-existing threads, nor change default values that can have an impact on newly spawned threads.

The next step would be looking for direct usage of the **Default8087CW** and **DefaultMXCSR** globals, and instead using **Get8087CW** and **GetMXCSR** to read the current value directly from the floating point control register. The only place where globals should be used is in **ThreadWrapper**, which initializes the FPCR for new threads, and in the **System** unit's initialization section—this is done by calling **_FpuInit**.

And finally, the last step is modifying exception handlers in **System** to avoid changing FPCR settings, and ensuring that all functions that temporarily change the FPCR restore its original value.

For exception handlers, we cannot use **_FpuInit**, and we need an additional procedure that will handle clearing the FPU state without modifying the FPCR flags—**_FpuClear**.

For Windows platforms, **_FpuClear** needs to clear the 8087 registers and exception status flags, as well as the MXCSR exception status flags.

```
procedure _FpuClear;
asm
  FNSTCW  [ESP-$02]
  FNINIT
  FLDCW   [ESP-$02]
{$IF Defined(CPUX86)}
  CMP     System.TestSSE, 0
  JE      @Exit
{$ENDIF}
  STMXCSR [ESP-$04]
  AND    [ESP-$04], $FFC0 // Remove flag bits
  LDMXCSR [ESP-$04]
@Exit:
end;
```

On Posix platforms, the **System** unit implements the **FClearExcept** procedure:

```
procedure _FpuClear;
begin
  FClearExcept;
end;
```

If you want to use LIBM directly, you can call **feclearexcept**, but in that case, you will need to import that external function as it is not imported by default.

```
procedure _FpuClear;
begin
  feclearexcept(feeALLEXCEPT); // FE_ALL_EXCEPT
end;
```

Once you have **_FpuClear**, you can simply replace calls to **_FpuInit** from the exception handlers in the **System** unit with calls to **_FpuClear**.

You might want to write additional **SetDefault8087CW** and **SetDefaultMXCSR** procedures that can be used when you need to change the default values used for thread initialization, instead of modifying the default variables directly. The one thing that needs to be taken into account is that when applying the value to the FPCR, the passed parameter should be applied, and not the global. This is more of a precautionary measure, since adjusting defaults is not something you should be doing once threads start running anyway.

```
procedure SetDefault8087CW(NewCW: Word);
var
  CW: Word;
asm
  MOV CW, AX
  MOV Default8087CW, AX
  FNCLEX // Don't raise pending exceptions enabled by the new flags
  FLDCW CW
end;
```

```
procedure SetDefaultMXCSR(NewMXCSR: LongWord);
var
  MXCSR: LongWord;
asm
  AND     EAX, $FFC0 // Remove flag bits
  MOV     MXCSR, EAX
  MOV     DefaultMXCSR, EAX
  LDMXCSR MXCSR
end;
```

Chapter 6

FormatSettings and formatting routines

If, for a moment, we ignore thread safety issues caused by the unsafety of floating-point control registers, we can focus on inspecting the thread safety of the `FormatSettings` global, which is most likely the most abused global (or at least the most known global) setting in Delphi. It should be mentioned that the [TFormatSettings](#) page is also one of the rare places in the Delphi documentation where thread safety and unsafety are actually documented.

`TFormatSettings` is a record holding various locale settings used by various formatting and conversion functions. All fields in the record are mutable. Since its fields are mutable, any given instance of `TFormatSettings` is thread-unsafe by default - it is only thread-safe if none of the code running in your various threads changes the values stored in that particular record instance. The only exceptions to that rule are instances declared as non-writeable constants.

The globally accessible `FormatSettings` variable is initialized with the user's default locale settings. In theory, if every thread in the application only used that record for reading, then you could say that using the `FormatSettings` global is thread-safe, as it will always contain the local user settings.

In practice, such mutable global settings are generally considered to be thread-unsafe and should only be used from the context of the main thread, because any piece of code, including libraries beyond your control, has the ability to change such data and wreak havoc.

And this is exactly what was happening with `FormatSettings`. While using local formatting settings is great for showing appropriately formatted data to the user, it is not so great for data storage and exchange. For instance, if you are exchanging floating-point numbers in textual form, the decimal delimiter will be fixed, and possibly different from the user's locale setting. For JSON, and many other formats, that will be a period—`.`—and for users that use a comma—`,`—as a decimal delimiter, converting strings to floats and back will fail if the default `FormatSettings` is used for conversion.

A common mistake in such situations was to change the global locale settings to the appropriate format for conversion, and then change it back to the original value after the conversion was finished. In single-threaded applications, and many Delphi applications were originally single-threaded, that kind of approach—while rather clumsy—worked. So even today, you can find such code in real life, still being used. If the developer is already using a locale compatible with the conversion, the code will work even with multithreading, causing spurious errors on customers' machines, along with "But it works on my machine!" syndrome.

And it is not just a case of developers writing end-user applications that were misusing global settings. You could find libraries that were misusing them, and occasionally such issues could even be found in core Delphi frameworks.

Imagine a situation where you've simply added a library to your multithreaded application, and suddenly some totally unrelated code started to break down because of formatting issues. In the case of `FormatSettings`, you don't even need an additional library to break thread safety. The Delphi application itself will happily do it by responding to OS changes in locale settings and updating the `FormatSettings` global, while you might be using it in a background thread.

Any formatting function that directly uses the `FormatSettings` global is not thread-safe, and can only be used in the context of the main thread (and this is also the case for any other function, method, or helper function that directly uses any mutable global data). Thread-safe formatting functions take an additional `FormatSettings` parameter, where you can pass an appropriate format settings instance that you have constructed specifically for that use case, which will only be read and will never change after creation.

Specifically, if you pass the global `FormatSettings` variable to a formatting function, you haven't made your code any more thread-safe than code using the thread-unsafe formatting function variant.

Since all `TFormatSettings` instances are mutable, there is always a chance that a developer will accidentally change some formatting instance that should not be changed, so even code using thread-safe formatting functions can be broken, but in such cases, that would be because the code outside the formatting functions was relying on mutable state, not the code inside the function. As you can see, thread safety is like onions (or ogres). It has layers: If you add thread-safe code onto thread-safe layers, the whole onion will be thread-safe. If you break the thread safety of some layer by reaching into unsafe shared data, then the thread safety of all the outer layers from that point will also be broken.

However, if you have some thread-unsafe data and code that is isolated from the rest of the code, you can put a protective thread-safe layer around that whole part, and as long as outside interactions must go through that layer, such data and the code manipulating it will also be thread-safe.

When it comes to formatting functions, some functions don't rely on any kind of format settings and are therefore thread-safe: For instance, the `IntToStr` and `StrToInt` functions. Date and float formatting functions, such as `DateTimeToString`, `FloatToStr`, `StrToFloat`, depend on format settings. Overloads of those functions that don't have a `FormatSettings` parameter are thread-unsafe, while overloads that take an additional `FormatSettings` parameter are thread-safe. Some caution is needed even with those thread-safe variants when formatting dates. See: *SysLocale*

Newer Delphi versions also have the class method `TFormatSettings.Invariant`, that returns a new `FormatSettings` instance populated with standard values corresponding to the English locale, but does not have any country/region set. You can freely tweak the returned instance to match your needs, because it will be a copy (`TFormatSettings` is a value type), and you don't need to worry that changing the returned `Invariant` will cause application-wide problems.

If the `Invariant` format settings were declared as a global variable like `FormatSettings` is, then any changes to such invariant could cause threading issues, and such a variable would not be thread-safe to use—unless you could guarantee (maybe with an obligatory trip to hell and back) that no developers ever changed the default invariant values.

6.1 SysLocale

Another similarly thread-unsafe type is the `TSysLocale` record and the global `SysLocale` variable. Some of the date formatting functions also depend on `SysLocale`, and because of this, they are not absolutely thread-safe even when format settings are passed as a parameter.

`SysLocale` is only used to determine whether a locale belongs to Far Eastern languages when formatting a date's era, as well as calculating character length and indexes in some MBCS string functions. So any issues that may arise can happen in those circumstances. The thread unsafety around `SysLocale` can only cause incorrectly formatted dates and parsed strings, but will not cause crashes and application instability.

Unfortunately, the `SysLocale` global is also used to fill data format information when constructing `TFormatSettings` instance even when you are passing locale ID in constructor, so in a way, constructing `TFormatSettings` in background threads is also not fully thread-safe. Again, this only impacts date formatting functionality around Far Eastern languages if the `SysLocale` change coincides with `TFormatSettings` initialization.

`SysLocale`, as well as `FormatSettings`, are initialized as part of the global `GetFormatSettings` function that is called by the VCL `Application` if the OS settings change. You can avoid this call by disabling `UpdateFormatSettings`:

```
Application.UpdateFormatSettings := False;
```

If you need to react to those changes, *and* your application uses the affected functionality in background threads, your only option will be to write your own version of those functions that don't rely on any global settings.

Thread-unsafe - broken

- formatting functions that are broken due to the FPCR issue—see: *FPCR*
 - `InternalTextToExtended` on Win64, and all functions that call it, like `StrToFloat`

Thread-safe

- formatting functions that don't use any kind of format settings
 - `Val`, `IntToStr`
 - `StrToInt`
 - all other integer formatting functions
- formatting functions that use a `TFormatSettings` instance passed as a parameter
 - `FloatToStr`
 - all other float, currency formatting functions with a `TFormatSettings` parameter
- the `TFormatSettings.Invariant` function

Thread-safe - under some circumstances formatting dates is not thread-safe

Depending on the format, and whether the application changes the contents of `SysLocale`, formatting dates and initializing `TFormatSettings` date formatting configuration, might not be fully thread-safe. See: `SysLocale`

- date formatting functions that use a `TFormatSettings` instance passed as a parameter
 - `DateTimeToString`
 - all other date formatting functions with a `TFormatSettings` parameter
- `TFormatSettings.Create` constructors

Single-thread-safe

- `TFormatSettings`

Thread-unsafe - main thread only

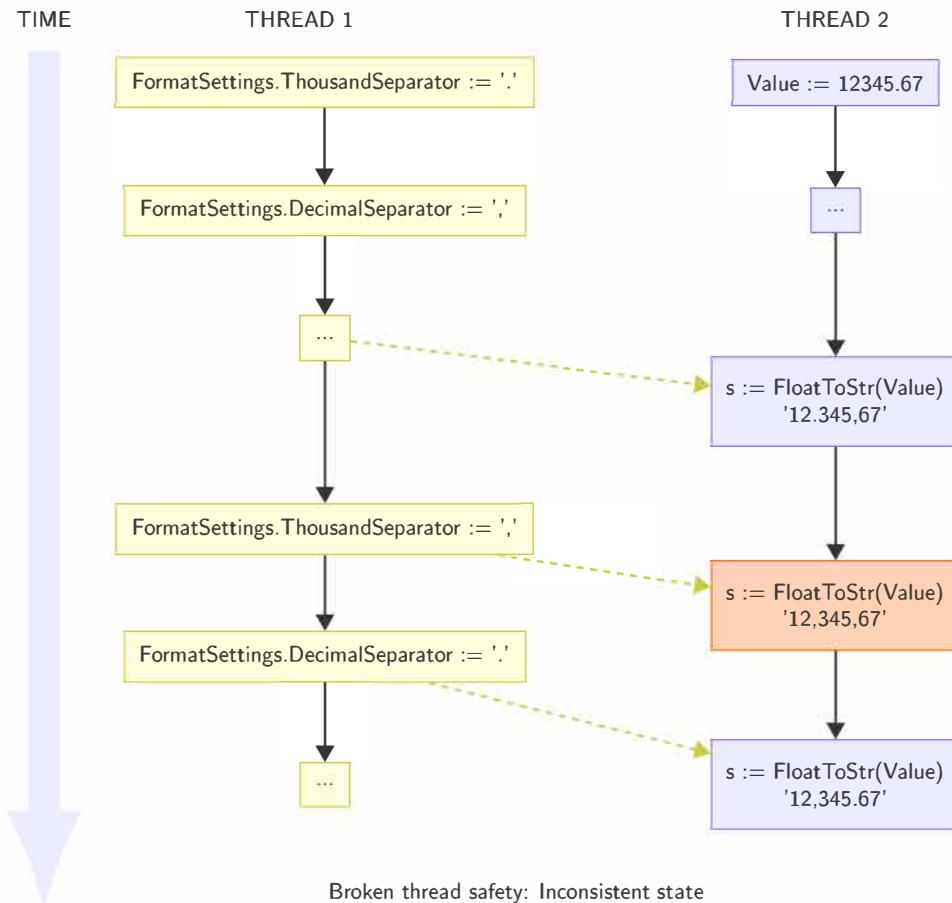
- the `SysLocale` global variable
- the `FormatSettings` global variable
- the `GetFormatSettings` function - calling this function even in the main thread can impact any code that uses `SysLocale` global: initializing `TFormatSettings` date formats, formatting dates with era information, and string functions that calculate character length and indexes in MBCS strings
- formatting functions that use the `FormatSettings` global variable
 - `DateTimeToString`
 - `FloatToStr`
 - all other float, currency and date formatting functions that use the `FormatSettings` global

6.2 Creating custom thread-safe format settings

Using the `FormatSettings` global and any formatting functions that use it are not an option for background threads and they can only safely use formatting functions with a `TFormatSettings` parameter. But we still need to figure out the best way to handle the variables that we will pass as parameters, as we already know that passing `FormatSettings` is not thread-safe either.

There are various ways we can solve `TFormatSettings` thread safety problems. The solutions are also applicable to other similar situations. One of the challenges of format settings concurrency issues is not just in maintaining and protecting the consistency of the stored settings, but also preventing changes while some particular code sequence is actively using it.

Inconsistent state

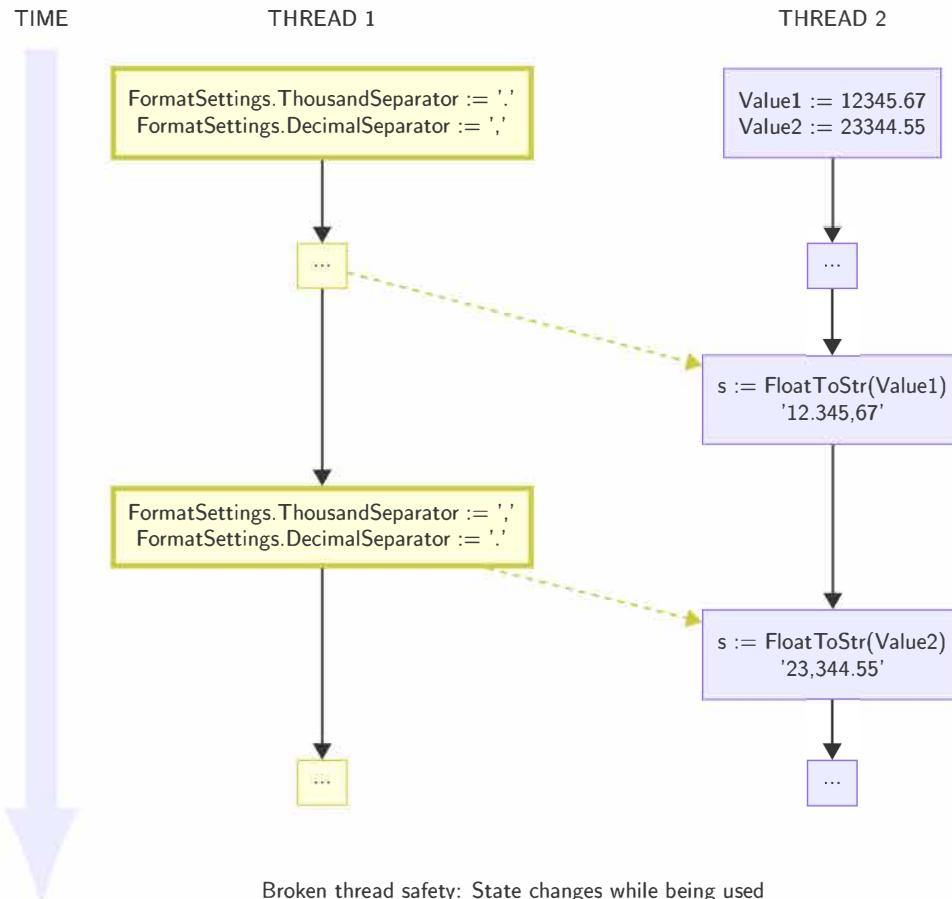


Failing to maintain data consistency is always the primary source of thread unsafety issues. Unprotected data can be modified from one thread and that can cause the other thread to read data while it is in an inconsistent state—when the needed changes have only been partially applied. This can cause various errors.

For instance, changing the `FormatSettings` separators used for formatting floating point numbers, can result in invalid formatting, where both separators will have the same value.

State changes while in use

Even after we solve problems caused by inconsistent state, there is another issue we need to resolve. Preventing any state changes while we are actively using some format settings variable. If we fail to do that, one thread can change settings while another thread is using those settings, and as a result, we can have data formatted (or parsed) with different settings which can represent a logical error.



Format settings thread safety revolves around two separate issues. The first is the availability of a single, global, unprotected `TFormatSettings` instance when applications may need multiple different sets of settings to cover specific formatting needs. Next is the fact that the `TFormatSettings` record is a mutable type. So any unprotected instance, will not be thread-safe, as it depends on discipline in order to achieve thread-safety. If an instance is accessible in a broader context, it is very hard to inspect and reason about the thread safety of a given chunk of code. The more broadly such instance is scoped, the easier it will be to inadvertently create thread-safety-breaking mutations.

Even if we add protection and locking mechanisms around any `TFormatSettings` instance that can solve issues with inconsistent state, we still have the problem of mutability and changing state while instance is in use.

As we already know thread-safe access requires protecting both writing and reading. The problem with `TFormatSettings` and consistency—this also applies to other similar configuration instances—is that we should not only lock the variable for the short time we spend calling some formatting function, but we should lock it for the complete code sequence where we need to ensure consistency, and such locking could create severe thread contention.

On the other hand, removing mutability also solves the consistency issue and we don't have to use any locking mechanism to achieve that. So potential solutions will fall into two categories: relying on immutability and locking. Immutability in this context does not necessarily mean true immutability, where changing data in code is impossible, but also includes a less strict definition where we rely on the assumption that the data will not be modified once it is initialized.

Immutable solutions:

- multiple, non-writeable typed constants - immutable data
- multiple, unprotected `TFormatSettings` variables initialized only once
- using private, unprotected instances initialized only once
- calling `TFormatSettings.Invariant` function, with settings additionally configured just before they are used
- factory function similar to `Invariant`, but returning a more specific configuration that does not require additional changes
- creating an immutable class or record - this is only applicable if we are creating some similar configuration settings from scratch, we cannot add immutability on top of the existing `TFormatSettings` record
- wrapping formatting functions together with a `TFormatSettings` instance that will be initialized only once, when the wrapper class or record is initialized

Locked solutions:

- wrap `TFormatSettings` in a record or a class together with a locking mechanism that can be used to protect larger pieces of code

6.2.1 Multiple, non-writeable typed constants

TFormatSettings is a record, and Delphi allows declaring constant records—they are immutable and that automatically makes them thread-safe. Trying to modify a constant record will cause a compiler error. As far as performance is concerned, using such a constant is also the fastest option. However, this approach is only feasible for format settings with predetermined values, that don't require any runtime configuration.

Note: Delphi also has a compiler switch that can enable declaration of writeable typed constants. To avoid issues if that compiler switch is turned on globally, it is prudent to explicitly disable it around constant declarations.

Writeable constants on:

```
{$J+} or {$WRITEABLECONST ON}
```

Writeable constants off:

```
{$J-} or {$WRITEABLECONST OFF}
```

```
{$WRITEABLECONST OFF}
const
  NumberSettings: TFormatSettings = (
    ThousandSeparator: ',';
    DecimalSeparator: '.');
```

And then such a constant can be used like this:

```
procedure FormatValues(Value1, Value2: Double);
begin
  Writeln(FloatToStr(Value1, NumberSettings));
  Writeln(FloatToStr(Value2, NumberSettings));
end;
```

6.2.2 Multiple, unprotected variables, initialized only once

If, for any reason, you cannot declare the needed settings as constants, you can use variables instead. This is the worst solution as it relies solely on discipline. From a performance perspective, it is as fast as using constant declarations. If you can guarantee that nobody will accidentally change the contents of such pre-initialized variables, code using such instances will be thread-safe.

To ensure that the used instances are properly initialized, you should do that very early on, during the initialization of the unit that declared those variables. If there is a possibility that

they can be used in class constructors, then you should also move their initialization to the class constructor. Pay attention to the order in which classes and units are initialized! If it is not straightforward, it would be better to rethink your design and initialize all dependencies explicitly outside the automatic initialization sequence.

```

interface

var
  NumberSettings: TFormatSettings;

implementation

initialization

  NumberSettings.ThousandSeparator := ',';
  NumberSettings.DecimalSeparator := '.';

end.

```

6.2.3 Using private instances

Another rather simple solution is using private instances of **TFormatSettings** in code (usually classes) that needs to use one. The downside of this approach is that you can end up creating plenty of redundant format settings instances. On the upside, each class or unit having its own instance makes them more bulletproof for possible changes in requirements. Any of the examples with private fields or more scoped variables can also use constant record declarations if the values used can be hardcoded. Using constants is always more thread-safe, because the compiler prevents inadvertent changes.

The following examples show a class with a private **FNumberSettings** field or constant that will be used for any formatting purposes within **TSomeData** class instances. The field is initialized in the instance constructor and not changed afterwards:

```

type
  TSomeData = class
    strict private const
      FNumberSettings: TFormatSettings = (
        ThousandSeparator: ',';
        DecimalSeparator: '.');
    ...
    public
      function Value1ToString: string;
    end;

```

```

type
  TSomeData = class
    strict private
      FNumberSettings: TFormatSettings;

      FValue1: Double;
      ...
    public
      constructor Create;
      function Value1ToString: string;
    end;

    constructor TSomeData.Create;
    begin
      FNumberSettings.ThousandSeparator := ',';
      FNumberSettings.DecimalSeparator := '.';
    end;

    function TSomeData.Value1ToString: string;
    begin
      Result := FloatToStr(FValue1, FNumberSettings);
    end;
  
```

FNumberSettings is unprotected, and its safety relies only on the fact that it will not be changed once initialized. Compared to globally shared unprotected settings, this pattern can be considered thread-safe, because the task of ensuring **FNumberSettings** will not be modified after initialization is confined to that single class' implementation code.

There are other issues with this kind of code. Having a large field within every instance of the class wastes memory, and this approach is appropriate only if you need to instantiate just a few instances of such a class.

This can be easily solved by re-declaring **FNumberSettings** as a class field or constant, and moving related initialization code into the class constructor. Every instance will share the same field. Such a field is as thread-safe as a private instance field because, again, the implementation is confined to a single class and not accessible to the outside world. Of course, this approach only works if the contents of the **FNumberSettings** field are allowed to be the same for all instances.

```

type
  TSomeData = class
    strict private
      class var FNumberSettings: TFormatSettings;
      class constructor ClassCreate;
      ...
    end;
  
```

If **TSomeData** class has descendant classes that also need access to the format settings field, we can still use it without breaking thread-safety as it will be confined to limited number of classes carrying additional requirement **FNumberSettings** field should not be written by any of them.

Instead of being private, we just need to redeclare **FNumberSettings** field as protected. Another approach is to expose it to descendant classes as a protected field-backed read-only property that will give us additional level of safety and prevent accidental mistakes. Field-backed properties of value types will not be copied while being accessed. Passing the **NumberSettings** property as a parameter will have the same performance impact as using the **FNumberSettings** field directly.

```

type
  TSomeData = class
    strict private
      class var FNumberSettings: TFormatSettings;
      class constructor ClassCreate;
    strict private
      FValue1: Double;
      ...
    strict protected
      class property NumberSettings: TFormatSettings read FNumberSettings;
      ...
    end;

  TOtherData = class(TSomeData)
    strict private
      FValue2: Double;
      ...
    public
      function Value2ToString: string;
    end;

    function TOtherData.Value2ToString: string;
    begin
      Result := FloatToStr(FValue2, NumberSettings);
    end;
  
```

If we need to use different sets of format settings within a single class hierarchy, we can easily declare additional, separate fields that will hold different configuration settings. It is important to remember that those private **TFormatSettings** instances are only allowed to be initialized once and must not be changed afterwards, or their thread safety will be broken.

If you have a collection of classes confined in a single unit, sharing the same formatting requirements, that are not all in the same class hierarchy, using a local shared format settings variable declared in the implementation part of the unit and initialized in the initialization section is also a viable option. Being confined to a single unit makes its scope small enough to keep such an instance under strict control and ensure its thread-safe use.

```
interface

implementation

var
  NumberSettings: TFormatSettings;

initialization

  NumberSettings.ThousandSeparator := ',';
  NumberSettings.DecimalSeparator := '.'

end.
```

6.2.4 TFormatSettings.Invariant

Using a factory function that always returns a new instance is a much safer approach in custom code when some particular format must be used than having a shared, mutable `TFormatSettings` variable. The downside of creating a copy every time is always some performance loss, and if performance is critical, then factory functions are far from being the best option.

Another issue with the `TFormatSettings.Invariant` function is that it needs additional changes to the returned record to match the current requirements, which is a fairly error-prone process.

```
procedure FormatValues(Value1, Value2: Double);
var
  Settings: TFormatSettings;
begin
  Settings := TFormatSettings.Invariant;
  // additional configuration of returned record, if necessary
  Settings.... := ...

  Writeln(FloatToStr(Value1, Settings));
  Writeln(FloatToStr(Value2, Settings));
end;
```

6.2.5 Factory functions

Creating different factory functions for different formatting needs solves the error-prone part of using the `TFormatSettings.Invariant` function. It suffers from the same performance issues, but otherwise, it is the simplest solution that also provides proper thread safety and does not require additional discipline to maintain:

```

function NumberSettings: TFormatSettings;
begin
  Result.ThousandSeparator := ',';
  Result.DecimalSeparator := '.';
end;

procedure FormatValues(Value1, Value2: Double);
begin
  Writeln(FloatToStr(Value1, NumberSettings));
  Writeln(FloatToStr(Value2, NumberSettings));
end;

```

While factory functions are extremely simple to write, and the same factory function can be safely used across plenty of different classes, private instances encapsulated in the class hierarchy are not only faster, but also a more flexible solution because they don't require a shared external dependency. It is far easier to refactor and change the behavior of such class hierarchy if formatting requirements change.

6.2.6 Immutable types

Creating immutable types can only be used when writing our own types. We cannot create an immutable wrapper around **TFormatSettings** that would not suffer from the same performance issues caused by the record copying as the factory functions.

The following immutable class and record examples can be used as templates for creating other kinds of custom configurations. If the globally accessible instance is immutable, then inadvertent changes—while still possible if the whole instance is replaced with another one—are less likely.

```

type
  TCustomSettings = record
    private
      FValue1: Integer;
      FValue2: string;
    public
      constructor Create(AValue1: Integer; const AValue2: string);
      property Value1: Integer read FValue1;
      property Value2: string read FValue2;
    end;

constructor TCustomSettings.Create(AValue1: Integer; const AValue2: string);
begin
  FValue1 := AValue1;
  FValue2 := AValue2;
end;

```

```

type
  TCustomSettings = class
  private
    FValue1: Integer;
    FValue2: string;
  public
    constructor Create(AValue1: Integer; const AValue2: string);
    property Value1: Integer read FValue1;
    property Value2: string read FValue2;
  end;

constructor TCustomSettings.Create(AValue1: Integer; const AValue2: string);
begin
  FValue1 := AValue1;
  FValue2 := AValue2;
end;

```

If the settings hold many different value fields, then the constructor declaration may get out of hand. This is the price of immutability.

While records are convenient because they don't require additional memory management, being value types makes them less viable if they have to be copied around. For hardcoded settings, records that can be declared as constants are clear winners, and for other use cases in need of completely protected, thread-safe solutions, classes are a better option.

A completely thread-safe solution can be achieved by protecting access to such immutable configuration variables. See: *Class fields, singletons, and default instances* chapter.

6.2.7 Complete functionality wrapper

Creating a functionality wrapper class or record is similar to private instances. The main difference is that the functionality wrapper can be available in the complete application scope, and is not confined to a small number of classes belonging to a class hierarchy or a unit.

Just like the private instance approach, a functionality wrapper will also hold instance(s) of a format settings record, but instead of being combined with other kinds of data and their particular formatting requirements, it will simply wrap the functionality of globally available formatting functions. Whether it wraps all of them or only a subset is just a question of requirements.

Particular instances of such a class then can be created as singletons and reused when needed to avoid unnecessary heap allocations/deallocations during the application lifetime.

If we need support for multiple different settings, we can always use different constructors with parameters that will define the initial values used.

```
type
  TFormatWrapper = class
    strict private
      FSettings: TFormatSettings;
    public
      constructor Create;
      function FloatToStr(Value: Double): string;
    end;

  constructor TFormatWrapper.Create;
begin
  FSettings.ThousandSeparator := ',';
  FSettings.DecimalSeparator := '.';
end;

function TFormatWrapper.FloatToStr(Value: Double): string;
begin
  Result := System.SysUtils.FloatToStr(Value, FSettings);
end;
```

```
procedure FormatValues(Value1, Value2: Double);
var
  Settings: TFormatWrapper;
begin
  Settings := TFormatWrapper.Create;
  Writeln(Settings.FloatToStr(Value1));
  Writeln(Settings.FloatToStr(Value2));
end;
```

```
type
  TFormatWrapper = class
    strict private
      FSettings: TFormatSettings;
    public
      constructor Create(const AThousandSep, ADecimalSep: string);
      ...

  constructor TFormatWrapper.Create(const AThousandSep, ADecimalSep: string);
begin
  FSettings.ThousandSeparator := AThousandSep;
  FSettings.DecimalSeparator := ADecimalSep;
end;
```

This wrapper approach can be simplified if there is only a handful of different format settings instances needed. In such case we can declare all needed settings as class fields in a single class and use differently named class functions for formatting.

We can also split different settings to different classes, while still using the class var approach. This and previous solutions all support a number of different code variations that still achieve the same final goal.

```

type
  TFormatWrapper = class
    strict private
      class var FNumberSettings: TFormatSettings;
      class var FOtherSettings: TFormatSettings;
    public
      class constructor Create;
      class function FloatToStr(Value: Double): string; static;
      class function OtherFloatToStr(Value: Double): string; static;
    end;

    class constructor TFormatWrapper.Create;
    begin
      FNumberSettings.ThousandSeparator := ',';
      FNumberSettings.DecimalSeparator := '.';
      FOtherSettings.ThousandSeparator := '.';
      FOtherSettings.DecimalSeparator := ',';
    end;

    class function TFormatWrapper.FloatToStr(Value: Double): string;
    begin
      Result := System.SysUtils.FloatToStr(Value, FNumberSettings);
    end;

    class function TFormatWrapper.OtherFloatToStr(Value: Double): string;
    begin
      Result := System.SysUtils.FloatToStr(Value, FOtherSettings);
    end;
  
```

```

procedure FormatValues(Value1, Value2: Double);
begin
  Writeln(TFormatWrapper.FloatToStr(Value1));
  Writeln(TFormatWrapper.OtherFloatToStr(Value1));
  Writeln(TFormatWrapper.FloatToStr(Value2));
end;
  
```

From a thread safety perspective, all solutions relying on immutability are thread-safe, and they can be simultaneously used across different threads. If you can use it—if configuration values can be hard coded—declaring a non-writeable constant (whether globally or more locally scoped) is absolutely the safest. The differences between other presented solutions are minimal in terms of thread safety. The only outliers are globally shared unprotected variables that are also thread-safe to use, but their thread safety can be more easily broken without realizing.

While the code doesn't vary much between the different solutions, some are faster than the others, some require less code, and some require less memory. Which ones (or their variation) you choose depends on your actual requirements and personal preferences.

Once again, the main requirement for any of those solutions (except constants, which are initialized at compile time) is to initialize the settings only once, as early as possible to prevent them from being used before initialization happens, and not modifying them after that.

We can also opt to use either classes or records for various implementations, but we need to keep in mind the functional differences between the two. Classes require explicit construction and destruction, while for records we can also manage initialization automatically with custom managed records, or even better, they can be initialized by declaring them as non-writeable constants. On the other hand, classes can be passed around as references, while records can trigger copying if we are not careful with our code.

6.2.8 Wrapper with locks

The previous solutions could only be used for configuration settings that don't have to be modified during application runtime. We still need to solve safely sharing mutable configuration settings across different threads. Since we cannot rely on immutability, we need to use some locking mechanism to ensure thread safety.

As previously mentioned, mutating requires protecting both write and read access. When it comes to configuration settings that will be frequently read and rarely written, using multiple-read/exclusive-write (MREW) locks will give better performance than using exclusive locks.

To prevent thread safety issues caused by state changes while the settings are being used in some code, we may need to apply a read lock to a significant amount of code while processing some data. Using exclusive locks like `TCriticalSection` or `TMonitor` in such scenarios could seriously reduce performance. Because of this, we also cannot use some types of MREW locks whose implementations do not give true multiple-read access and actually provide exclusive access.

For instance, `TMultiReadExclusiveWriteSynchronizer` implements MREW only on the Windows platform, while on other platforms, its implementation uses exclusive locking for all access types. Delphi 10.4.1 Sydney introduced `TLightweightMREW`, which fully supports MREW lock on all platforms, and I will use it to make a format settings wrapper with a lock.

The following is a rather crude locking solution. The main deficiency is that the `Settings` property is readily accessible for reading without invoking the locking mechanisms the way it is implemented in the *Exposing internal collection* chapter.

This was done because returning the `FSettings` field through `BeginRead` would imply copying its value, and that is something we want to avoid for performance reasons. This comes at the expense of safety, as it requires discipline to avoid thread-unsafe usage.

Of course, if you are not concerned about performance, or the wrapped settings are a class, not a record, then returning the field through the `BeginRead` function instead of exposing it directly through a property is a much safer option.

```
type
  TThreadFormatSettings = record
    strict private
      FLock: TLightweightMREW;
      FSettings: TFormatSettings;
      procedure SetSettings(Value: TFormatSettings);
    public
      procedure BeginRead;
      procedure EndRead;
      property Settings: TFormatSettings read FSettings write SetSettings;
    end;

    procedure TThreadFormatSettings.SetSettings(Value: TFormatSettings);
    begin
      FLock.BeginWrite;
      try
        FSettings := Value;
      finally
        FLock.EndWrite;
      end;
    end;

    procedure TThreadFormatSettings.BeginRead;
    begin
      FLock.BeginRead;
    end;

    procedure TThreadFormatSettings.EndRead;
    begin
      FLock.EndRead;
    end;
```

```

var
  Wrapper: TThreadFormatSettings;

procedure FormatValues(Value1, Value2: Double);
begin
  Wrapper.BeginRead;
  try
    Writeln(FloatToStr(Value1, Wrapper.Settings));
    Writeln(FloatToStr(Value2, Wrapper.Settings));
  finally
    Wrapper.EndRead;
  end;
end;

procedure ModifySettings;
var
  LSettings: TFormatSettings;
begin
  LSettings.ThousandSeparator := ',';
  LSettings.DecimalSeparator := '.';
  Wrapper.Settings := LSettings;
end;

```

If the code you need to lock for reading is large enough, where locking would create more of slowdown than copying the settings, long locks can be avoided by assigning settings to a local variable that will then be used in subsequent code. Please note that this approach only works with value types, where assignment creates a copy. In the case of reference types, it would require making a deep copy of the instance, not just taking a reference.

```

procedure FormatValues(Value1, Value2,...: Double);
var
  LSettings: TFormatSettings;
begin
  Wrapper.BeginRead;
  try
    LSettings := Wrapper.Settings;
  finally
    Wrapper.EndRead;
  end;
  Writeln(FloatToStr(Value1, LSettings));
  Writeln(FloatToStr(Value2, LSettings));
  ...
end;

```

Another, safer solution that does not require exposing the inner settings field would be creating a functionality wrapper with an added lock to avoid unprotected access, where every wrapped function would ask for a read lock before usage. That approach would also cause slowdowns due to lock access for every single use, as well as multiple `try...finally` blocks.

In any case, regardless of the approach, the same mutable configuration settings should not be abused for making transient configuration changes in some code and reverting the old value back, similarly to the way the global `FormatSettings` were abused, as such usage will have negative impact on multithreading performance.

Chapter 7

Global functions and procedures

From the previous format settings example, we can see that functions and procedures are thread-safe if they don't access any unprotected global state. But what about passed parameters? What kind of impact do they have on thread safety?

Bringing parameters into the discussion creates a *The Pirate Code* moment, where the rules are not as clear, and are more like guidelines. When you are considering actual code, then you can always say whether such code is thread-safe or not, but considering the thread safety of general, unwritten code, can be more complicated. And as always, when thread safety is concerned, it is way easier to detect or define thread-unsafe code, than to define thread-safe code.

Functions and procedures that don't directly or indirectly access any unprotected global state and don't have any parameters are obviously thread-safe. With parameters, the situation gets a bit more complicated, considering the different behaviors of different types and various parameter modifiers that can have a significant impact on thread safety—or so it seems at first glance.

7.1 Parameters

The default pass-by-value parameter handling, combined with a simple value type, should be thread-safe, right?

After all, pass-by-value creates a copy of the passed parameter. Passing a value type by value creates a copy of the whole data. If the function has a local copy, then calling such a function or procedure should be thread-safe.

Well, it depends on how you define thread safety and correct behavior in a broader context. If you look at the `IntValue` function alone, then it is obviously thread-safe—it does not directly access any shared state, and it simply returns a copy of the passed integer parameter. You can hardly imagine any simpler function that actually uses a passed parameter.

But, what happens if the passed parameter is being shared between multiple threads, and not protected by any synchronization mechanism? Can we expect that code in the second thread that passes `0` to the `IntValue` function will always return `0`? Well, if zero goes in, zero will certainly go out. But does that code guarantee that zero will actually always get in?

It does not. At some point, the first thread will be able to set the unprotected, shared `Value` variable to `-1`, between the point where the second thread sets it to `0`, and the point where `Value` is copied and its copy is passed to the `IntValue` function:

```
var
  Canceled: Boolean;

function IntValue(Value: Integer): Integer;
begin
  Result := Value;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
var
  Value: Integer;
begin
  Canceled := False;
  TThread.CreateAnonymousThread(
    procedure
    begin
      while not Canceled do
        begin
          Value := -1;
        end;
    end).Start;

  TThread.CreateAnonymousThread(
    procedure
    var
      ValueCopy: Integer;
    begin
      while not Canceled do
        begin
          Value := 0;
          ValueCopy := IntValue(Value);
          if ValueCopy <> 0 then
            Canceled := True;
        end;
    end).Start;
end;
```

Now, let's imagine that we are fine with the behavior of the above code... and we don't care whether zero is actually passed in or not. All we care is that we get either **0** or **-1**. All is good... but then your requirements change and instead of a 32-bit integer, you need to use a 64-bit integer on a 32-bit platform:

```
function Int64Value(Value: Int64): Int64;
begin
  Result := Value;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
var
  Value: Int64;
begin
  Canceled := False;
  TThread.CreateAnonymousThread(
    procedure
    begin
      while not Canceled do
        begin
          Value := -1;
        end;
    end).Start;

  TThread.CreateAnonymousThread(
    procedure
    var
      ValueCopy: Int64;
    begin
      while not Canceled do
        begin
          Value := 0;
          ValueCopy := Int64Value(Value);
          if (ValueCopy <> 0) and (ValueCopy <> -1) then
            begin
              Canceled := True;
              OutputDebugString(PWideChar(ValueCopy.ToString));
            end;
        end;
    end).Start;
end;
```

If you think you will get lucky with the above code.... well, you just might... but in real life, at some point, data tearing will occur, and your 64-bit integer will end up having the value

00000000FFFFFF instead of 0000000000000000 or FFFFFFFFFFFFFF.

If you consider passing more complex value types or reference types, it is rather obvious that such unprotected and thread-unsafe parameters can wreak havoc when passed to an otherwise thread-safe function. The real question is whether we should consider similar functions/procedures/methods thread-safe or not.

Well, saying they are thread-safe might get people in trouble... but saying they are not thread-safe is not helpful either, because in that case basically nothing would be thread-safe. And this is where our *Pirate Code* kicks in. Playing with pirates or threads is never ever quite safe, but if you keep in mind that every thread-safe function is as safe as the parameters passed to it, we can call such functions thread-safe, to distinguish them from truly thread-unsafe functions that are directly messing with unprotected global state.

Since even pass-by-value parameters are not fully thread-safe, are other kinds of parameters more thread-unsafe? Can we still add some kind of additional thread safety categorization based on parameters alone?

The answer is, unfortunately, no.

Eventually, it all comes back to the safety of the passed parameters alone, not the parameter modifiers. Even when you use modifiers like `var` that directly enable modifying the parameter within the function body, that fact will not tell us anything about thread safety of such function. Only when we take into account passed parameter we can determine safety of the actual code. If the parameter is itself thread-safe, then the function will be thread-safe, too.

Thread safety is ultimately defined by the code inside the function, as well as the outside context and the thread safety of the passed parameter combined. This is what creates the thin line between thread safety or thread unsafety when you actually use some particular function.

7.2 FreeAndNil

Thread-unsafe

If we were to follow the *Pirate Code* to the letter, we would be able to say that `FreeAndNil` is thread-safe, or at least as thread-safe as `StrToInt`. But this is one of those places where saying one thing immediately makes people believe we said something else.

We will treat the `FreeAndNil` procedure as thread-unsafe, as it can only be safely used if either the passed object instance is not shared between threads, or its access is protected by additional synchronization mechanisms.

We will treat it as unsafe not because of the above (which is valid for any other function with parameters), but because saying `FreeAndNil` is thread-safe would immediately result in developers shooting themselves in the foot by using `FreeAndNil` in a thread-unsafe manner, trying to `nil` the same, unprotected object reference from multiple threads. Shooting yourself in the foot with `StrToInt`, while certainly possible, is not as common a scenario.

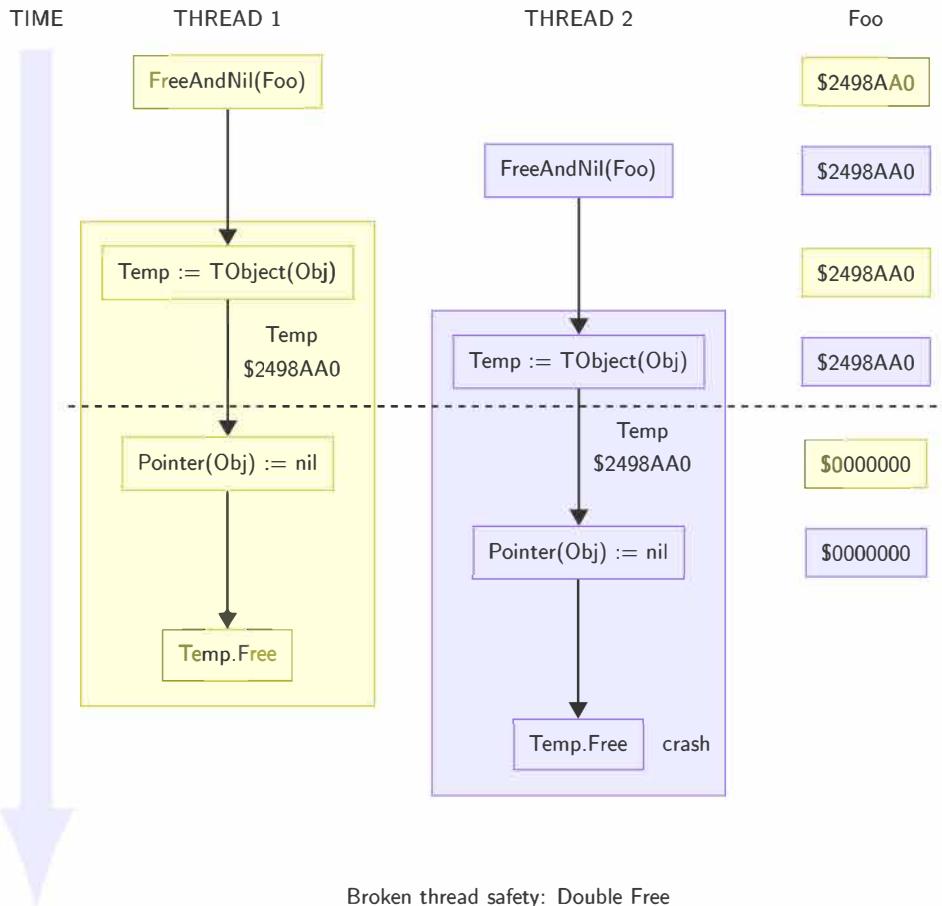
To better understand a common misunderstanding we should look at what **FreeAndNil** does and what is a common misconception. Depending on the Delphi version, there are two **FreeAndNil** variants. The outcome is the same in both: The object is destroyed, and the passed object reference is set to **nil**: (Please note that I am describing the final outcome, not the exact order in which those two operations happen.)

```
procedure FreeAndNil(var Obj);
var
  Temp: TObject;
begin
  Temp := TObject(Obj);
  Pointer(Obj) := nil;
  Temp.Free;
end;
```

```
// available since Delphi 10.4
procedure FreeAndNil(const [ref] Obj: TObject);
var
  Temp: TObject;
begin
  Temp := Obj;
  TObject(Pointer(@Obj)^) := nil;
  Temp.Free;
end;
```

Reasoning about the first variant's unsafety is rather simple. Not only does that **var** parameter modifier allow the procedure to modify the passed object instance reference which is being set to **nil**, but there is also an assignment to a temporary reference happening first, and that temporary reference is then used to free the object. It is quite obvious that if the passed object reference is not protected in the external context, it could be simultaneously accessed from multiple threads, and that **FreeAndNil** code could easily cause a double free. Not only that, but one quite possible scenario is that one thread releases the object, while another is still using it.

In the second procedure code, the internal logic is basically the same. The only difference is in the particular code that assigns and nils the reference, because different parameter modifiers require slightly different code to achieve the desired result. As far as unsafety is concerned, the reasons are the same. What is particularly interesting in the second procedure is that **const [ref]** modifier does not make it obvious that the passed variable can be modified and yet, as we can see, such a thing is still possible by doing some pointer trickery. This is yet another example that parameter modifiers cannot be used as any kind of *thread safety proof*.



At this point, you are probably thinking that the initial reasoning about `FreeAndNil` being thread-unsafe was solid, and that we cannot ever say that it is thread-safe in the way `StrToInt` is thread-safe. The main source of `FreeAndNil`'s unsafety is its internal code, not whether or not the parameter is protected in the outside context.

Anyone that has been in a position where they used `FreeAndNil` on an unprotected variable has probably had the idea that making `FreeAndNil` thread-safe would be a solution. After all, when using interfaces, clearing the interface reference and consequently destroying the object is a thread-safe operation.

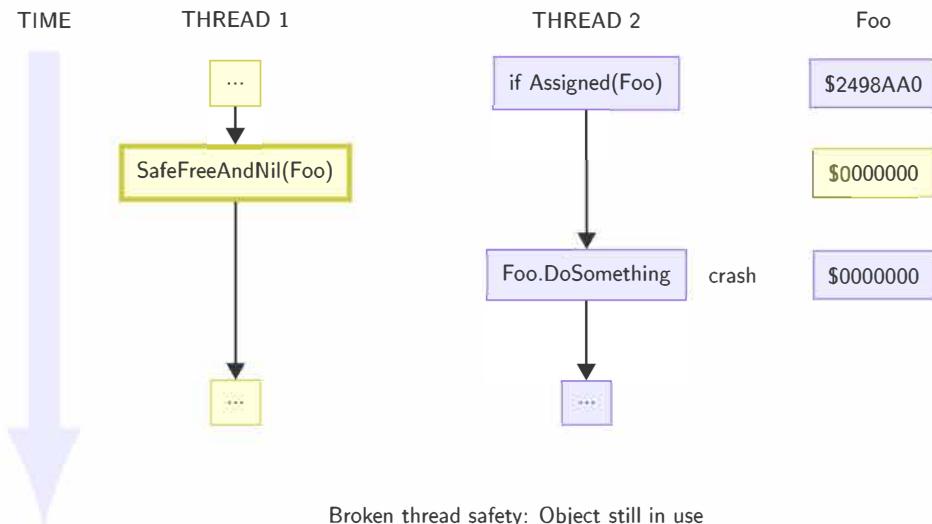
Now, we could easily use the Interlocked API to assign the passed variable to a temporary variable and set it to `nil` in one go, achieving inner thread safety and avoiding a double free. But, the main problem is not in `FreeAndNil` itself, but the code in the outer context. Preventing a double free does nothing for the situation where a variable that is still being used in one thread can be destroyed in another.

```

procedure SafeFreeAndNil(var Obj);
var
  Temp: TObject;
begin
  Temp := TInterlocked.Exchange(Pointer(Obj), nil);
  Temp.Free;
end;

```

This is ultimately a memory management issue. With interface references, every thread can hold a strong reference to object and that strong reference will keep that object alive as long as the thread is using it. It is the reference counting that ensures the thread-safe destruction of a shared object only when nobody else is using it. With regular, untracked object references, you cannot possibly detect that moment, nor keep some thread from destroying an object that is still in use.



Even if you make `FreeAndNil`'s internal code thread-safe, you cannot achieve thread safety in the broader context without an additional synchronization mechanism. Making `FreeAndNil` thread-safe would be an exercise in futility, and it would only make it run slower. If we protect the passed variable in the outside context, then all variants of the `FreeAndNil` are equally thread-safe.

For the same reasons, any mutating function is equally thread-unsafe without an external protection layer, but all such functions are also basically thread-safe, as it is not their job to—nor can they—provide thread safety in a broader context. Adding that on top of the example where even `StrToInt` and `IntToStr` are not really thread-safe if passed parameters can be unsafely modified in the broader context, all such functions—mutating or not—can be put in the same thread safety category.

It is really a “*potato, potahto, tomato, tomahto*” situation.

Since putting just about everything in the thread-unsafe category is as helpful as saying all pirates are dangerous, having different categorizations (green and blue) for basically the same thing—in other words, distinguishing between dangerous and very dangerous pirates—helps in raising your alert levels when dealing with code that can more easily be misused.

Another variant of this book would contain only a single page with the sentence “It is not thread-safe unless you make it thread-safe”. Keep that in mind while reading the rest of this book, and know that in the end, thread safety depends not only on how thread-safe or broken the building blocks you use are, but also on how you combine them together, and all the glue code between them.

7.3 Class methods

Class methods (class or record methods marked with the `class` keyword) are just global functions in disguise, and the process of determining their thread safety is no different than determining the thread safety of global functions. If they access unprotected global variables, they are not thread-safe; otherwise, they can be considered thread-safe if the passed parameters are protected, or not shared in the outside context.

When it comes to helper functions, they follow the same pattern. The only thing to remember is that if they are not class methods, and they are called on a variable, that variable is effectively passed as a parameter. If the variable is not thread-safe, neither is the function call—in other words, the regular method rules apply.

If class methods access class fields, then the thread safety of those fields must be considered in order to determine the thread safety of such a method. More about them can be found in the *Class fields, singletons, and default instances* chapter.

7.4 Other global functions

Previous chapters specifically covered the thread safety of formatting functions and the `FreeAndNil` procedure. But the Delphi RTL has many more global functions, including static functions organized in various utility records and classes, as well as helpers. Covering each and every one of them would be almost impossible.

Luckily, most of those functions don’t access any global state and they are thread-safe to use, just like thread-safe formatting functions or `FreeAndNil`, as long as you don’t pass any unprotected shared data as their parameters. The majority of those that are not thread-safe suffer from the previously mentioned problems: the FPCR issue, and formatting.

Generally, the list of thread-unsafe functions that cannot be used in background threads will be rather short, and it is also much easier to think about a few functions you need to avoid when writing thread-safe code, than to dwell on the thread safety of each and every line of code you

might write (which you will actually do anyway, but on a more subconscious level that will not impact your coding speed). Still, even short lists can be long enough, and there will always be something not on the list that might not be thread-safe, and the RTL may be the beginning, but it is not the end, and there are other functions and code beyond it, in which you will need to know whether it is thread-safe. Eventually, for most code, you will need to figure out thread safety yourself.

If you have access to the source code, determining thread safety is usually rather trivial, as you can easily spot whether some function is accessing unprotected global data or not. If you don't have access to source code, you will have to rely on documentation.

You can also deduce the thread safety based on the expected functionality, and logically conclude whether a function needs to access something outside the passed parameters to perform its functionality. Deducing without seeing the real code is ultimately not the best way to reason about thread safety, and if you wrongly assume a function is thread-safe when it is not, such an assumption can have a negative impact on the behavior of your whole application.

I am only mentioning this as a possibility, not to encourage such deductions, but to cover all options and warn about the roads you should never ever take, even when you are out of other options.

But the real danger does not lie in individual functions, as they will be unsafe in some rare occasions, but from the combined logic you need to perform. Some examples can be found in the *Thread-safe data types used in an unsafe manner* or *Thread-safe iteration* chapters, where combining individual thread-safe operations can still result in thread-unsafe handling of the data, and failure to achieve the required functionality.

Also when interacting with the OS and other outside resources, not only do you need to be careful to avoid race conditions within your application, but also race conditions between other running applications and processes.

For instance, if you are working with the Windows Registry, operations will be atomic, which ensures the consistency of the data written, but overall, you cannot guarantee the full thread safety of some code, because other applications may access and modify those values behind your back. And in such cases, not even having a single-threaded application will do you much good.

Working with file systems suffers from similar logical issues. If your code logic involves checking whether some file exists and then performing some operation depending on that check, such code will never be thread-safe. And even if you don't have a race condition inside the application itself, you may have a race condition with other applications running, as they can create or delete that particular file.

Another rather obviously thread-unsafe function involving the Windows filesystem is the `SetCurrentDirectory` function, and all other directory-changing functions across the RTL. If the application starts changing the current directory while multiple threads are running, then reading the current directory with the `GetCurrentDirectory` function will also suffer from concurrency issues.

What may not be so obvious in this case, is that the `GetFullPathName` function, which expands relative paths into absolute paths, uses the current directory value as the starting point for

computing the absolute path. In multithreaded applications where the current directory is being changed, consecutive calls to `GetFullPathName` on the same relative path may return different results.

`ExpandFileName`, which uses the thread-unsafe `GetFullPathName` function, is also affected, as is any code calling it. Of course, this is only a problem if the current directory is being changed after multiple threads start running.

When dealing with OS and other APIs, even if the Delphi side of the code seems thread-safe, that does not mean that the API behind it is thread-safe, and caution is always necessary when dealing with such code.

When it comes to deducing thread safety based on expected functionality, the most common potentially thread-unsafe functions are ones adding or removing list items.

For instance, `System.SysUtils.AddExitProc` or `System.Classes.RegisterClass`. A quick look at the source code is enough to establish that `AddExitProc` is not thread-safe, as it handles unprotected global data, while `RegisterClass` and other related functions are thread-safe, because they use locks to protect access.

Another example of a thread-unsafe function is `System.SysUtils.Languages`, that lazily constructs and returns a global list of languages used by the system. Its lazy initialization is not thread-safe, but once constructed, it can be safely used.

Chapter 8

Class fields, singletons, and default instances

The singleton design pattern restricts instantiation of a class to a single object instance. This terminology is also commonly used for the default instances of some classes that are not necessarily restricted to being instantiated only once.

Default instances are commonly stored in the static fields of a class or record, and accessed through class methods or class properties. Besides default instances, the static members of a class or record—fields and methods—also provide functionality and behavior similar to a singleton. Before class fields were introduced, a common way to declare default instances and singletons was to use variables declared in the implementation section of a unit—to limit their visibility and uncontrolled access in a broader scope—and accessed through a set of publicly-available global functions.

From a thread safety perspective, there is not much difference between a pure singleton instance, a default instance, a static class or record as a whole, or a unit-local variable, and in this chapter, I will use the term singleton to refer to any of them.

A singleton is just another device of global state. How thread-safe they are depends on whether their state can be mutated, and whether such mutation is protected and has unintended side-effects when the singleton is used from multiple threads. Thread safety also depends on whether the singleton reference or other static fields are directly exposed and assignable from a broader context. If they are, then their thread safety can be compromised. In that regard, all public fields are automatically thread-unsafe, just like any global variable that can be accessed without going through any protection mechanisms.

Just like any other data type can have some additional conditions attached that define whether a particular usage is thread-safe or not, singletons can also have them. If those specific requirements are not met, then such usage will not be thread-safe.

However, the main requirement for a singleton instance to be thread-safe is that its type is

thread-safe. If the type itself is not thread-safe, then even if we protect the instance itself, we cannot possibly use such a singleton in a thread-safe manner, or at least we can only use it safely following the thread safety constraints imposed by its type.

To give some examples:

Thread-unsafe - main thread only

- **TMessageManager.DefaultManager** - it is not thread-safe because **TMessageManager** instances are not thread-safe when used across different threads, and the **DefaultManager** instance can only be used from the context of the main thread.

Thread-safe

- **NxHorizon.Instance** - is the default instance of a thread-safe event bus class, and the instance itself is also protected, so it can be freely used across multiple threads.
- **TNetEncoding.Html** - this singleton is lazily constructed in a thread-safe manner, and the **THTMLEncoding** class is thread-safe because it does not hold any state
- **TNetEncoding.Base64** - is also lazily constructed in a thread-safe manner, but unlike **THTMLEncoding**, **TBase64Encoding** class holds configuration data. However, configuration data can be only set through the constructor, and the **TBase64Encoding** class is therefore thread-safe (unless you resort to hacking)

Thread-safe - additional conditions apply

- **NxLog** - the logger implemented in the *Logging* chapter is example of a singleton that is thread-safe only under certain conditions. Configuring the logger is not thread-safe, and must be completed before multiple threads start using logging functionality. If that requirement is met, and the class implementing **INxLogger** interface is also thread-safe, then further usage of the logger will be thread-safe. If the class implementing the **INxLogger** interface is not thread-safe, then such a logger can only be used in the context of the main thread.

The lifecycle of every variable, and therefore singletons as well, has three distinct stages: initialization, usage, and finalization. What kind of protection is required depends on how the variable is used, its type, and when the initial and final stages can occur. For instance, a lazily initialized instance requires protection during the initialization stage, while an instance that is initialized before threads start running does not. If the reference (variable) to the thread-safe instance needs to be writeable at any moment while it is being used, then it needs to be protected by a lock.

8.1 Class constructors and initialization section

Class constructors and code in the initialization section runs in the context of the main thread, before other threads start running, and generally, initialization in those parts of code does not require any additional protection.

Protection is only needed in situations with complex unit dependencies, where initialization order is not guaranteed, and where you spawn new threads from inside class constructors or the initialization section. This kind of scenario can easily turn into a nightmare, not only from an initialization standpoint, but also finalization, so the best approach is avoiding such complex automatic initialization involving threads.

I am not saying it is impossible to handle such scenarios, and there are ways you can ensure that threads wait until the whole initialization process is completed, but I am not going to give examples on how to do that. There is a huge difference between a simple example that works, and a complex real-life situation where you can lose control no matter how hard you try. If you think you can handle such scenarios, then go right ahead, but any trouble you run into will be self-inflicted.

Just as we can live with an explicit `Application.Initialize` in the main project block, called after we know all other necessary automatic initialization is completed, we can also live with an additional line of code explicitly starting other complex processes that involves threads.

If we assume that there are no threads involved when class constructors and initialization sections are executed, and the instance is otherwise read-only, then no protection is needed for such an instance, and it will be safe to use in threads. We can also extend this *no protection* rule to the singleton instances that are explicitly initialized later on, as long as that initialization happens before any background threads that will use them start running.

When it comes to the finalization of such singletons, and any others for that matter, it is important to ensure that there are no threads running at the point of finalization. That means we need to take care of proper shutdown, and wait for all threads to finish running before the automatic cleanup in the finalization sections and class destructors kicks in.

Initialization of such a singleton is rather simple. It is important that the variable holding the singleton is declared in the implementation section, so that it is not accessible from an outside scope. This is important to prevent writing to the variable, which would not be thread-safe. Of course, any code in the implementation section of this unit is also not allowed to write to the singleton variable once it is initialized.

The following example uses a regular, manually managed class as its instance type. Using a reference-counted class is not much different. The only difference is that the variable and function declarations should be interfaces in such a case, and we wouldn't need to explicitly free such an object.

When it comes to performing a clean shutdown, automatically managed instances have one advantage. If the code using them has additionally captured such an instance and thus increased its reference count, then even if the owning unit has been finalized, such an instance will still be alive due to its reference count.

However, I don't advise exploiting that fact, as there are a number of factors to consider, and once units start their finalization process, it can be very hard to perform a clean shutdown of still-running threads.

Thread-safe unit-local singleton

```
interface

type
  TFooObject = class
  end;

  function Local: TFooObject;

implementation

var
  FooObj: TFooObject;

  function Local: TFooObject;
begin
  Result := FooObj;
end;

initialization
  FooObj := TFooObject.Create;

finalization
  FooObj.Free;
end.
```

Using class constructors follows a similar pattern. The important parts of this code are that the instance field should be declared private, and the class property must be read-only. We don't need a getter function here, because it would just simply return the field, as we don't need to implement additional protection. Again, no part of the code outside the class constructor is allowed to write to the **FInstance** field, or that code would break the singleton's thread safety.

Class-property singletons also have a few advantages comparing to unit-local singletons. First, class constructors and destructors are automatically called and if the class is never used in code, any such class related code will not be linked in the final application, reducing unnecessary code bloat. Another advantage is that accessing a class property directly backed by a field is faster

than accessing a variable through a function.

Regardless of the singleton approach we take, we can safely declare and use more than one singleton inside each unit or class.

Thread-safe class-property singleton

```
interface

type
  TFoo = class
  private
    class var FInstance: TFoo;
    class constructor ClassCreate;
    class destructor ClassDestroy;
  public
    class property Instance: TFoo read FInstance;
  end;

implementation

class constructor TFoo.ClassCreate;
begin
  FInstance := TFoo.Create;
end;

class destructor TFoo.ClassDestroy;
begin
  FInstance.Free;
end;

end.
```

8.2 Thread-safe lazy initialization

Initializing instances in initialization sections or class constructors is suitable for singletons that are needed for the whole application lifetime, where lazy initialization does not make much sense, because the singleton will be initialized at the very beginning anyway.

For other instances, which are not immediately needed or might not be needed at all—depending on the end user’s workflow—lazy initialization is often used to prevent unnecessary consumption of resources.

Because lazy initialization can happen at any time, the commonly used lazy initialization pattern—checking for whether an instance is constructed and constructing it if it is not—is not thread-safe, and we need to prevent concurrency issues in there.

Thread-unsafe lazy initialization

```
class function TSingleton: GetInstance: TFooObject;
begin
  if FInstance = nil then
    FInstance := TFooObject.Create;
  Result := FInstance;
end;
```

There are many ways to do so, but the simplest approach is using a lock to protect the instance. However, locking the instance with an exclusive lock every time it is accessed is a real performance killer, especially if multiple threads often access such an instance simultaneously.

Thread-safe lazy initialization with a lock

```
type
  TSingleton = class
  private
    class var FInstance: TFooObject;
    class var FLock: TCriticalSection;
    class constructor ClassCreate;
    class destructor ClassDestroy;
    class function GetInstance: TFooObject; static;
  public
    class property Instance: TFooObject read GetInstance;
  end;

  class constructor TSingleton.ClassCreate;
  begin
    FLock := TCriticalSection.Create;
  end;

  class destructor TSingleton.ClassDestroy;
  begin
    FInstance.Free;
    FLock.Free;
  end;
```

```

class function TSingleton.GetInstance: TFooObject;
begin
  FLock.Enter;
  try
    if FInstance = nil then
      FInstance := TFooObject.Create;
  finally
    FLock.Leave;
  end;
  Result := FInstance;
end;

```

Because the singleton will be written only once, we can optimize the above example and use the lock only if the singleton has not yet been constructed. If the singleton instance is assigned, that means the instance is already constructed and ready for use, so we can just use it. If the instance has not been constructed, locking ensures that only a single thread can enter that part of the code, and the additional nil check will ensure that another thread hasn't constructed the instance while the current thread was waiting to enter the lock. This pattern is called *double-checked locking*.

Thread-safe double-checked locking lazy initialization

```

class function TSingleton.GetInstance: TFooObject;
begin
  if FInstance = nil then
    begin
      FLock.Enter;
      try
        if FInstance = nil then
          FInstance := TFooObject.Create;
      finally
        FLock.Leave;
      end;
    end;
  Result := FInstance;
end;

```

While the previous example with double-checked locking does not suffer from performance issues, it is still wasteful, because you need to maintain a separate lock that will be useless once the instance is initialized.

We can avoid maintaining a separate lock with the *lock-free lazy initialization* pattern. If the instance is not yet assigned, the new instance is constructed and stored in a local reference. Then, with the help of the **TInterlocked** API, it is compared and assigned in a thread-safe

manner to the field. If the `CompareExchange` call fails, that means another thread has successfully constructed and assigned `FInstance` in the meantime, and the current thread's redundant instance needs to be released.

A downside of this pattern is that if multiple threads try to initialize the instance at the same time, more than one instance of the required class will be constructed. From a functionality standpoint, the code will work correctly because only one of those instances will be assigned to the `FInstance` field, and the rest will be released.

Because there is the possibility of constructing multiple instances, this approach is suitable only for lightweight classes, where construction is fast or otherwise not resource-intensive.

Thread-safe lock-free lazy initialization

```
type
  TSingleton = class
  private
    class var FInstance: TFooObject;
    class destructor ClassDestroy;
    class function GetInstance: TFooObject; static;
  public
    class property Instance: TFooObject read GetInstance;
  end;

  class destructor TSingleton.ClassDestroy;
begin
  FInstance.Free;
end;

class function TSingleton.GetInstance: TFooObject;
var
  LInstance: TFooObject;
begin
  if FInstance = nil then
    begin
      LInstance := TFooObject.Create;
      if TInterlocked.CompareExchange<TFooObject>(FInstance, LInstance, nil)
        <> nil then
        LInstance.Free; // Free redundant instance
    end;
  Result := FInstance;
end;
```

For reference-counted classes, lock-free lazy initialization requires slightly different code to keep the reference count in order. In the following example, `Result` is used as temporary storage for

the newly constructed object instance, to avoid the declaration of an additional local variable.

CompareExchange works on pointers, and when used with interface references, it doesn't update the reference count. Because of that, if the new object referenced by **Result** was successfully moved to **FInstance**, its reference count will still be one, while being pointed to by two interface references. To fix the reference count, we need to clear the value in **Result** without triggering reference counting.

If **CompareExchange** fails, the redundant object instance will be automatically released with the last line of code, that assigns the other object referenced by **FInstance** to **Result**.

Thread-safe lock-free lazy initialization

```

type
  IFoo = interface
  end;

  TFoo = class(TInterfacedObject, IFoo);

  TSingleton = class
  private
    class var FInstance: IFoo;
    class function GetInstance: IFoo; static;
  public
    class property Instance: IFoo read GetInstance;
  end;

  class function TSingleton.GetInstance: IFoo;
begin
  if FInstance = nil then
    begin
      Result := TFoo.Create;
      if TInterlocked.CompareExchange(Pointer(FInstance), Pointer(Result), nil)
        = nil then
        Pointer(Result) := nil; // Clear without triggering ARC
    end;
  Result := FInstance;
end;

```

The question that comes up next is whether we can somehow avoid constructing the lock for lazy initialization, in situations where constructing redundant instances is not an option. One of the issues with a lock is not just that we need to pack an additional variable, but we are also constructing the lock object on the heap. No matter how small that is, it is still too much.

Of course, we can always use a more lightweight record-based lock, like for instance,

TLightweightMREW, even though it will only be used for writing. But what we can actually do here is spinning until we successfully construct an instance using a simple boolean flag to ensure we only create it once—we'll use an integer because **TInterlocked** doesn't work on booleans.

Thread-safe spin wait lazy initialization

```
type
  TSingleton = class
  private
    class var FInstance: IFoo;
    class var FFlag: Integer;
    class function GetInstance: IFoo; static;
  public
    class property Instance: IFoo read GetInstance;
  end;

  class function TSingleton.GetInstance: IFoo;
begin
  while FInstance = nil do
  begin
    if TInterlocked.CompareExchange(FFlag, 1, 0) = 0 then
      FInstance := TFoo.Create
    else
      YieldProcessor; // hints the processor it can give control to other threads
  end;
  Result := FInstance;
end;
```

8.3 Writeable instances

The previous examples were dealing with instances that were only initialized once—actual singletons. In such cases, we have options allowing us to avoid locks and protecting the instance while it was used further in code, because it only needs read-only access.

If we want to have a class instance that can also be written, we need to use locks. Whether we will use an exclusive lock or an MREW lock depends on the usage. If instance only occasionally needs to be written, then an MREW lock will be better.

The implementation of a writeable instance is similar to what we have in the “lazy initialization with lock” example, but we also need to write a setter. Because we have a setter, we will not use lazy instantiation for such an instance.

If the class or record contains multiple fields that need to be kept in a consistent state together, then such fields need to be protected by a single lock.

If the class just holds multiple fields that can be used independently, then using multiple locks for each field is possible. Whether or not you will use multiple locks or you protect all fields with a single lock depends on how often they are accessed. A single lock is less performant, but also uses less resources.

Thread-safe writeable class instance

```
type
  TWriteableInstance = class
  private
    class var FInstance: TFooObject;
    class var FLock: TCriticalSection;
    class constructor ClassCreate;
    class destructor ClassDestroy;
    class function GetInstance: TFooObject; static;
    class procedure SetInstance(Value: TFooObject); static;
  public
    class property Instance: TFooObject read GetInstance write SetInstance;
  end;

  class constructor TWriteableInstance.ClassCreate;
begin
  FLock := TCriticalSection.Create;
end;

  class destructor TWriteableInstance.ClassDestroy;
begin
  FInstance.Free;
  FLock.Free;
end;

  class function TWriteableInstance.GetInstance: TFooObject;
begin
  FLock.Enter;
  try
    Result := FInstance;
  finally
    FLock.Leave;
  end;
end;
```

```
class procedure TWriteableInstance.SetInstance(Value: TFooObject);
begin
  FLock.Enter;
  try
    FInstance.Free;
    FInstance := Value;
  finally
    FLock.Leave;
  end;
end;
```

Chapter 9

Core classes

The most basic, core classes in Delphi are thread-safe, as they either don't access any shared state at all, or they do that in a thread-safe manner. Whether their descendant classes will be thread-safe or not solely depends on the code introduced in the derived classes.

It is important to note that this *safety* stems from the mere absence of state, and not any other mechanism. The moment you start deriving from those classes to add state, those derived classes will only be thread-safe if you build additional thread safety mechanisms. If you just declare a few fields or properties and allow their mutation via properties or methods you will have thread-unsafe class—actually, single thread-safe if you are not accessing any unprotected global state.

Thread-safe

- `System.TObject`
- `System.TInterfacedObject`
- `System.TAggregatedObject`
- `System.TContainedObject`
- `System.Classes.TPersistent`
- `System.Classes.TInterfacedPersistent`

Note: `TAggregatedObject`, `TContainedObject` and `TInterfacedPersistent` are thread-safe as long as they are used as intended—as fields inside the owning controller class.

As soon as we step into the realm of other commonly used classes, thread safety is no longer guaranteed. Some classes, like `TThread`, are thread-safe if you follow some basic rules. The rest of those base classes are single thread-safe. They can be used in background threads, but they hold mutable data not protected by any thread safety mechanisms.

Thread-safe - with some additional constraints

- `System.Classes.TThread`

Thread-unsafe - single-thread-safe

- `System.Classes.TBits`
- `System.Classes.TStream`
- `System.Classes.TCustomMemoryStream`
- `System.Classes.TMemoryStream`
- `System.Classes.TPointerStream`
- `System.Classes.TBytesStream`
- `System.Classes.TResourceStream`
- `System.Classes.THandleStream`
- `System.Classes.TFileStream`
- `System.Classes.TBufferedFileStream`
- `System.Classes.TStreamAdapter`

Thread-unsafe - main thread only, for the rest *it is complicated*

- `System.Classes.TComponent`
- `System.Classes.TDataModule`

9.1 Custom classes

While the base classes are thread-safe mostly because they don't hold any state, as soon as you start writing your own classes, the burden of making them thread-safe is on you. Fortunately, most of the classes (not counting visual controls and components) need to be single thread-safe—they need to be usable in background threads, but they don't need to be thread-safe for sharing between threads. As long as you don't access any global unprotected data in those classes or you don't use any code that needs to run in the context of the main thread, custom classes that are descended from thread-safe or single thread-safe classes will automatically be single thread-safe.

When you need to make your classes thread-safe, there are many ways to do so, depending on their purpose and functionality. Not all parts of thread-safe classes need to be thread-safe. It is important that you document how the class can be used in a thread-safe manner and which parts cannot be used simultaneously from multiple threads. Documentation not only provides

guidance for the developers (even if it is only you) that will use those classes without having to waste time digging through the implementation and deducing the appropriate usage, but it is also important for finding concurrency bugs. If someone writes code that breaks in multi-threading and the code is written according to the specification, then this is due to a bug in the used class that needs fixing. On the other hand, if the code using the class doesn't follow the usage guidelines, it means that the code using the class needs fixing.

The simplest way of achieving thread safety is having immutable data. Immutable classes and records can be shared safely between threads without the need for additional protection. By sharing, in this context, I am talking about the thread safety of the content, not the references to that content—not the variables.

Declaring immutable classes or records is simple. All data should be initialized through the constructor, and should be accessible only through read-only properties. Methods should not mutate any data, not even temporarily.

Thread-safe immutable data type

```
type
  TFoo = class
  private
    FValue: Int64;
  public
    constructor Create(AValue: Int64);
    property Value: Int64 read FValue;
  end;

  constructor TFoo.Create(AValue: Int64);
begin
  FValue := AValue;
end;
```

For more complex data, having fully immutable data types can be impractical, especially when it comes to serializing data to and from other formats. Handling user input and changes is also more complicated with fully immutable data. Because of that, classes are often designed to be mutable, to simplify handling the data when being accessed by a single thread, and when the data is being used by multiple threads, those threads are not allowed to mutate the data, only read it. If there is writing involved, then such processing requires protection in the broader scope.

A commonly used approach when dealing with data that is treated as immutable is cloning the instance and then applying the necessary changes to the clone. **TPersistent** has the **Assign** method, which can be extended in descendant classes and can be used for cloning. Another commonly used approach is implementing a dedicated **Clone** method used to create a copy of the data. In general, there are no functional differences between the two, and opting for the

former or the latter is merely a matter of opinion. There may be situations where both methods are used, with slight variations in behavior, and in such cases, it is prudent to document the differences in functionality and appropriate usage.

If the class is immutable, then cloning the instance will create an exactly identical instance, which cannot be modified after cloning, thus defeating its purpose. If the data can stay the same, then it doesn't need to be cloned in the first place.

```
function TFoo.Clone: TFoo;
begin
  Result := TFoo.Create(Value);
end;
```

To have a fully-functional class where cloning is useful, we need to have a mutable class. We can still have a constructor that will also fully initialize data in the instance, but we can also have a parameterless constructor and initialize data after construction. As long as the original instance is not otherwise mutated in another thread during cloning, creating the clone will be a thread-safe operation, as it will not mutate the original instance.

Single thread-safe cloneable data type

```
type
  TFoo = class
  private
    FValue: Int64;
  public
    constructor Create; overload;
    constructor Create(AValue: Int64); overload;
    function Clone: TFoo;
    property Value: Int64 read FValue write FValue;
  end;

  constructor TFoo.Create;
  begin
  end;

  constructor TFoo.Create(AValue: Int64);
  begin
    FValue := AValue;
  end;
```

Depending on the constructors used, there are two possible variants of the clone method:

```
function TFoo.Clone: TFoo;
begin
  Result := TFoo.Create(Value);
end;
```

or

```
function TFoo.Clone: TFoo;
begin
  Result := TFoo.Create;
  Result.Value := Value;
end;
```

Note: The thread safety of `Assign`, `AssignTo`, and other cloning methods depends on the particular class and code within those methods. Put simply, some cloning methods can temporarily change the state of the original while copying it, or use other thread-unsafe code. Such cloning methods are not thread-safe. Before you use cloning as a way of achieving the thread safety of otherwise read-only data, make sure that the cloning methods used are actually thread-safe.

Both variants of the `TFoo.Clone` method don't mutate the original instance and can be safely used to create clones, if the original instance is not mutated by any other code.

Besides immutability, another way of protecting the data is to use locks to keep the data in a consistent state. Including a lock within each and every object instance is not commonly done because it is not only wasteful, but also often unsuccessful at solving the logical issues that arise while processing such data.

For instance, if you have a collection holding some numerical data, which you want to summarize or perform some other calculations with, locking the individual object instance and modifying some value will keep the data in that instance in a consistent state, but it can yield incorrect calculation results. You will need to keep the whole collection protected while performing calculations, instead of protecting each individual piece of data.

Because of that, locks are commonly used for locking larger parts of data. Sometimes locking on a single-collection level is enough, and sometimes we need to apply locks to an even broader data context. More about using locks can be found in the *Collections* chapter.

Chapter 10

Threads

Thread-safe - with some additional constraints

The `TThread` class from the `System.Classes` unit provides easy-to-use threading support. It can be used either as a base ancestor class to implement custom threads that will run particular code in an overridden `Execute` method, or it can be used to create anonymous threads through its `Class` method, that will execute code passed into it as an anonymous method or an ordinary method parameter.

In both use cases, some parts (methods and properties) of the `TThread` class are thread-safe and can be used simultaneously from multiple threads, and some parts can only be safely used during the initialization and shutdown processes. In other words, you cannot safely change some configuration properties while the spawned background thread is already running.

Thread-safe

Thread-safe class methods that can be called from any thread:

- `CreateAnonymousThread`
- `Synchronize`
- `Queue`
- `ForceQueue`
- `RemoveQueuedEvents`

Thread-safe class methods and properties that apply on, or return information about, the current thread's context:

- `NameThreadForDebugging`
- `Current`

- `CurrentThread`
- `CheckTerminated`
- `SpinWait`
- `Sleep`
- `Yield`
- `SetReturnValue`

Thread-safe class methods and properties that return global system information or just operate on passed parameters:

- `GetSystemTimes`
- `ProcessorCount`
- `IsSingleProcessor`
- `GetCPUUsage`
- `GetTickCount`
- `GetTickCount64`
- `IsTimeout`

Thread-safe methods and properties:

- `Terminated`
- `Terminate`
- `ThreadID`
- `WaitFor` - cannot be called from within the thread's own `Execute` method
- `ExternalThread`
- `Started`
- `Finished`
- `Handle`

Thread-safe properties for reading, can be written only before the thread starts running

- `Priority`
- `Policy`
- `Suspended` - the ability to suspend an already-running thread is meant only for debugging purposes (debugger) and can cause deadlocks or undefined behavior if used in an application

Thread-unsafe - main thread only

- `OnSynchronize`

Thread-unsafe - safe only under certain conditions

Configuration-related methods and properties that can be safely set before the thread starts running. Under certain conditions, some of those can also be changed during thread shutdown.

- **Start** - can be safely used from any thread, for starting a thread that is constructed in a suspended state
- **FreeOnTerminate** - can be safely set before the thread starts running, any changes after that point are the fastest way to shoot yourself in the foot
- **OnTerminate** - can be safely set before the thread starts running, or from within the thread's **Execute** method
- **FatalException** - can be used only after thread exits the **Execute** method

10.1 FreeOnTerminate

Most of the methods and properties in the **TThread** class are thread-safe, but the remaining few are the nasty ones that cause most of the trouble in real-life code. And among those, by far the most dangerous property is **FreeOnTerminate**.

When it comes to **FreeOnTerminate**, the source of unsafety is not in the variable itself—it is a boolean and assigning a boolean is an atomic operation—but the consequences of such an assignment: In other words, what that variable controls, and its purpose. This is also another good example of how atomicity does not imply thread safety.

When **FreeOnTerminate** is set, the thread object instance will be destroyed automatically on thread termination. Once such a *self-destroying* thread starts running, you cannot change **FreeOnTerminate** back to false from external code, because you are basically operating on a potentially invalid object instance.

Not only can you not touch **FreeOnTerminate**, but you cannot safely do anything else with such a thread instance. And this is quite a common point of failure. One of the problems with such code is that the thread needs some time to start, and then some more time to execute and shutdown before it self-destructs. Quite often running such code and accessing the thread instance will seemingly work properly only because the thread instance is accidentally still alive. But such code can also randomly fail.

FreeOnTerminate can be set inside a custom thread constructor or factory function, so you need to be careful when handling such threads. One such factory function is **TThread.CreateAnonymousThread** which constructs a self-destroying thread instance in suspended mode, to allow additional setup and configuration before the thread is started with the **Start** method.

A very common usage pattern for anonymous threads is constructing and starting a thread without assigning it to a variable, as the thread will self-destruct after executing code in the anonymous method:

Thread-safe - correct code

```
TThread.CreateAnonymousThread(  
  procedure  
  begin  
    // do something  
  end).Start;
```

But quite often, that is not enough, and some additional configuration is required, like setting an **OnTerminate** event, and the code introduces a thread variable in order to reference such a thread. If code does not reference that variable after calling **Start**, everything will work fine.

Thread-safe - correct code

```
var  
  Thread: TThread;  
begin  
  Thread := TThread.CreateAnonymousThread(  
    procedure  
    begin  
      // do something  
    end);  
  Thread.OnTerminate := OnThreadTerminate;  
  Thread.Start;  
  // after this point Thread variable must not be used
```

Because the thread will not be destroyed while code in the anonymous method is running, it is also safe to access the thread variable from within the anonymous method.

Thread-safe - correct code

```
var  
  Thread: TThread;  
begin  
  Thread := TThread.CreateAnonymousThread(  
    procedure  
    begin  
      Thread. ... // access thread variable  
    end);  
  Thread.Start;
```

So far so good.

Even though at times you can find bad code in the wild, where a self-destroying thread variable will be referenced after the thread is started, if this is only due to misplacing a few lines of code, such trivial mistakes can be easily spotted and fixed. Like, for instance, in the following example, where the **OnTerminate** event is assigned after calling **Start**.

Thread-unsafe - incorrect code

```
var
  Thread: TThread;
begin
  Thread := TThread.CreateAnonymousThread(
    procedure
    begin
      // do something
    end);
  Thread.Start;
  // unsafe access to Thread instance after thread is started
  Thread.OnTerminate := OnThreadTerminate;
```

Self-destroying threads are really neat, as you don't have to manage their memory, and for simple, short-running tasks, using fire-and-forget threads is rather appealing. They are also the easiest way to write some proof-of-concept code. There is nothing wrong with that. The problems arise when such simple or temporary code is being enhanced, and requires handling the thread after it started running. For instance, waiting for the thread to finish when the user closes the form.

And a common mistake in such cases is that when the thread variable needs to be moved from local scope to a broader scope, developers try to dynamically adjust behavior at runtime, depending on certain conditions, instead of switching to manually managed threads. And at the same time, either it is forgotten that one should not touch self-destroying thread instances after the thread started running, or there are some more subtle race conditions in the code.

The simplest, and usually the best way of resolving such *you can't have your cake and eat it* issues is simply opting for manual memory management. While this is the fastest and the cleanest way, for one reason or another, there is plenty of real-life code lurking around where developers took the longer path, trying to keep automatic memory management at all costs, where the final cost is thread-unsafe and often fairly convoluted code.

Another particular issue with self-destroying threads is cleanup on application shutdown. For manually managed threads you need to call **Free** at some point, and this will also terminate and wait for thread completion. In other words, you will have controlled cleanup and destruction of a thread during application shutdown.

On the other hand, self-destroying threads will keep running until they are just killed by the OS, because their owning (application) process exited. There is no proper waiting and cleanup for those threads in the RTL, due to their self destroying nature. Such a thread can wreak havoc

during shutdown if it accesses any shared resources that might be gone at that time. Also, the possibility of being killed at any point makes such threads unsuitable for the execution of any operation which cannot be arbitrarily interrupted at a random point.

Having said that, there is a range of scenarios where it is possible to have some additional handling of self-destroying threads, after the thread started running, without compromising thread safety nor code readability. However, neither of those solutions solves the application shutdown cleanup.

Note: For brevity, further examples will not handle thread re-entrancy scenarios, since the focus is showing thread unsafety issues caused by `FreeOnTerminate`. If methods that start the thread are run more than once, before the thread fully shutdowns, the application can misbehave.

The following example tries to handle thread cancellation in combination with a self-destroying anonymous thread. The problematic code is in `StopBtnClick` with the `if Assigned(Thread)` check, where the code running in the background thread could set the `Thread` variable to `nil` after we have successfully passed the `if ...` check.

Thread-unsafe - incorrect code

```
type
  TMainForm = class(TForm)
  ...
private
  Thread: TThread;
end;

procedure TMainForm.StartBtnClick(Sender: TObject);
begin
  Thread := TThread.CreateAnonymousThread(
    procedure
    begin
      try
        // do something
      finally
        Thread := nil;
      end;
    end);
  Thread.Start;
end;

procedure TMainForm.StopBtnClick(Sender: TObject);
begin
  if Assigned(Thread) then Thread.Terminate;
end;
```

If we make sure that the code assigning `nil` and checking for `nil` both run in the context of the same thread, we can make that code thread-safe. There are two ways to achieve that. One is calling `TThread.Synchronize` around the `nil` assignment in `finally` block, and another one is moving the `nil` assignment to the thread's `OnTerminate` event that also runs in the context of the main thread. That way, `StopBtnClick` will be able to run completely before `OnTerminate` runs, or vice versa.

It is important to note that this kind of code will only work if the logic (code) starting and stopping the thread is also running in the context of the main thread. If you have a scenario where either one of those runs in the context of some other thread, such code will not be thread-safe. If you have some other code using a similar pattern and `nil` checks, you must also ensure that such code can only run in the context of the main thread.

Thread-safe - correct code

Solution 1: Synchronization

If you apply this solution, you cannot replace `TThread.Synchronize` with `TThread.Queue`, as the queued procedure can run after thread is already destroyed, and the code in `StopBtnClick` could be operating on a dangling pointer even though all code will run in the main thread.

```
procedure TMainForm.StartBtnClick(Sender: TObject);
begin
  Thread := TThread.CreateAnonymousThread(
    procedure
    begin
      try
        // do something
      finally
        TThread.Synchronize(nil,
          procedure
          begin
            Thread := nil;
          end);
      end;
    end);
  Thread.Start;
end;
```

Solution 2: Using the **OnTerminate** event handler

```
procedure TMainForm.StartBtnClick(Sender: TObject);
begin
  Thread := TThread.CreateAnonymousThread(
    procedure
    begin
      // do something
    end);
  Thread.OnTerminate := OnThreadTerminate;
  Thread.Start;
end;

procedure TMainForm.OnThreadTerminate(Sender: TObject);
begin
  Thread := nil;
end;
```

The previous examples only added the ability to handle an already-running self-destroying thread. Trying to change a running thread's memory management is another story and it closely resembles fixing an airplane in flight.

The base example code is the same, but let's say you don't want to just call **Terminate** on such thread, but you want to change how it is managed for one reason or another.

Thread-unsafe - incorrect code

```
procedure TMainForm.StartBtnClick(Sender: TObject);
begin
  IsClosing := False;
  Thread := TThread.CreateAnonymousThread(
    procedure
    begin
      // do something
    end);
  Thread.OnTerminate := OnThreadTerminate;
  Thread.Start;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  ...
  if Assigned(Thread) then
    begin
```

```
    IsClosing := True;
    // freeing a thread will also terminate and wait for completion
    Thread.Free;
    Thread := nil;
  end;
end;

procedure TMainForm.OnThreadTerminate(Sender: TObject);
begin
  if IsClosing then
    Thread.FreeOnTerminate := False
  else
    Thread := nil;
end;
```

The above example tries to wait on a thread if the user closes the form. In order to do that it tries to switch the thread's memory management. But, at the point when the **OnTerminate** event handler is called, it is too late to do that. Thread is already on its way to being automatically released, and calling **Thread.Free** will cause a double free.

On the other hand, if you want to do the opposite and auto-release a manually managed thread at that point, you will also fail, because it is too late to initialize automatic release, and the thread instance will leak.

We already have a blueprint for solving such situations. Instead of fiddling with thread memory management, we can just terminate and wait on a thread instance if it is assigned.

```
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  ...
  if Assigned(Thread) then
    begin
      Thread.Terminate;
      Thread.WaitFor;
    end;
end;

procedure TMainForm.OnThreadTerminate(Sender: TObject);
begin
  Thread := nil;
end;
```

Problem solved!

But is it, really?

You happily run the above code, and suddenly an error box pops up with this message: *Thread Error: The handle is invalid (6)*.

The core of the issue here is there is huge difference between calling `Thread.Terminate` and `Thread.WaitFor`. Terminating a thread basically just flips a boolean and calls the virtual `TerminatedSet` method, whose default implementation is empty. No matter when you call `Terminate`, as long as the instance is not deallocated, nothing bad will happen (unless you have some more complex code with side-effects in `TerminatedSet`).

On the other hand, waiting for a thread is a different story, because while you are waiting, the thread will complete its work at some point, and in self-destroying threads, that will also initiate shutdown and destruction. During that process which will happen in the context of the background thread, the thread handle will be closed, and in the waiting loop, `MsgWaitForMultipleObjects` will at some point fail because the thread handle has become invalid.

The same thing will happen if you don't call `Terminate` on a thread, and just call `WaitFor`. Now, on the Windows platform you can currently add a `try...except` handler around that and eat up the exception, but on other platforms, that code will cause undefined behavior. It might work a few times, then it might deadlock.

Problem is that waiting initiates complex logic that expects the thread instance to be fully operational for the duration of the process. It does not work properly with self-destroying threads, because such threads break one important promise: A self-destroying thread instance **must not** be referenced by outside code once it starts running. We have tried to circumvent that rule, by setting the instance to `nil` in the `OnTerminate` event handler, but it was a hack with serious limitations.

This brings us back to my original advice—if you need to handle a thread once it starts running, don't use self-destroying threads. At some point, that approach will just blow up.

10.2 Starting a thread

Each thread instance can be constructed in suspended mode, or it can automatically start after it is constructed. Automatic start is triggered inside the `AfterConstruction` method, so you should perform all initialization inside the constructor chain, or if really necessary, you can add some code in `AfterConstruction`, but in that case, you need to make sure you call `inherited AfterConstruction` after your initialization code.

During construction, any code inside the constructor (including in derived `TThread` classes) and all inherited constructors will run in the context of the thread that initiated thread construction and before the thread is started (even for automatically started ones). So you don't have to be concerned about the thread safety of the code that runs there.

The only exception to that rule is code that has thread affinity and needs to run in the context of the particular thread to run correctly. For instance, you cannot call `CoInitialize` to initialize COM inside the constructor, because it needs to run in the context of the not-yet-started

background thread, and as such it must be called from within the thread's **Execute** method. When anonymous threads are concerned, such code needs to run within the anonymous method or object method passed to **CreateAnonymousThread**.

Because non-suspended threads are started from **AfterConstruction**, you don't need to worry about the order in which the constructors are executed. In other words, you can place the **inherited** call anywhere within your custom constructor. But, don't forget to call **inherited**, because calling the **TThread** constructor is vital for properly initializing a thread. You can also freely change thread configuration inside constructors(s), like modifying **FreeOnTerminate** or similar properties.

For instance, both variants of the following thread constructor are correct, and will construct and assign the **TData** instance before the thread starts running:

Thread-safe

```
constructor TMyThread.Create(CreateSuspended: Boolean);
begin
  inherited;
  FData := TData.Create;
end;
```

```
constructor TMyThread.Create(CreateSuspended: Boolean);
begin
  FData := TData.Create;
  inherited;
end;
```

Of course, once the thread starts running, any data accessed from that background thread context needs to be protected if it is shared with other threads while the thread is running.

Once a thread is constructed, if it is constructed in suspended state, at some point it needs to be started with the **Start** method. You **must not** call **Start** on the same thread more than once.

Delphi also has two deprecated methods, **Suspend** and **Resume**, and while they are still present for backward compatibility, do not use them in new code, and if possible—legacy code logic can be convoluted at times, and just removing those methods can break the code—remove them from existing code.

Suspending a thread once it is initially started is not recommended, because the thread can be interrupted at any moment, completely outside of your control. That means the thread can lock some resource and be put to sleep, holding that resource for indeterminate amount of time, eventually causing deadlocks and other misbehaviors.

Originally, **Suspend** and **Resume** were intended to be used by debuggers, but unfortunately, many

developers didn't properly understand the full implications of their usage, and quickly started abusing them in regular code.

Please note that you cannot use the boolean flag `Started` in order to determine whether thread is already running or not in order to `Start`, as it is not thread-safe.

`Started` and similar boolean flags representing some state that will be changed only once after initialization stage are safe to read as their access is atomical, but you need to be careful, because you cannot use them as an indication that something has not happened, as they can only tell you something *has* happened.

In other words, `Started` will be set in the context of the background thread just before the `Execute` method is called, so it is possible that the above `if` check will pass while the thread is already running, but has not yet had chance to flip the flag.

Thread-unsafe - incorrect code

```
var
  Thread: TThread;
begin
  ...
  if not Thread.Started then
    Start;
  ...

```

10.3 Order of destruction

The destructor runs in the context of a single thread, the one that initiated the destruction of the thread object instance. Similarly to construction, finalization code with thread affinity needs to run inside the background thread's context—the thread's `Execute`, or the anonymous or object methods passed to `CreateAnonymousThread`.

While code in the destructor runs in the context of a single thread, the one that initiated the destruction process, the thread safety of destructor code is a bit different from that of the constructor. Namely, when the destructor is explicitly called, the background thread may still be running, and in such situations, the `TThread.Destroy` destructor initiates the thread shutdown process—it will terminate and wait for the background thread to finish.

And here is the catch: If you are deriving a new class from `TThread` and overriding its destructor, you need to pay attention to the order of destruction, because if the background thread is still running, you can bump into race conditions.

For instance, if your thread class constructs and destroys some object, and you destroy it before you have called the inherited destructor, you will destroy the object while the background thread is still using it.

Thread-unsafe - incorrect code

```
destructor TMyThread.Destroy;
begin
  FData.Free;
  inherited;
end;

var
  Thread: TMyThread;
begin
  Thread := TMyThread.Create;
  ...
  Thread.Free;
end;
```

The solution is simple: Just call the inherited destructor first, and then write your finalization code.

Thread-safe - correct code

```
destructor TMyThread.Destroy;
begin
  inherited;
  FData.Free;
end;

var
  Thread: TMyThread;
begin
  Thread := TMyThread.Create;
  ...
  Thread.Free;
end;
```

An alternative solution for this problem is waiting for the thread to finish before you release the thread instance. However, this kind of solution is rather fragile, as you need to remember to call `WaitFor` explicitly everywhere before you call `Free`. Also, this approach does not work for self-destroying threads, as you will not be able to wait for such a thread.

Thread-safe - correct code, use with caution

```
destructor TMyThread.Destroy;
begin
  FData.Free;
  inherited;
end;

var
  Thread: TMyThread;
begin
  Thread := TMyThread.Create;
  ...
  Thread.WaitFor;
  Thread.Free;
end;
```

10.4 Custom thread data

One of the purposes of implementing custom thread classes is packing necessary data with the thread instance. There are many ways such data can be used and protected. If the data is passed to or constructed inside the constructor and not accessed from other threads while the thread is running, then such data doesn't need any additional protection. The same is valid for data captured by and used inside anonymous threads or tasks from the Parallel Programming Library. If the data is not accessed from other threads while the background thread is using it, no additional protection is necessary.

Once the thread is finished, such data can be safely used. This can be done from the **OnTerminate** event handler that runs in the context of the main thread, or from the external context if we use **WaitFor** to make sure the thread has finished using the data.

For instance, we can declare a public **Data** property in the custom thread class, and as long as we don't touch that public property while the background thread is using it, such code will be thread-safe. We only need to keep in mind that assigning reference types doesn't create a copy of the data, and if we want to mutate the contents of such data passed to a thread, then we either need to manually make a copy, or we need to ensure that other threads don't access it while the background thread is running.

```
TMyThread = class(TThread)
private
  FData: TData;
public
  ...
  property Data: TData read FData write FData;
end;
```

It is not uncommon for you to need to pack two different sets of data: one for input and one for output. Input data that will only be read by the background thread can be safely used under more relaxed rules. If you are only taking a reference to such data, you still need to ensure that data will be alive during the thread lifetime, and that it will not be mutated by other threads during that period.

If we need to access any data from multiple threads, we either need to synchronize such code with the main thread, making sure only the main thread has access to that data at any point in time, or use locks or other synchronization objects to coordinate access between different threads.

Chapter 11

Streams

Thread-unsafe - single-thread-safe

Streams can be freely constructed and used in background threads. They can also be constructed in one thread and then handed over to another, but once you make such a handover, the first thread is not allowed to use the instance while the second thread is using it. You don't need any additional protection mechanisms to handle such scenarios.

You can also fully share the same stream instance between threads, but in such cases, you need to protect simultaneous access from multiple threads. Unlike some other classes that may be more easily protected just by adding some locking mechanism around their individual methods, streams are more complex, and just protecting reading or writing access is not enough because any reading or writing also changes the stream's **Position** property. Of course, writing can also change the stream's **Size**.

If you want to share a stream between threads, you will have to take those specifics into account, and will have to adjust the **Position** property every time before you start your reading or writing sequence. Because the **Position** property also changes during reading, you cannot use MREW locks on streams, and they can only be protected by exclusive locks.

When a particular stream class is just a wrapper around some external resource that has additional access modes—like file streams—you also need to prevent simultaneous usage of the same stream in exclusive access mode. Even if you use two separate stream instances, such attempts will fail regardless of the threads involved.

However, if all threads or other processes will only read from such an external seekable stream, then you can construct separate stream object instances using the same external stream. If each thread has own stream instance with a separate handle, you can avoid unnecessary locking while reading from the stream.

Please note that thread-safe reading is only possible if each stream object instance operates on a different OS-provided stream handle, which is true for the **TFileStream** implementation.

Operations on the same handle from multiple threads is not thread-safe, because seeking and reading can interfere on the OS level.

Handling non-seekable streams simultaneously from multiple threads is not possible without conflicts.

The following example shows how to protect reading a buffer from a stream. You can also protect multiple, related reads within the same locking instruction, to prevent unnecessary seeking.

Thread-safe

```
procedure SafeReadStreamBuffer(AStream: TStream; ALock: TCriticalSection;
  APosition, ACount: Integer; var Buf);
begin
  ALock.Enter;
  try
    AStream.Position := APosition;
    AStream.Read(Buf, ACount);
  finally
    ALock.Leave;
  end;
end;

procedure SafeReadStreamBuffers(AStream: TStream; ALock: TCriticalSection;
  APosition, ACount1, ACount2: Integer; var Buf1, Buf2);
begin
  ALock.Enter;
  try
    AStream.Position := APosition;
    AStream.Read(Buf1, ACount1);
    AStream.Read(Buf2, ACount2);
  finally
    ALock.Leave;
  end;
end;
```

The following example uses an MREW lock, and is requesting read access for reading from the stream. However, read access is not enough to ensure thread safety, because assigning a value to **Position** or calling the **Seek** method modifies the internal state of the stream object, and it is actually a mutating operation that requires exclusive (or write) lock.

The **Read** method alone, also changes the internal state of the object and requires exclusive access. The **Read** and **Write** method names only tell you what kind of operation is requested on the stream, and the name alone is not enough to determine whether the internal state of the object will mutate or not.

A similar principle applies to other classes. Just because some method implies a read operation that does not automatically mean that it will not change the inner state of the object instance. You need to know what the method actually does in order to correctly assess whether you can use MREW locks and read access for protection or not.

Thread-unsafe - incorrect code

```
procedure SafeReadStreamBuffer(AStream: TStream; ALock: TLightweightMREW;
  APosition, ACount: Integer; var Buf);
begin
  ALock.BeginRead;
  try
    // This line mutates internal stream state - it is a write operation
    AStream.Position := APosition;
    // This line mutates internal stream state - it is a write operation
    AStream.Read(Buf, ACount);
  finally
    ALock.EndRead;
  end;
end;
```

Even an exclusive lock applied on the **Read** operation alone will not be thread-safe, as the **Read** operation will change the stream's position, and when a particular thread enters the lock, it is possible that it will start reading the stream from the wrong position.

Protecting the **Position** property or the **Seek** method separately from the **Read** operation will not solve the thread safety problem, because again, it is possible that another thread acquires the lock in the middle of a particular seek/read sequence and change the position halfway through, which will cause reading the wrong data.

Thread-unsafe - incorrect code

```
procedure SafeSeekStream(AStream: TStream; ALock: TCriticalSection;
  APosition: Integer);
begin
  ALock.Enter;
  try
    AStream.Position := APosition;
  finally
    ALock.Leave;
  end;
end;
```

```
procedure SafeReadStreamBuffer(AStream: TStream; ALock: TCriticalSection;
  ACount: Integer; var Buf);
begin
  ALock.Enter;
  try
    AStream.Read(Buf, ACount);
  finally
    ALock.Leave;
  end;
end;
```

Besides using additional locks, we can also protect streams and any other object instance with **TMonitor** static functions that use an internal object field—part of every object instance—for controlling exclusive access to the object. This enables us to create simpler thread-safe access functions, that don't require maintaining separate locks, for any seekable stream, as long as we keep in mind that all access to such a stream needs to be called through those safe functions.

Thread-safe

```
procedure SafeReadStreamBuffer(Stream: TStream; Position, Count: Integer; var Buf);
begin
  TMonitor.Enter(Stream);
  try
    Stream.Position := Position;
    Stream.Read(Buf, Count);
  finally
    TMonitor.Exit(Stream);
  end;
end;
```

If you want to write safer and more foolproof code, you will have to create a thread-safe wrapper class around a private stream field that can only be accessed through thread-safe read and write methods.

Generally, sharing streams across threads is not a common practice, because only one thread is allowed to work on the stream at any moment. If threads perform most of their work inside a stream lock, that will create contention, and you will just have a bunch of threads sitting and waiting for other threads to finish their job. If accessing the stream is only a very small part of the job the thread is doing, then sharing might be acceptable.

Also, accessing external streams is an I/O-bound operation, which will not run faster just because you have thrown more threads—more CPU power—at it. On the contrary, multiple threads can slow things down even further, depending on the physical characteristics of the device holding the stream.

The **TStreamAdapter** class provides an implementation for the **IStream** interface, and it is just a wrapper around an external **TStream** instance passed as a parameter in its constructor. **TStreamAdapter** can be constructed in two ways: it can take ownership over the passed stream instance, and will free the stream in its destructor, or it can just reference the stream without interfering with its memory management.

Regardless of the chosen ownership model, you are not allowed to pass the same stream instance to two different adapters and use them at the same time, as such code will not be thread-safe.

Thread-unsafe - incorrect code

```
var
  Stream: TStream;
  Adapter1: IStream;
  Adapter2: IStream;
begin
  ...

  Adapter1 := TStreamAdapter.Create(Stream, soReference);
  Adapter2 := TStreamAdapter.Create(Stream, soReference);

  TThread.CreateAnonymousThread(
    procedure
    begin
      // use Adapter1
      Adapter1. ...
    end).Start;

  TThread.CreateAnonymousThread(
    procedure
    begin
      // use Adapter2
      Adapter2. ...
    end).Start;
end;
```


Chapter 12

Collections

The base collection classes are generally thread-unsafe. However, there are some thread-safe collection types that protect collection integrity and allow modifications when the same collection instance is used across multiple threads. Of course, that does not mean that the collection reference itself is thread-safe for writing.

The thread safety or unsafety of the collection does not imply anything about the thread safety of the stored items individually. How thread-safe or unsafe each item is depends solely on the item type and should be determined independently from the collection itself. For instance, you can maintain a thread-safe collection of objects, but the thread safety of each stored object instance depends on whether you can modify that particular object's state in a thread-safe manner or not.

Why are collections generally unsafe?

Collections are complex structures when it comes to thread safety. The simplest implementation is a dynamic array. Adding to and removing items from a dynamic array is not a thread-safe operation because it consists of multiple steps, where multiple threads can easily cause unintended interactions.

Imagine one thread modifying a dynamic array by removing items, while another thread is iterating through the same array. As soon as an item is removed, the item count will no longer be the same, and the first thread accessing the item at the now-invalid item index will either raise a range exception, crash, or cause other undefined behavior.

There are several ways to make collections thread-safe: immutability, where every modification creates a new copy; synchronization objects; or implementing lock-free data structures. Achieving collection thread-safety always comes with a performance penalty or significantly increased code complexity. Since collections are widely used data structures, most collection implementations are not thread-safe to avoid unnecessary penalties in all code. More precisely, they are single thread-safe. They can be freely used in background threads, and even shared between threads, as long as there is no possibility of simultaneous write access. When there is a need

for thread safety, where threads will modify the collection, it is rather easy to add a protection layer around an existing collection type or use a special thread-safe collection.

The following example shows a simple dynamic array that is simultaneously accessed and modified from three threads—two would be enough to break thread safety, but with three, the breakage is easier to reproduce. The first thread is adding items to the array, the second is removing items, and the third is iterating. Running the code with the debugger attached will break sooner or later with a “Range check error”. The debugger is needed to show the exception, because it will be raised inside the thread and it will just break thread execution, but it will not be displayed to the end user. In a real-life situation, you can use `try...except` blocks inside the thread to capture and appropriately handle or show exceptions. Of course, you can also handle exceptions in the `TThread.OnTerminate` event handler.

Thread-unsafe - incorrect code

```
{$R+} // turn on range checking
procedure TMainForm.ButtonClick(Sender: TObject);
var
  List: array of Integer;
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      i, Len: Integer;
    begin
      for i := 0 to 1000 do
        begin
          Len := Length(List);
          SetLength(List, Len + 1);
          List[Len] := Len;
        end;
    end).Start;

  TThread.CreateAnonymousThread(
    procedure
    var
      i, Len: Integer;
    begin
      for i := 0 to 100 do
        begin
          Sleep(10);
          Len := Length(List);
          if Len > 0 then
            SetLength(List, Len -1);
        end;
    end).Start;
```

```
    end;
end).Start;

TThread.CreateAnonymousThread(
procedure
var
  i, Idx: Integer;
begin
  for i := 0 to 1000 do
    begin
      for Idx := 0 to High(List) do
        OutputDebugString(PChar(List[Idx].ToString));
    end;
  end).Start;
end;
```

In order to protect the above `List`, we would need to add some kind of locking mechanism to avoid conflicts. In this case, because `List` is a local variable, and handling thread completion would be too complex for this simple example, we will use an interface-based lock instead of the commonly used critical section, so its memory will be automatically handled. When using locks to protect instance(s), you must make sure that the lock lifetime will be longer than the lifetime of the protected instance(s). A single lock can protect multiple data instances if their functionality is related and they need to be consistently protected as a whole.

Thread-safe - correct code

```
{$R+} // turn on range checking
procedure TMainForm.Button2Click(Sender: TObject);
var
  List: array of Integer;
  Lock: IReadWriteSync;
begin
  Lock := TSimpleRWSync.Create;
  TThread.CreateAnonymousThread(
    procedure
    var
      i, Len: Integer;
    begin
      for i := 0 to 1000 do
        begin
          Lock.BeginWrite;
          try
            Len := Length(List);
```

```
        SetLength(List, Len + 1);
        List[Len] := Len;
    finally
        Lock.EndWrite;
    end;
end;
end;

TThread.CreateAnonymousThread(
procedure
var
    i, Len: Integer;
begin
    for i := 0 to 100 do
    begin
        Sleep(10);
        Lock.BeginWrite;
        try
            Len := Length(List);
            if Len > 0 then
                SetLength(List, Len - 1);
        finally
            Lock.EndWrite;
        end;
    end;
end).Start;

TThread.CreateAnonymousThread(
procedure
var
    i, Idx: Integer;
begin
    for i := 0 to 1000 do
    begin
        Lock.BeginRead;
        try
            for Idx := 0 to High(List) do
                OutputDebugString(PChar(List[Idx].ToString));
        finally
            Lock.EndRead;
        end;
    end;
end).Start;
end;
```

The Delphi RTL offers a variety of collection classes that are commonly implemented as wrappers around dynamic arrays, offering more user-friendly APIs and extended functionality. The thread safety of such collections can be achieved by using locks. Collections that don't use any locking mechanism are not thread-safe, for the same reasons a dynamic array is not thread-safe. Adding, removing, or exchanging items is a complex operation, and if multiple threads can access the collection while those operations are taking place, unintended interaction is inevitable.

While only a few of the provided collection classes are thread-safe, allowing that same collection instance to be safely used from multiple threads, other collection classes are single thread-safe. They can be used in background threads, but they need additional protection if they are simultaneously accessed from multiple threads.

Thread-safe

- `System.Classes.TThreadList`
- `System.Classes.TInterfaceList`
- `System.Generics.Collections.TThreadList<T>`
- `System.Generics.Collections.TThreadedQueue<T>`

Single thread-safe

- `System.Classes.TList`
- `System.Classes.TStrings`
- `System.Classes TStringList`
- `System.Classes.TCollection`
- `System.Classes.TOwnedCollection`
- `System.Contnrs.TObjectList`
- `System.Contnrs.TComponentList`
- `System.Contnrs.TClassList`
- `System.Contnrs.TOrderedList`
- `System.Contnrs.TStack`
- `System.Contnrs.TObjectStack`
- `System.Contnrs.TQueue`
- `System.Contnrs.TObjectQueue`
- `System.Contnrs.TCustomBucketList` and its descendants
- `System.Generics.Collections.TList<T>`
- `System.Generics.Collections.TQueue<T>`
- `System.Generics.Collections.TStack<T>`
- `System.Generics.Collections.TDictionary<K,V>`
- `System.Generics.Collections.TObjectList<T>`
- `System.Generics.Collections.TObjectQueue<T>`
- `System.Generics.Collections.TObjectStack<T>`
- `System.Generics.Collections.TObjectDictionary<K,V>`

12.1 Making a thread-safe variant of a thread-unsafe collection

Quite often, you will need a thread-safe variant of some collection type, and there is none provided out of the box.

One of the possible solutions is similar to the one shown in the protected array example. You can add an external lock for the collection you need to protect, and all code that needs to access that collection must go through the locking mechanism. But that approach is rather error-prone, as it is easy to forget to call the locking code. Besides that, you need to separately manage the lifetimes of the collection and the lock.

With a collection object, compared to dynamic arrays, you do have another option, and you can use the collection object itself as a lock with `System.TMonitor`. That simplifies lifetime management, but doesn't solve the problem of forgetting to lock the collection every time you use it.

Thread-safe collection classes wrap the lock and the collection container (usually array) together. That makes protection easier and safer, and solves lifetime management. You can apply a similar principle and write a thread-safe wrapper around the whole collection, or any other class for that matter. Which particular locking mechanism you will use depends solely on your use case. There are upsides and downsides to each one; also, their behavior can change between versions, for better (usually) or for worse (one should never forget about that possibility, too).

Delphi 10.4.1 Sydney introduced a very lightweight, cross-platform, multiple readers/single writer lock, `TLightweightMREW`, that is a thin wrapper record upon an OS-provided locking mechanism. While it is not recursive and does not support upgrading a reader lock to a writer, it is very suitable as a lightweight collection locking mechanism, as it does not require additional heap allocations. I will use it in the following examples. Another advantage of packing your lock with the collection is that you can easily replace one lock type with another, without the need for changing other code. Encapsulation, for the win.

The basic principle behind making a wrapper around any collection is declaring a wrapper class with two private fields: the lock and the collection. Then you need to declare all methods you need from the collection in your wrapper class. The implementation of those methods is rather simple, as you will just call the original collection methods, but protected with locking code. As a base wrapper class, you can use any base class you find appropriate, like a plain `TObject` class, or `TInterfacedObject` to take advantage of automatic memory management. If you opt for ARC, you will also need to declare an appropriate interface.

As an example, I will create a wrapper around `TList<T>`, but the same principles apply for other collections, too. You can make a thread-safe queue, stack, or dictionary using existing classes.

```
type
  TThreadList<T> = class(TObject)
  private
    FList: TList<T>;
    // FLock is a custom managed record
    // which is automatically initialized
    FLock: TLightweightMREW;
  public
    constructor Create;
    destructor Destroy; override;
    procedure Clear;
    function Add(const Value: T): Integer;
    procedure Delete(Index: Integer);
    function Contains(const Value: T): Boolean;
    ...
  end;

constructor TThreadList<T>.Create;
begin
  FList := TList<T>.Create;
end;

destructor TThreadList<T>.Destroy;
begin
  FList.Free;
  inherited;
end;

procedure TThreadList<T>.Clear;
begin
  FLock.BeginWrite;
  try
    FList.Clear;
  finally
    FLock.EndWrite;
  end;
end;

function TThreadList<T>.Add(const Value: T): Integer;
begin
  FLock.BeginWrite;
  try
    Result := FList.Add(Value);
  finally
    FLock.EndWrite;
  end;
end;
```

```
finally
  FLock.EndWrite;
end;
end;

procedure TThreadList<T>.Delete(Index: Integer);
begin
  FLock.BeginWrite;
  try
    FList.Delete(Index);
  finally
    FLock.EndWrite;
  end;
end;

function TThreadList<T>.Contains(const Value: T): Boolean;
begin
  FLock.BeginRead;
  try
    Result := FList.Contains(Value);
  finally
    FLock.EndRead;
  end;
end;
```

As you can see, rather simple. If you are using a multiple readers/single writer lock, you need to pay attention to the whether original collection method just reads the list or also modifies it, and call the appropriate read or write methods on the lock. With exclusive locks, the logic is a tad simpler, because you don't need to distinguish between reading and writing.

A wrapper class using **TCriticalSection** would look like the following:

```
type
  TThreadList<T> = class(TObject)
private
  FList: TList<T>;
  FLock: TCriticalSection;
public
  ...
constructor TThreadList<T>.Create;
begin
  FLock := TCriticalSection.Create;
  FList := TList<T>.Create;
end;
```

```
destructor TThreadList<T>.Destroy;
begin
  FList.Free;
  FLock.Free;
  inherited;
end;

procedure TThreadList<T>.Clear;
begin
  FLock.Enter;
  try
    FList.Clear;
  finally
    FLock.Leave;
  end;
end;
```

So far, so good.

12.2 Thread-safe iteration

By now you have probably noticed that none of the ported methods support iterating through the list.

Piece of cake... or... err... not so much...

Thread-unsafe - incorrect code

```
type
  TThreadList<T> = class
  private
    ...
    function GetCount: Integer;
    function GetItem(Index: Integer): T;
    procedure SetCount(const Value: Integer);
    procedure SetItem(Index: Integer; const Value: T);
  public
    ...
    property Count: Integer read GetCount write SetCount;
    property Items[Index: Integer]: T read GetItem write SetItem; default;
  end;
```

```
function TThreadList<T>.GetCount: Integer;
begin
  FLock.BeginRead;
  try
    Result := FList.Count;
  finally
    FLock.EndRead;
  end;
end;

procedure TThreadList<T>.SetCount(const Value: Integer);
begin
  FLock.BeginWrite;
  try
    FList.Count := Value;
  finally
    FLock.EndWrite;
  end;
end;

function TThreadList<T>.GetItem(Index: Integer): T;
begin
  FLock.BeginRead;
  try
    Result := FList.Items[Index];
  finally
    FLock.EndRead;
  end;
end;

procedure TThreadList<T>.SetItem(Index: Integer; const Value: T);
begin
  FLock.BeginWrite;
  try
    FList.Items[Index] := Value;
  finally
    FLock.EndWrite;
  end;
end;
```

Well, we reimplemented the appropriate methods used for iteration with a **for** loop, but can we get thread-safe iteration with the above methods?

```
for Idx := 0 to List.Count - 1 do
  Writeln(List.Items[Idx]);
```

Sure, getting **Count** is thread-safe, and retrieving each individual item in the list is also thread-safe. But the thread safety of individual pieces does not guarantee the thread safety of the larger code block.

From our previous examples with external locks, we can see that in order to safely iterate through the list, we need to protect the whole iteration block. If we don't protect the whole block, the number of items, or their contents or position, can change while we are iterating, causing all kinds of misbehavior and crashes.

If you are thinking that a **while** loop that checks the index range at every pass is thread-safe, you are mistaken. Sure, checking the index at every pass shortens the time frame when a collision between two threads can happen, but it **can happen** nonetheless.

How can we protect and make thread-safe the iteration, when there is no single method in the original collection we can easily wrap?

There are several ways to solve thread-safe iteration:

- add an **Enumerate/Iterate** method with a procedural parameter for code that executes in the loop (method or anonymous method)
- add a thread-safe **TEnumerator<T>** class
- temporarily expose the inner collection, while protecting access with public **Lock/Unlock**, or **BeginRead/EndRead** and **BeginWrite/EndWrite** methods

12.2.1 Iterating method with procedural parameter

Writing a thread-safe **Enumerate/Iterate** method is rather simple:

```
type
  TThreadList<T> = class
  ...
  public
  ...
    procedure Enumerate(const AProc: TProc<T>);
  end;

procedure TThreadList<T>.Enumerate(const AProc: TProc<T>);
var
  Idx: Integer;
begin
  FLock.BeginRead;
```

```
try
  for Idx := 0 to FList.Count - 1 do
    AProc(FList.List[Idx]);
  finally
    FLock.EndRead;
  end;
end;
```

And such an **Enumerate** method can be used in the following manner:

```
var
  List: TThreadList<Integer>;

  List.Enumerate(
    procedure(Item: Integer)
    begin
      // do something with the Item
    end);
```

The above **Enumerate** method provides plain read-only iteration. If the collection uses a multiple readers/single writer lock, you will need different **Enumerate** methods for reading and writing. You might also need directional enumeration—from first to last item, and vice versa. In addition you might want the ability to break out of the enumeration, in which case you will need to use a function returning a boolean, like **TPredicate<T>** instead of **TProc<T>**.

Instead of anonymous methods, you can use regular methods to provide the same functionality, which will result in faster code (as there will be no need to allocate a hidden object instance to back the anonymous method). On the other hand, code with anonymous methods is simpler to read, because the code is written in-place so you can immediately see what kind of logic will be performed inside the iteration without the need to look at some displaced method.

12.2.2 Thread-safe enumerator

The following code shows how to write a thread-safe read-only enumerator. The problem with enumerators and the **for..in** loop is that they are generally not suitable for modifying the list. So even if you would lock the list with a write lock, you still don't have the means to alter the list (you could write some really bad code, counting the items during iteration, but such code would cause other issues, and since it is a bad approach to begin with, I will not even try to pursue it).

```
type
  TThreadList<T> = class
  ...
public
  type
    TEnumerator = class(TEnumerator<T>)
  private
    FList: TThreadList<T>;
    FIndex: Integer;
    function GetCurrent: T; inline;
  protected
    function DoGetCurrent: T; override;
    function DoMoveNext: Boolean; override;
  public
    constructor Create(const AList: TThreadList<T>);
    destructor Destroy; override;
    function MoveNext: Boolean; inline;
    property Current: T read GetCurrent;
  end;

  function GetEnumerator: TEnumerator; inline;
end;

constructor TThreadList<T>.TEnumerator.Create(const AList: TThreadList<T>);
begin
  inherited Create;
  FList := AList;
  FIndex := -1;
  FList.FLock.BeginRead;
end;

destructor TThreadList<T>.TEnumerator.Destroy;
begin
  FList.FLock.EndRead;
  inherited;
end;

function TThreadList<T>.TEnumerator.GetCurrent: T;
begin
  Result := FList.FList.List[FIndex];
end;
```

```

function TThreadList<T>.TEnumerator.MoveNext: Boolean;
begin
  Inc(FIndex);
  Result := FIndex < FList.FList.Count;
end;

function TThreadList<T>.TEnumerator.DoGetCurrent: T;
begin
  Result := Current;
end;

function TThreadList<T>.TEnumerator.DoMoveNext: Boolean;
begin
  Result := MoveNext;
end;

function TThreadList<T>.GetEnumerator: TEnumerator;
begin
  Result := TEnumerator.Create(Self);
end;

```

And then we can safely iterate over the thread-safe collection with a `for..in` loop:

```

var
  List: TThreadList<Integer>;
  Value: Integer;

  for Value in List do
    Writeln(Value);

```

Delphi automatically constructs an enumerator instance before the `for..in` loop iteration starts, and that instance is automatically destroyed when enumeration is completed. The destruction of the enumerator is protected by an implicit `try...finally` block, so even if an exception happens inside the loop, the enumerator destructor will always run and unlock the collection.

12.2.3 Exposing internal collection

And the last approach is temporarily exposing the internal collection. This approach has some downsides and some upsides. A major downside is exposing the internals, as well as leaving thread safety protection to outside code. On the upside, writing more complex code with such an exposed collection is easier, as you can easily lock the collection while you perform multiple operations. Covering every possible scenario while protecting the collection from the inside could be rather tricky to achieve.

Lock/Unlock methods are suitable for exclusive locks, while multiple readers/single writer locks require you to implement both reading and writing method pairs.

```
type
  TThreadList<T> = class
  ...
  public
    function BeginRead: TList<T>;
    procedure EndRead;

    function BeginWrite: TList<T>;
    procedure EndWrite;
  end;

  function TThreadList<T>.BeginRead: TList<T>;
begin
  FLock.BeginRead;
  Result := FList;
end;

procedure TThreadList<T>.EndRead;
begin
  FLock.EndRead;
end;

function TThreadList<T>.BeginWrite: TList<T>;
begin
  FLock.BeginWrite;
  Result := FList;
end;

procedure TThreadList<T>.EndWrite;
begin
  FLock.EndWrite;
end;
```

The following example shows how you can use such temporary collection locking. What you must ensure, is that **TempList** is never ever used outside a locked block, as such usage would not be thread-safe. If you don't need to modify the collection, you can use the **BeginRead/EndRead** methods.

```
var
  List: TThreadList<Integer>;
  TempList: TList<Integer>;

  TempList := List.BeginWrite;
  try
    // you can perform any number and any kind of operations on TempList
    // while you are inside this block
  finally
    List.EndWrite;
  end;
```

Which of the presented iteration methods you will choose depends on your use case (you probably hate the word *depends* by now, but *one size fits all* situations in programming are extremely rare). There are some trade-offs in each approach and it is a choice between code speed, readability, and thread safety encapsulation. No matter which one you use, you will have thread-safe code, unless you use read-only iteration mode while you are modifying the collection, or you keep on using the temporary collection after you have released its lock.

The pattern of exposing an internal collection—which can be used for other purposes beyond iteration—is supported by the thread-safe collections in the RTL, typically via **LockList/UnlockList** method pairs. Using those methods follows the same rules as the custom implementation shown in this chapter: After you acquire the lock, you need to protect the code with a **try...finally** block, in order to prevent the collection from being locked indefinitely in the event of an exception, and you must never use the returned collection reference after you have released the lock.

Thread-unsafe - incorrect code

```
var
  List: TThreadList;
  TempList: TList;

  ...

  TempList := List.LockList;
  List.UnlockList;
  // use TempList
  TempList....
```

Thread-safe - correct code

```
var
  List: TThreadList;
  TempList: TList;

  ...

  TempList := List.LockList;
  try
    // use TempList
    TempList.....
  finally
    List.UnlockList;
  end;
```

12.3 Partial locking

The rules for locking the collection are pretty simple: You lock the collection at the beginning of some operation, and you unlock it after you are done. This works well for simple code, that either runs fast or is not called often, so that the time the collection will be in a locked state is minimal.

However, there are some scenarios where such a simplistic locking approach can create contention, and locking only parts of the collection can speed things up. When you use partial locks, it is important to keep two things in mind: first, you still need to lock the whole collection for any reading or modifications that access the underlying collection's storage, and second, you need to make sure that the pieces of the collection data you are taking out of the global collection protection will not live longer than the collection itself, if their memory is managed by the collection.

The most common situation where you may want to have partial locking is having a collection where items are other collections. If that collection itself is seldom modified, locking the whole collection together with the owned collections can slow down processing the individual collections. The ability to take each inner collection and work with it separately outside an outer collection lock can make a huge difference.

If said outer collection is not the owner of the individual items, partial locking can be achieved with relative ease. You would apply the lock on the outer collection to find the appropriate inner collection, store its reference, unlock the outer collection, and then do whatever processing (including locking that inner collection alone) you need to do.

If the outer collection owns the inner collections, the cleanest solution for independent memory management is using interfaces and reference counting. But reference counting itself has an

impact on performance, especially when used in multithreading, and it is possible that that impact can neutralize all the performance you have gained with partial locks. Also with partial locking you will need to apply two lock/unlock operations instead of one. So before you start replacing simplistic locking with more complex solutions for optimization purposes at large, make sure (measure) that those solutions will actually perform better in your particular scenario.

The following is a minimal implementation of such a thread-safe dictionary and reference-counted thread-safe list. To simplify the code, both the list interface and the dictionary only expose locking methods, and all operations on the list must be performed on the wrapped collection. In a real-life situation, those declarations can be expanded to provide commonly-used thread-safe operations, to avoid unnecessary repetitive code when using the collections. Examples of adding such thread-safe methods are provided in the previous chapters.

If needed, the dictionary (or any other outer collection) can also be implemented as a reference-counted class and have its functionality exposed through an interface. However, automatic memory management for the outer collection is not necessary for demonstration of partial locking, and is omitted for brevity.

```
type
  IList<T> = interface
    function BeginRead: TList<T>;
    procedure EndRead;
    function BeginWrite: TList<T>;
    procedure EndWrite;
  end;

  TIntfThreadList<T> = class(TInterfacedObject, IList<T>)
protected
  FLock: TLightweightMREW;
  FList: TList<T>;
public
  constructor Create;
  destructor Destroy; override;
  function BeginRead: TList<T>;
  procedure EndRead;
  function BeginWrite: TList<T>;
  procedure EndWrite;
end;

TThreadDictionary<K, V> = class
protected
  FLock: TLightweightMREW;
  FDict: TDictionary<K, V>;
public
  constructor Create;
```

```
    destructor Destroy; override;
    function BeginRead: TDDictionary<K, V>;
    procedure EndRead;
    function BeginWrite: TDDictionary<K, V>;
    procedure EndWrite;
  end;

  constructor TIntfThreadList<T>.Create;
begin
  FList := TList<T>.Create;
end;

destructor TIntfThreadList<T>.Destroy;
begin
  FList.Free;
  inherited;
end;

function TIntfThreadList<T>.BeginRead: TList<T>;
begin
  FLock.BeginRead;
  Result := FList;
end;

function TIntfThreadList<T>.BeginWrite: TList<T>;
begin
  FLock.BeginWrite;
  Result := FList;
end;

procedure TIntfThreadList<T>.EndRead;
begin
  FLock.EndRead;
end;

procedure TIntfThreadList<T>.EndWrite;
begin
  FLock.EndWrite;
end;

constructor TThreadDictionary<K, V>.Create;
begin
  FDict := TDDictionary<K, V>.Create;
end;
```

```

destructor TThreadDictionary<K, V>.Destroy;
begin
  FDict.Free;
  inherited;
end;

function TThreadDictionary<K, V>.BeginRead: TDictionay<K, V>;
begin
  FLock.BeginRead;
  Result := FDict;
end;

function TThreadDictionary<K, V>.BeginWrite: TDictionay<K, V>;
begin
  FLock.BeginWrite;
  Result := FDict;
end;

procedure TThreadDictionary<K, V>.EndRead;
begin
  FLock.EndRead;
end;

procedure TThreadDictionary<K, V>.EndWrite;
begin
  FLock.EndWrite;
end;

```

And then you can lock such a dictionary only enough to locate or add a contained list, while the list will be processed outside the dictionary lock. That will allow other threads to add or retrieve dictionary items, without waiting for processing. Of course, during processing of the particular list item, if some other thread needs to do anything with it, it will have to wait, but overall waiting will be reduced, especially if processing takes a long time.

```

var
  Dict: TThreadDictionary<string, IList<Integer>>;
  TempDict: TDictionay<string, IList<Integer>>;
  ListIntf: IList<Integer>;
  TempList: TList<Integer>;

  // run following code from multiple threads

  ListIntf := nil;
  TempDict := Dict.BeginRead;

```

```
try
  TempDict.TryGetValue(Key, ListIntf);
finally
  Dict.EndRead;
end;

if Assigned(ListIntf) then
begin
  TempList := ListIntf.BeginWrite;
  try
    // process list
  finally
    ListIntf.EndWrite;
  end;
end;
```

In the above example, while processing happens outside the lock, you may ask yourself why we need to use an interface-based inner list. Obviously the lifetime of the dictionary must be longer than any processing anyway.

A memory management problem may arise if some thread needs to remove list items from the dictionary. In such case, one thread might destroy the list, while another thread is still doing some processing on the list. This is where automatic memory management helps. We will hold a strong reference to the list we are processing, and removing the list from the dictionary will not cause premature destruction. The only *conflict* can arise if removing list from the dictionary requires cancelling any active processing for that list. But that would be a logical error in behavior, not a thread safety issue, and such workflows would require some additional mechanism that could cancel processing.

Another possible scenario that benefits from ARC, is where processing the list must be performed asynchronously in another thread. In such case, even the dictionary itself could vanish before we have finished processing the list items.

Before you start applying partial locking everywhere in your code, make sure that you actually have the appropriate use case. Otherwise, instead of speeding up processing, you may end up slowing it down.

12.4 False sense of security

Cleanup is probably the hardest thing to do right in multithreading.

If you have ever looked at the RTL thread-safe collections' code, you might have noticed the following (or similar) piece of code in the destructor:

```
destructor TThreadList.Destroy;
begin
  LockList;    // Make sure nobody else is inside the list.
  try
    FList.Free;
    inherited Destroy;
  finally
    UnlockList;
    FLock.Free;
  end;
end;
```

“Make sure nobody else is inside the list”—sounds perfectly reasonable and logical, doesn't it? You don't want to destroy the list if some other thread is still using it.

But, is it really thread-safe?

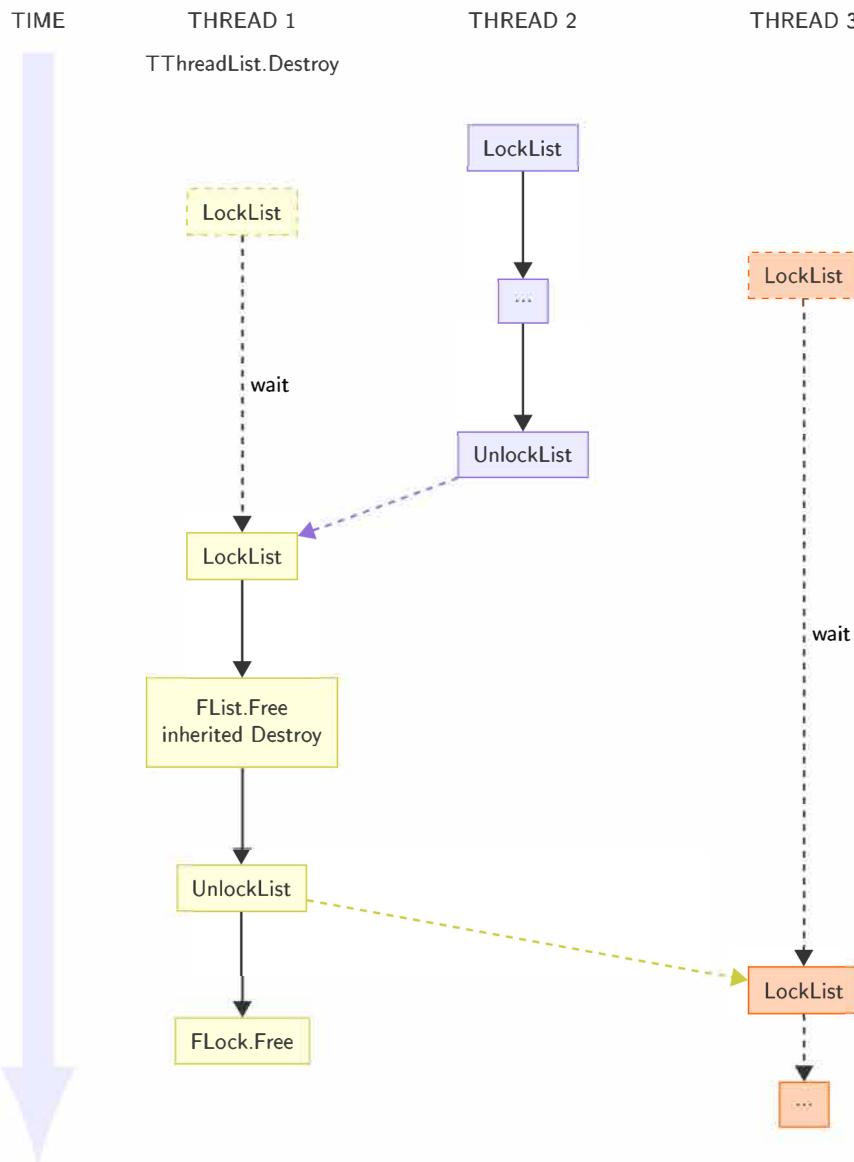
What does it actually mean that some other thread is inside the list? It means another thread has access to exposed the `FList` instance. Since access to the exposed `FList` instance needs to be protected by `LockList/UnlockList` calls on the owning `TThreadList` instance, you cannot possibly use the inner object without having a valid owner object reference.

At this point it is obvious that the core issue is not multithreading, but an ownership issue—in other words, another example of a memory management issue. As long as the `TThreadList` instance is being used, it shouldn't be destroyed. Yes, making sure that the instance is not destroyed while it is being used is a bit harder in multithreading code than in single-threaded code, but this is the original problem that needs to be solved.

It is true that while the secondary thread is inside the lock doing the `LockList -> use FList -> UnlockList` sequence, the first thread will not be able to pass the call to `LockList` inside the destructor, temporarily preventing the destruction process.

But, what if you have yet another thread using the same `TThreadList` instance? What if a third thread is waiting in line to access the list?

You can have the following sequence:

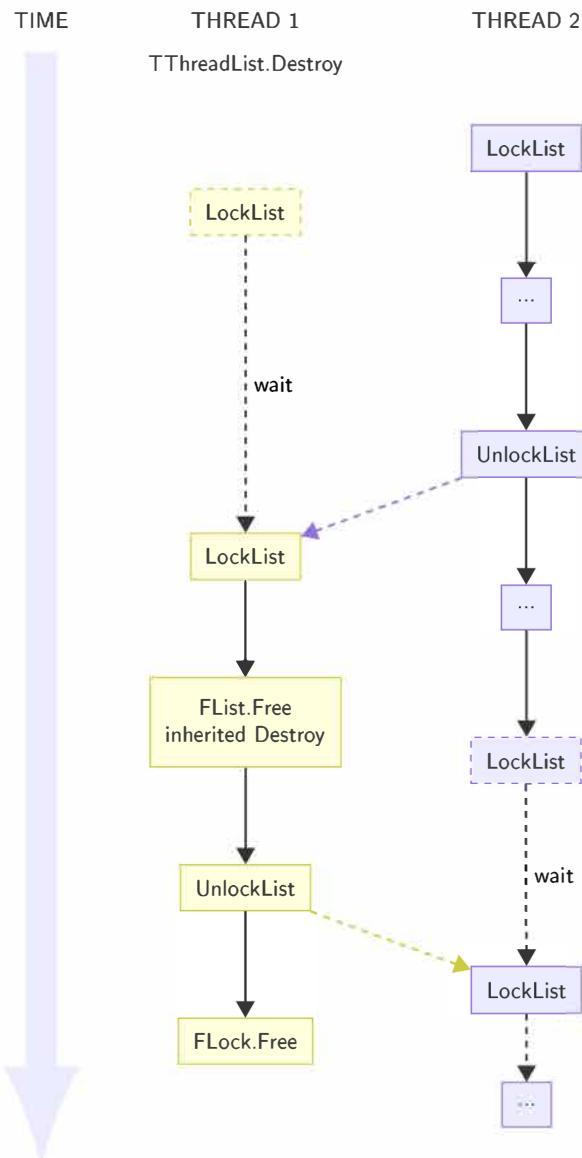


By the time the third thread manages to enter the lock with a `LockList` call, the `TThreadList` instance will be destroyed, and the whole *safe* destruction concept falls apart. This approach simply doesn't scale. Now, you may start thinking that this is good enough, because it works with two threads, and if you don't have more, you will have an easy ride.

But, does it really work with two threads?

You can easily shoot yourself in the foot with two threads, too. There is nothing that prevents the second thread from releasing the lock and trying to acquire it again.

So, the following sequence is also possible:



Just like in the previous scenario with three threads, the `TThreadList` instance will be destroyed by the time the second thread manages to enter the lock in the subsequent `LockList` call.

No amount of dancing around object destruction from within the actual destructor will really solve the problem. Simply put, you cannot protect the destruction of an object from within its destructor.

All the code in the destructor will accomplish is slowing down the destruction, and will not make destructor any more thread-safe than it would be without the locking.

You need to make sure that shared object instances are not destroyed while they can be used, and you also need to take care of thread-safe access in that broader context.

If you think that a little slowdown does not hurt, and that anything that can even remotely prevent some multithreading issues is a plus, you are mistaken.

The real danger in such *just in case* code, comes from giving you a false sense of security. It gives you license to write bad multithreaded code and makes you think you can get away with it. Once you get comfortable with writing bad code, you will soon find yourself in a situation where it will get out of hand.

It prevents you from learning proper coding patterns in simple cases, and if you don't know how to write good code in simple scenarios, you will certainly never learn to write good code in complex ones. Maintaining code that uses bad coding patterns is also hard because it is extremely fragile and even small, seemingly unrelated, changes can break it.

If the following, correct destructor of some thread-safe collection does not work for you, then the problem is not in the destructor, but in your code. You need to fix your code and achieve thread safety in the broader code context. Trying to fix it from within the destructor will not work. The same principle applies, not only for collections, but for other similar scenarios.

Thread-safe - correct code

```
destructor TThreadList.Destroy;
begin
  FList.Free;
  FLock.Free;
  inherited Destroy;
end;
```

Why that locking is introduced in the destructor in the first place is hard to tell. It has certainly been sitting there for a long time. Maybe there is a perfectly reasonable explanation behind that code, and it was meant to patch some very specific issue that, for some reason, couldn't be patched in another way at that time. And temporary patches tend to have a rather long life.

The road to hell is paved with code that fixes the symptoms instead of fixing the root cause.

12.5 Wrappers vs inheritance

In previous examples we used a wrapper class to implement a thread-safe collection. The Delphi RTL classes use same approach. An inevitable question is:

Can we implement a thread-safe collection (or any other class) through inheritance?

The answer would be: Yes, inheritance can be used, but it is not a recommended approach.

To understand why wrappers are a better option, we will compare two collections, one based on inheritance and another using a wrapper. The declarations for both the wrapper class and the inherited class are quite similar. The method declarations are identical: The only difference is an additional **FList** field in wrapper class, and of course, they use different base classes.

Wrapper class declaration:

```
TThreadList<T> = class(TObject)
private
  FList: TList<T>;
  FLock: TCriticalSection;
public
  constructor Create;
  destructor Destroy; override;
  function LockList: TList<T>;
  procedure UnlockList;
end;
```

Inherited class declaration:

```
TThreadList<T> = class(TList<T>)
private
  FLock: TCriticalSection;
public
  constructor Create;
  destructor Destroy; override;
  procedure LockList;
  procedure UnlockList;
end;
```

Wrapper class implementation:

```
constructor TThreadList<T>.Create;
begin
    inherited;
    FLock := TCriticalSection.Create;
    FList := TList<T>.Create;
end;

destructor TThreadList<T>.Destroy;
begin
    FList.Free;
    FLock.Free;
    inherited;
end;

function TThreadList<T>.LockList: TList<T>;
begin
    FLock.Enter;
    Result := FList;
end;

procedure TThreadList<T>.UnlockList;
begin
    FLock.Leave;
end;
```

Inherited class implementation:

```
constructor TThreadList<T>.Create;
begin
    inherited;
    FLock := TCriticalSection.Create;
end;

destructor TThreadList<T>.Destroy;
begin
    inherited;
    FLock.Free;
end;

procedure TThreadList<T>.LockList;
begin
    FLock.Enter;
```

```
end;

procedure TThreadList<T>.UnlockList;
begin
  FLock.Leave;
end;
```

If you compare the wrapper class implementation with the inherited class implementation, focusing solely on the locking functionality, there is not much difference besides a few more lines of code in the wrapper class' constructor and destructor, as well as setting the function result in the `LockList` method. The inherited class may seem like a simpler and better solution, not because of a few less lines of code, but more because it avoids additional heap allocation—the `FList` field.

Usage is also quite similar:

```
var
  List: TThreadList<Integer>;
  ...
  // wrapper class
procedure AddFive;
var
  TmpList: TList<Integer>;
begin
  TmpList := List.LockList;
  try
    TmpList.Add(5);
  finally
    List.UnlockList;
  end;
end;

  // inherited class
procedure AddFive;
begin
  List.LockList;
  try
    List.Add(5);
  finally
    List.UnlockList;
  end;
end;
```

If you look carefully, the above example starts to show the real differences between the two approaches.

The most obvious one is that while you cannot avoid calling `List.LockList` with the wrapper implementation, you can easily make a mistake with the inherited class implementation and call `List.Add` without locking the list.

Next, less obvious issues can rise to the surface if you try to simplify your code working with a thread list. There is plenty of ceremony around adding an item to the list, and it makes sense to add a thread-safe `Add` method that would take care of the locking instead of repeating the `LockList/UnlockList` pattern everywhere.

Implementations of a thread-safe `Add` method:

```
type
  TThreadList<T> = ...
public
  ...
  procedure Add(Item: T);
end;

// wrapper class
procedure TThreadList<T>.Add(Item: T);
begin
  LockList;
  try
    FList.Add(Item);
  finally
    UnlockList;
  end;
end;

// inherited class
procedure TThreadList<T>.Add(Item: T);
begin
  LockList;
  try
    inherited Add(Item);
  finally
    UnlockList;
  end;
end;
```

Using the new thread-safe `Add` method, our `AddFive` method can be simplified to the following code (and eventually we can even remove the `AddFive` method, because it no longer serves any purpose):

```
procedure AddFive;
begin
  List.Add(5);
end;
```

Let's compare implementations: While the `Add` method in the wrapper class does not suffer from any issues, the thread-safe `Add` method in the inherited class implementation reduces the chances that someone will forget to lock the list, on the one hand, but it can cause different problems on the other.

Namely, the `TList<T>` class already has a non-virtual `Add` method. So when you declare the new `Add` method in the descendant `TThreadList<T>` class, you cannot override the existing behavior, and which `Add` method will be called depends only on the variable type.

If the variable is declared as `TList<Integer>`, then instead of calling new thread-safe `Add` method, following code will actually call the inherited, thread-unsafe `Add` from the `TList<T>` class:

```
var List: TList<Integer>;
begin
  List := TThreadList<Integer>.Create;
  List.Add(5);
end;

procedure DoSomething(List: TList<Integer>);
begin
  ...
  List.Add(5);
end;
```

And now it is rather obvious that the wrapper implementation is the superior approach, as it avoids a whole range of multithreading bugs. It encapsulates the thread-unsafe collection, maintains its integrity and thread safety, and does not expose those internals, unless you explicitly call `LockList/UnlockList`, and when you do so, the intent of the code within will be clear, and thread safety fully maintained.

There may be rare situations where using the inherited class approach is more suitable—when avoiding additional heap allocation is crucial—but this also requires that all parts of the unsafe base class API can be fully protected in the inherited class, and that can only be achieved if all methods that manipulate data are virtual in the base class, so that the overridden thread-safe implementation will always be called. If that primary requirement is not met, then the thread safety of such a class will be extremely fragile, and its use error-prone.

However, virtual methods alone are not a guarantee that some class can be made thread-safe in such a manner that thread safety will be fully encapsulated without leaving loopholes. That also depends on the class' functionality and its public API. As we have seen from the collections example, even if the base collection classes had virtual methods and we could be certain that any

method called will go through the locking mechanism, we would still have huge loopholes open when using such a collection, because iterations are too complex to be protected from within. While we can certainly make additions that would enable us to have thread-safe iteration, we would never be able to close all the loopholes and prevent accidental mistakes.

Of course, using wrapper classes will not always give us absolute protection in more complex code, but they can significantly reduce thread safety issues, and usually you will have to bend over backwards to write thread-unsafe code.

12.6 Immutable collections

Thread-safety loves immutable data. If it cannot be modified, it is thread-safe by design. Immutable collections are another tool in achieving thread safety. Having data that does not change throughout the application lifetime is the simplest form of immutable data. If you just read some piece of data, it is always thread-safe.

But immutable collections and other kinds of immutable data types are not always used just for data that never needs to change. They can be used for achieving thread safety in situations where data needs to change, but instead of modifying the data directly, it will be copied and then the change will be made on that copy, leaving the original data intact—immutable.

The immutability of collections that store all data within the collection's internal storage is rather simple to achieve and protect. Since all data is contained within the collection, it is easy to create a copy of that data in a thread-safe manner—knowing that the collection is immutable implies that copying the data will also be thread-safe.

On the other hand, if the collection items are reference types—where the associated data is stored in another location, and the collection storage only holds a reference (pointer) to that data, the immutability of the collection storage is not enough to guarantee thread-safe copying of the whole data, only the storage part.

If the associated data is also immutable, then copying such a collection will also be thread-safe. We still need to consider the lifetime of such reference-type-based items, because immutability does not prevent another thread from destroying the data. Using reference-counted classes and making the collection hold strong references to its items also guarantees the items' lifetime and simplifies handling such collections.

When it comes to making copies, there is another thing we need to keep in mind. Thread safety depends not only on the thread safety of data—in this case, the collection—but also on the thread safety of the variable that stores the data or points to it.

If you make a copy, modify a copy, and then you store that copy back in the original's variable, you will break thread safety, unless you add protection around the variable.

This also reduces the usability of the copying approach. Depending on the data size, creating a copy can be more performance-expensive than locking, and if you need to store data back in the original place, this whole operation will defeat the original intent of using immutable data.

In some circumstances, using the **TInterlocked** API to assign the data back will be enough to provide the required thread-safety without using more expensive locking mechanisms.

Because copying a collection is usually performance expensive, immutable collections fit in fairly narrow use cases, mostly when you can guarantee that the collection always holds only a small amount of data.

On rare occasions, copying even larger collections will still be a solution if processing requires locking from many threads, where deadlocks or livelocks can occur due to code complexity.

Chapter 13

Parallel collection processing

The fastest lock is no lock at all. If a collection is populated and you just need to individually process the stored items in multiple threads, without making changes to collection storage, you can perform such processing without locking the collection from each thread working on it. You can even swap each item's contents with another's, without breaking thread safety. The main requirement for such unprotected, yet thread-safe processing, is that each thread is working on a separate batch, a non-overlapping range of items.

There are several ways of processing collections in parallel. The simplest code can be written by using `TParallel.For` from `System.Threading`, or using similar functionality from some other threading library. If you are working with older Delphi versions that don't have the Parallel Programming Library, and you don't want to use other frameworks, you can rather easily solve this with the regular `TThread` class.

The main advantage of the PPL or similar frameworks, over plain threads, is their utilization of thread pools that can more evenly spread the tasks across multiple threads, without overwhelming the system. This is especially important if you have to process items where the time needed for each item can vary significantly. So if you have 100 items, and you spread them across four threads, giving each thread a fixed 25 tasks for processing, some threads might finish much sooner than others.

If you need to combine the results of your processing, you will still need some synchronization mechanism (locking or synchronization) to protect access to the shared result data, but you should also write your code in such a way that only a minimal amount of code needs to run in protected mode. Otherwise, you are losing the advantages of parallel processing.

Parallel processing can be done on completely thread-unsafe collections, provided that they are populated by a single thread before being processed in parallel, or on thread-safe collections that are populated by a single or multiple threads before additional processing starts.

13.1 Independent processing of individual collection items

The first set of examples will show how to use parallel processing for sorting individual arrays stored in a thread-unsafe list. Because an operation on each individual item is independent from operations on other items, this kind of processing is suitable for parallel processing and doesn't require any additional protection mechanisms. The list is populated by a single thread before processing starts.

13.1.1 Single-threaded processing

This is a list sorting example using a single thread and a classic **for** loop:

```
procedure SortItemsForLoop(List: TList<TArray<Integer>>)
var
  i: Integer;
  Sorted: TArray<Integer>;
begin
  for i := 0 to List.Count - 1 do
    begin
      Sorted := List[i];
      TArray.Sort<Integer>(Sorted);
    end;
end;
```

13.1.2 Multiple threads

The first logical step to speed up sorting, would be to spawn multiple threads to sort the individual arrays. In order to correctly capture variables in the anonymous method—in this case, each list item—we need to write an additional procedure around our existing code to construct a thread with an anonymous method.

Using multiple threads will work well for lists with a fairly small number of items, where the number of spawned threads will also be small. But as soon as you need to sort larger lists, with hundreds, if not thousands of items, this approach will put so much stress on the system that will spend most of its processing power doing context switching between threads instead of doing actual work. That is, if you don't run into some other resource limit, making the whole thing crash.

Thread-safe but resource-intensive - can exhaust resources and crash

```
procedure SortItemsThreads(List: TList<TArray<Integer>>);

procedure RunThread(Sorted: TArray<Integer>);
begin
  TThread.CreateAnonymousThread(
    procedure
    begin
      TArray.Sort<Integer>(Sorted);
    end).Start;
end;

var
  i: Integer;
begin
  for i := 0 to List.Count - 1 do
    RunThread(List[i]);
end;
```

13.1.3 Multiple tasks

Tasks from the PPL or another similar threading library that use thread pools will not overwhelm the system with too many threads. The following example can handle processing lists with a much larger number of items.

Thread-safe - tasks reduce resource consumption

```
procedure SortItemsTask(List: TList<TArray<Integer>>);

procedure RunTask(Sorted: TArray<Integer>);
begin
  TTask.Run(
    procedure
    begin
      TArray.Sort<Integer>(Sorted);
    end);
end;
```

```
var
  i: Integer;
begin
  for i := 0 to List.Count - 1 do
    RunTask(List[i]);
end;
```

13.1.4 TParallel.For

Instead of running a new task for each item, you can easily process items in groups with **TParallel.For** from the Delphi PPL. Compared to tasks, where you need to write an additional procedure to properly capture task-related variables, **TParallel.For** enables you to write simpler and more readable code.

Another advantage over tasks is that **TParallel.For** is a blocking operation, which means you don't need to keep track of individual tasks if you need to run some code after processing is finished. To prevent blocking the main thread, you should use **TParallel.For** from a background thread, or within another task.

The first passed parameter to a **TParallel.For** call represents stride, the second is the lower boundary of the iteration, and the third is the upper boundary.

Stride separates processing into chunks. With a value of 10, items 0-9 are guaranteed to be processed inside the same thread. The next chunk, 10-19, will be processed in another thread, and so on. Since the PPL uses tasks to split work, some chunks can still end up being processed by the same thread, recycled from the thread pool. If each task is a resource-intensive operation, using stride to limit the number of threads used—and therefore tasks running in parallel—helps prevent resource exhaustion.

If the stride parameter is omitted, then each item will be processed individually. Regardless of the stride used, the values of **TaskIndex** will cover the whole range from **0** to **List.Count - 1**.

Thread-safe - tasks reduce resource consumption

```
procedure SortItemsParallelLoop(List: TList<TArray<Integer>>);
begin
  TParallel.For(10, 0, List.Count - 1,
    procedure(TaskIndex: Integer)
    begin
      TArray.Sort<Integer>(List[TaskIndex]);
    end);
end;
```

13.1.5 Batch processing with multiple threads

When you're using plain threads that are not managed by a thread pool, you need to limit the number of threads used for such processing, and divide the work into batches yourself.

When dividing tasks among threads, there are several considerations, and depending on the processing being done, the optimal number of threads may vary. The simplest solution for that will be to use as many threads as you have CPU cores. If you create more threads, you may lose performance on context switching, and if you don't have enough threads, you will not utilize the CPU as much as possible.

There is also a huge difference between CPU-bound tasks—where the processing bottleneck is the CPU itself—versus I/O-bound tasks, where the bottleneck is the I/O, and the CPU will be mostly idling waiting for it. With I/O-bound tasks, creating more threads than CPU cores can still increase performance, while on CPU-bound tasks, it will most certainly not help.

Because of the above, fine-tuning any multithreaded processing for a general use case is basically impossible. If you really need to fine-tune some processing, you will just have to determine (measure) what the actual bottleneck is in your code, and how to maximize processing power around that bottleneck.

Sorting arrays is a CPU-bound task, and we will take the number of CPU cores as a starting point for creating the appropriate number of threads. Limiting the number of threads based on the available CPU cores is a starting point, but thread pool-based frameworks still have an advantage over this kind of simplistic work allocation. If different batches need significantly different amounts of processing time, you will not be able to fully utilize the hardware resources you have available, and the processing will take longer than necessary.

Thread-safe - limiting the number of threads reduces resource consumption

```
procedure SortItemsBatch(List: TList<TArray<Integer>>; L, H: Integer);
begin
  TThread.CreateAnonymousThread(
    procedure
      var
        i: Integer;
        Sorted: TArray<Integer>;
      begin
        for i := L to H do
          begin
            Sorted := List[i];
            TArray.Sort<Integer>(Sorted);
          end;
        end).Start;
  end;
end;
```

```
procedure SortItemsParallelThreads(List: TList<TArray<Integer>>);
var
  i: Integer;
  Stride: Integer;
  L, H: Integer;
begin
  if List.Count <= CPUCount then
    Stride := 1
  else
    Stride := List.Count div CPUCount;

  for i := 0 to List.Count div Stride do
  begin
    L := i * Stride;
    H := (i + 1) * Stride - 1;
    if H >= List.Count then
      H := List.Count - 1;
    if L < H then
      SortItemsBatch(List, L, H);
  end;
end;
```

Note: While the simplest stride calculations just try to evenly split tasks across all available cores, such an approach might not be the best in every scenario. Optimizing algorithms on such a low level depends on many factors, not just stride, and is beyond the scope of this book, instead stepping into realm of *Parallel Algorithms*.

13.2 Dependent processing of individual collection items

Quite often, parallel processing also involves performing some dependent work. For instance, extending our previous example with sorting arrays, and calculating the sum of all items in all arrays. This kind of calculation will require some protection mechanism around the variable holding the total.

13.2.1 Single-threaded processing

Single-threaded code for doing such a calculation would look like:

```

procedure SortItemsAndSumForLoop(List: TList<TArray<Integer>>);
var
  i, j: Integer;
  Sorted: TArray<Integer>;
  Total: Int64;
begin
  Total := 0;
  for i := 0 to List.Count - 1 do
    begin
      Sorted := List[i];
      TArray.Sort<Integer>(Sorted);
      for j := 0 to High(Sorted) do
        Total := Total + Sorted[j];
    end;
end;

```

13.2.2 TParallel.For

Parallelizing that code requires us to protect **Total**, and there are several ways of doing so. The following examples use **TParallel.For** for simplicity, because the code protecting the calculation would be the same even if we used threads or tasks. The simplest way to ensure thread-safe calculation is to calculate the total in the context of the main thread.

However, this approach is highly ineffective, and depending on the number of arrays and their items, calculating the total can block the main thread for too long.

Note: **TParallel.For** is a blocking call, and calling **TThread.Synchronize** from within its anonymous method will cause a deadlock if **TParallel.For** is called from the main thread. This kind of code can only ever work if **SortItemsAndSumParallelFor** is called from a background thread.

Thread-safe - not very efficient

```

procedure SortItemsAndSumParallelFor(List: TList<TArray<Integer>>);
var Total: Int64;
begin
  Total := 0;
  TParallel.For(0, List.Count - 1,
    procedure(TaskIndex: Integer)
    var
      Sorted: TArray<Integer>;
    begin
      Sorted := List[TaskIndex];
      TArray.Sort<Integer>(Sorted);

```

```
TThread.Synchronize(nil,
  procedure
    var
      i: integer;
    begin
      for i := 0 to High(Sorted) do
        Total := Total + Sorted[i];
    end);
  end);
end;
```

While synchronizing with the main thread can be used as a way to achieve thread safety in more general scenarios, doing that only makes sense for code that absolutely needs to run in the main thread—for example, updating the UI. Protecting the calculated total using locks is a much more appropriate solution:

Thread-safe - not very efficient

```
procedure SortItemsAndSumParallelFor(List: TList<TArray<Integer>>);
var
  Total: Int64;
  Lock: TCriticalSection;
begin
  Total := 0;
  Lock := TCriticalSection.Create;
  try
    TParallel.For(0, List.Count - 1,
      procedure(TaskIndex: Integer)
        var
          Sorted: TArray<Integer>;
          i: integer;
        begin
          Sorted := List[TaskIndex];
          TArray.Sort<Integer>(Sorted);
          Lock.Enter;
          try
            for i := 0 to High(Sorted) do
              Total := Total + Sorted[i];
          finally
            Lock.Leave;
          end;
        end);
  finally
```

```
  Lock.Free;
end;
end;
```

How you use the lock is also extremely important. If you lock too much code, you will add an unnecessary bottleneck. *Too much code* does not merely mean lines of code, but how much time it takes to run the code within the lock. We have locked only two lines of code, but those two lines are performing calculations in a loop that can take some time to finish.

We need to protect **Total**, and from that perspective, the lock was correctly applied, but we can modify our calculation algorithm to minimize the time the code spends within the lock. By introducing an intermediate variable that will hold a temporary result, we can avoid locking the whole calculation and lock only the minimum amount of code needed to achieve thread safety:

Thread-safe - more efficient algorithm

```
procedure(TaskIndex: Integer)
var
  Sorted: TArray<Integer>;
  TempTotal: Int64;
  i: integer;
begin
  Sorted := List[TaskIndex];
  TArray.Sort<Integer>(Sorted);
  TempTotal := 0;
  for i := 0 to High(Sorted) do
    TempTotal := TempTotal + Sorted[i];
  Lock.Enter;
  try
    Total := Total + TempTotal;
  finally
    Lock.Leave;
  end;
end);
```

Modifying the algorithm to minimize the code that needs to be protected can be done regardless of the protection mechanism. Even if you need to synchronize with the main thread for some reason, you can use the same optimization:

Thread-safe - more efficient algorithm

```
procedure SortItemsAndSumParallelFor(List: TList<TArray<Integer>>);
var
  Total: Int64;
begin
  Total := 0;
  TParallel.For(0, List.Count - 1,
    procedure(TaskIndex: Integer)
    var
      Sorted: TArray<Integer>;
      TempTotal: Int64;
      i: integer;
    begin
      Sorted := List[TaskIndex];
      TArray.Sort<Integer>(Sorted);
      TempTotal := 0;
      for i := 0 to High(Sorted) do
        TempTotal := TempTotal + Sorted[i];
      TThread.Synchronize(nil,
        procedure
        begin
          Total := Total + TempTotal;
        end);
    end);
end;
```

In this example, we can also completely avoid the lock. The operation we are performing here, adding a number, can also be done with the help of the **TInterlocked** API. **TInterlocked.Add** will add a number in a thread-safe manner. Using the **TInterlocked** API and lock-free operations is not always possible, but whenever you can use them, they are the most performant solution.

Thread-safe - the most efficient algorithm

```
procedure SortItemsAndSumParallelForLockFree(List: TList<TArray<Integer>>);
var
  Total: Int64;
begin
  Total := 0;
  TParallel.For(0, List.Count - 1,
    procedure(TaskIndex: Integer)
    var
```

```
Sorted: TArray<Integer>;
TempTotal: Int64;
i: integer;
begin
  Sorted := List[TaskIndex];
  TArray.Sort<Integer>(Sorted);
  TempTotal := 0;
  for i := 0 to High(Sorted) do
    TempTotal := TempTotal + Sorted[i];
  TInterlocked.Add(Total, TempTotal);
end);
end;
```

Before you start processing collections or any other task in parallel, you should make sure that your problem is suitable for parallelization, meaning that it is easy to partition the algorithm into separate subtasks that can run independently.

Having some dependent operations, which can be executed in any order and where there is still plenty of time spent on independent processing—as shown in the previous example—is fine, but if each individual task depends on the results of the previous operation, then such processing might not benefit from being done in parallel, and using a single background thread can be a better option.

You should pay special attention to resource-intensive tasks (for instance, tasks which require a lot of temporary memory allocations), because running too many of them in parallel can exhaust those resources, causing crashes and other issues.

Chapter 14

Components

The thread safety classification of **TComponent** is extremely complicated. By design, **TComponent** is the base class for Delphi visual frameworks and all components and controls that can be handled at design time by the IDE and Form Designer. Because of such tight coupling with the user interface, **TComponent** and its descendants are not thread-safe, and can only be safely used in the context of the main thread.

Thread-unsafe - main thread only

- **System.Classes.TComponent**
- **VCL.Controls.TControl**
- **FMX.Types.TFmxObject**

However, real life is much more complicated, and some non-visual components can be used in background threads if some conditions are satisfied. The thread safety and usage rules for each descendant of **TComponent** can vary greatly, and they will be covered separately for each particular class covered in this book.

While some descendants of **TComponent** can be used in background threads under specific circumstances, it is important to remember that all visual controls should only be used in the context of the main thread.

Thread-unsafe - specific conditions apply

- **System.Classes.TComponent**
- **System.Classes.TDataModule**

There are several interconnected features in the **TComponent** class that make it thread-unsafe.

The first is its ownership model, and the **Owner** property. Each owner holds a thread-unsafe list of owned components. When each of the owned components is destroyed, that list is updated in a thread-unsafe manner. Every time a new component is constructed with that owner, the list of components is updated in thread-unsafe manner. When the owner component itself is destroyed, it will destroy all owned components.

The next thread-unsafe feature is its notification model, built around the **Notification** method (this is a separate feature from the OS Notification API, **System.Notification**), and the ability to register/unregister other components that will receive notifications when some component is added to or removed from that particular component's list of owned components, as well as when the component itself is destroyed. Again, having a thread-unsafe list backing the notification system is what makes the notification system unsafe.

And then, each **TComponent** has a thread-unsafe list of **Observers**, that sends notifications to registered observers about particular events.

Basically, whatever you touch in the base **TComponent** class is not thread-safe by design. If it is not thread-safe by design, how can any component be used in background threads?

Once the source of thread unsafety is removed (read: not used), some non-visual components can be safely used in the background threads. However, that safety also depends on the rest of the code (used) in each particular descendant of **TComponent**, and almost none of the component-based classes are thread-safe when simultaneously accessed by multiple threads. In other words, while they are being used, their inner state will change in a thread-unsafe manner. This is extremely important to keep in mind.

Note: In theory, a descendant of **TComponent** can be written in such a way that it allows full thread safety (minus the unsafe parts implemented in the base class) for the rest of their functionality. But, building a fully thread-safe class that can be shared across multiple threads on top of a **TComponent** base is a bit like building a high tower on soft ground. You may end up building the Tower of Pisa, but achieving such an architectural *miracle* that can collapse at any moment is probably not something software developers should aspire to.

The RAD features in Delphi include not only visual controls and non-visual components directly related to GUI design, but also some other features like database access, Internet clients and servers, and other similar components that can be dropped on a form or data module and configured through the visual designer. Because of that, Delphi has an abundance of components that can be used in background threads if some basic thread-safety rules are followed.

At the same time, those components have been commonly used in single-threaded applications, and once built, transforming such code to support multiple threads can be a daunting task. Writing new code using those non-visual RAD features in combination with visual containers is not recommended. Data modules give more separation from the thread-unsafe GUI, and they can be used to organize more complex configurations that can be used in background threads, but for simpler tasks, constructing and configuring components through code is preferred.

14.1 Using components in background threads

The primary source of unsafety in the **TComponent** class is the **Owner** property. If the component does not have an owner, then this source of unsafety is eliminated. That still does not mean that every non-visual component instance without an owner can be used in a background thread.

You may think that as long as the owner instance is alive while the thread is running, then constructing a component in a background thread with an owner belonging to the main thread is thread-safe.

But, that is not true, and the following example is not thread-safe.

The main problem here is not the lifetime of the owner, but the fact that the component collection that maintains the list of owned child components is not thread-safe. Adding a new component to that collection—or removing one—is not a thread-safe operation, and it can corrupt the collection data.

Thread-unsafe

```
procedure TMyForm.RunFoo;
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      Component: TFooComponent;
    begin
      Component := TFooComponent.Create(Self);
      try
        // use Component
      finally
        Component.Free;
      end;
    end).Start;
end;
```

Of course, owner lifetime could also be a source of problems. If the owner were to be destroyed while the background thread was still running, the owner would destroy the component while it was still being used in the background thread.

Not having an **Owner** is just one requirement. Another one is not using the notification system involving other components that don't belong to the same thread.

If we assume that **TFooComponent**'s functionality does not have any other thread safety issues that would prevent it from running in the context of the background thread, then the following coding pattern is thread-safe.

Thread-safe

```
procedure RunFoo;
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      Component: TFooComponent;
    begin
      Component := TFooComponent.Create(nil);
      try
        // use Component
      finally
        Component.Free;
      end;
    end).Start;
end;
```

However, if you have several dependent non-visual components that belong to and will be used in the same thread, it is safe to set one of them as the owner of the others. In other words, using an owner is unsafe only when the owner component belongs to another thread. This is a common approach when dealing with multiple components required to perform some background operation, as it simplifies memory management—freeing the parent component will take care of the other components constructed in the process.

Thread-safe

```
TThread.CreateAnonymousThread(
  procedure
  var
    Parent: TFooComponent;
    Child: TBarComponent;
  begin
    Parent := TFooComponent.Create(nil);
    try
      Child := TBarComponent.Create(Parent);
      ...
    finally
      Component.Free;
    end;
  end).Start;
```

We can also construct a component in the context of one thread, initialize it, and then hand it over to another thread, as long as we don't do anything else with the component in the first thread while the background thread is using it. It is also important to avoid any initialization code that might result in using other shared data, that exists outside that component, from within the background thread. If assigning some property creates a copy of the data, then it is safe to use in a background thread. If the data is just stored as a reference, then it is not, unless it will be only read by both origin and background thread—read-only (immutable) data is thread-safe.

As we can see, handing over the component complicates memory management a bit, but that would be the case with any other non-managed object instances.

Thread-safe

```
procedure RunFoo;
var
  Component: TFooComponent;
begin
  Component := TFooComponent.Create(nil);
  try
    // initialize Component data
    Component.XXX := ...
    ...
    TThread.CreateAnonymousThread(
      procedure
      begin
        try
          // use Component
        finally
          Component.Free;
        end;
      end).Start;
  except
    // free Component if initialization fails
    Component.Free;
    raise;
  end;
end;
```

We can use a handover in situations where we have dependent components. For instance, we don't have to construct dependent components in the same thread. We can construct the parent component in one thread, hand it over to another, and then construct the rest of the components.

Just like **Owner** is safe to use when the owner belongs to the same thread as its owned components,

it is safe to use other thread-unsafe parts of the component, like the notification model and observers, as long as they are used between components **belonging to the same thread**.

Belonging to the same thread is an extremely important requirement and if you don't follow it strictly you can very easily trip over the notification system. It is important to remember that assigning one component as a property of another one is not a one way change and it modifies content of both components. So even if it may seem that the component you are assigning to another is in read-only mode, it is not, and releasing any of those components will also trigger content modifying notifications in the other component.

Once we hand a component over to another thread, we shouldn't do anything with that component or any other related component in the original thread as long as the other thread is using the component. If you hand over one component to another thread, you are basically handing over all related components, too.

If you are kind of confused right now, and you think that putting **TComponent** and its descendants in the *use only in the main thread* category was over the top, you might be right to a point.

There are several reasons why I put a red flag on **TComponent**. First, visual controls are not thread-safe, and can only be used in the context of the main thread. Yes, I know **TComponent** is only an ancestor to visual controls, and it does not make much sense to judge the ancestor's thread safety based on its descendants. But, **TComponent** and its descendants are meant to be used in the visual designer, which creates a tight coupling between components and their completely thread-unsafe GUI surroundings. This is what makes any **TComponent** a liability when it comes to thread safety.

Simply saying that non-visual components are generally single-thread-safe would be a half-truth, and it would open up a fairly easy trap for many developers to fall into. It is far better to create an atmosphere where everyone would use components with caution, suspiciously looking at every line of code, than to allow them to drop a component on a form, and then wonder why their application falls apart when they use such a non-visual component in the background threads.

On top of that, not every non-visual component *can* be safely used in background threads in the first place.

14.2 Component streaming

Dropping a component on a form creates a tight coupling between the thread-unsafe GUI and the potentially single-thread-safe component. All components and controls on forms, frames or data modules (component containers) will be dynamically created and loaded through the component streaming system during construction of the container component. This is where things get more complicated.

Namely, the component streaming system has some thread safety built in. That doesn't mean you can safely load any container in a background thread, but it opens up some possibilities and certain scenarios when loading non-visual containers holding particular non-visual components

(again, not all such components are thread-safe). Forms and frames are visual containers, so they are inherently unsafe, but **TDataModule** and its descendants can be used in a thread-safe manner.

The main player in this process is the globally defined **GlobalNameSpace** lock.

```
var  
  GlobalNameSpace: IReadWriteSync;
```

It locks access to the global namespace list, and during container construction, streaming, and destruction, it prevents multiple threads from messing up the process and corrupting shared data. So if the streamed container and its components are not otherwise thread-unsafe, constructing and destructing such a container can safely be run in the context of the background thread.

This allows construction and loading of complex data modules in the background, leaving the UI responsive. However, only one thread can stream anything at a time, so streaming cannot be parallelized, and it only makes sense to do it in the background for things that need a significant time to load (more than a few seconds each), or where setup includes accessing and initializing remote resources (databases, web services, network, remote storage) that cannot be done within a predictable and consistent timeframe.

Again, constructing a data module without an owner is imperative for achieving thread safety. There are basically two construction scenarios for a data module in a background thread.

The first one is where the data module and all of its components will be used exclusively within the background thread. In such cases, the data module variable will be a thread-local variable, and the module will be constructed, used and destroyed within the thread's **Execute** method (or in case of anonymous thread or task, inside the method passed as parameter) in the following manner:

Thread-safe

```
procedure TMyThread.Execute;  
var  
  Module: TMyDataModule;  
begin  
  Module := TMyDataModule.Create(nil); // Owner must be nil  
  try  
    ...  
  finally  
    Module.Free;  
  end;  
end;
```

```
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      Module: TMyDataModule;
    begin
      Module := TMyDataModule.Create(nil); // Owner must be nil
      try
        ...
      finally
        Module.Free;
      end;
    end).Start;
end;
```

The second scenario is where the data module is only constructed in the background thread (because construction takes long time), but will be used in the context of the main thread. In such cases, it is important to guarantee that the constructing thread will run only once, and that the global **Module** variable is not accessed until it is fully constructed. Depending on the components created in the data module, it might also be important to ensure that the application will not be terminated before we have finished such an initialization process. This is less an issue of freeing up memory that will be cleared anyway when the application closes, but more of having a clean and error-free shutdown process.

During a construction process that runs in a background thread, the module owner must be **nil**, but after construction is finished, we can transfer ownership of that module to **Application**.

Thread-safe

```
var
  Module: TMyDataModule;
begin
  TThread.CreateAnonymousThread(
    procedure
    begin
      Module := TMyDataModule.Create(nil); // Owner must be nil
      TThread.Queue(nil,
        procedure
        begin
          // transfer ownership of the module to the Application instance
          Application.InsertComponent(Module);
        end);
    end).Start;
end;
```

Single-thread-safe containers

Can be constructed, destructed or used in a single background thread, if all owned components on the particular data module also can be constructed, destructed or used in a background thread.

For instance, some component classes commonly used in data modules that don't support loading in background threads, are `Vcl.ImageCollection.TImageCollection` and `Vcl.ImgList.TCustomImageList`. Unfortunately, the streaming process for those image collections includes creating graphic objects in a thread-unsafe manner, that only works properly if running in the context of the main thread. If those or any similarly unsafe components are part of the data module, then such a module can only be constructed and loaded from the main thread.

- `System.Classes.TDataModule`

Thread-unsafe containers - can be constructed, destructed and used in main thread only

While the shared global namespace is also protected during the construction and destruction processes of forms and frames, constructing and destroying forms and frames is never thread-safe, because they also involve unprotected access to other shared global data.

- `VCL.Forms.TCustomFrame`
- `VCL.Forms.TCustomForm`
- `FMX.Forms.TFrame`
- `FMX.Forms.TCommonCustomForm`

Chapter 15

RTTI

Delphi's enhanced RTTI is meant to be thread-safe. After all, it is just runtime support for metadata included in the binary during compilation. However, just like with any other code, the original design and intent do not mean there are no bugs.

One of the sources of thread unsafety in RTTI comes from handling the lifecycle of the `TRttiPool` singleton, which needs to be valid to perform any RTTI-related operations. Not only is this instance lazily initialized, but it can also be destroyed (and recreated) during the application lifetime.

The lifecycle of the `TRttiPool` singleton is handled through `TPoolToken` instances, which are not restricted to a single instance. Every time you call `TRttiContext.Create` or directly call any of the functions in `TRttiContext`, the `EnsurePoolToken` function will be called to make sure that the `FContextToken` field, holding a reference to a `TPoolToken` instance, is initialized.

As long as you have a valid context token instance, you will also have a valid `TRttiPool` singleton.

Context tokens are not just handled through the localized field in the `TRttiContext` record, but also through the global `FGlobalContextToken` reference, and its associated global token counter.

The lifecycle of a global context token is managed through the `TRttiContext.KeepContext` and `TRttiContext.DropContext` functions, which increase and decrease the global context counter, and initialize and release the `FGlobalContextToken` reference.

The problem lies in the `EnsurePoolToken` function, which is not thread-safe, and any code in background threads that touches enhanced RTTI can cause concurrency issues and crash the application. Specifically, the problem is caused by thread-unsafe assignment of the global token interface reference to a local variable. This assignment attempts to avoid the construction of another `TPoolToken` instance if the global token reference is alive at that moment.

Thread-unsafe - incorrect code

```
var
  tok: IInterface;
begin
  {$IFDEF USE_MONITOR_FOR_GLOBALCONTEXT}
  tok := TRttiContext.FGlobalContextToken;
  {$ELSE}
  tok := _GlobalContext.FGlobalContextToken;
  {$ENDIF}
  ...

```

Any code working with `TRttiContext` where the local `FContextToken` reference has not yet been initialized can interfere with code calling the `KeepContext/DropContext` sequence. Such code is spread all over the Delphi frameworks, and a clash is inevitable. That does not mean the issue will happen frequently, but it can happen nonetheless.

The simplest workaround for the issue is to initialize and store the global context token in the initialization section of some unit, which will execute before any background threads start running. If the global context token cannot be released while threads are running, then multiple threads will not be able to interfere, and the otherwise thread-unsafe interface assignment will run safely.

Constructing a shared RTTI context is a fairly common practice, but calling `KeepContext` is crucial for preventing concurrency issues, and most such initialization code misses that part. The order of statements in the initialization section is not important, as both `TRttiContext.Create` and `TRttiContext.KeepContext` can initialize the `TRttiPool` singleton. The main difference is whether the `ctx` record will hold a reference to the global context token, or hold an additional instance of `TPoolToken`.

```
var
  ctx: TRttiContext;

initialization
  TRttiContext.KeepContext;
  ctx := TRttiContext.Create;

finalization
  TRttiContext.DropContext;

end.
```

Another prominent issue in enhanced RTTI is a bug in `TRealPackage.FindType`, and other

functions that rely on the `FNameToType` lookup table initialized in the `MakeTypeLookupTable` procedure. The issue has been fixed in Delphi 10.3 Rio, and if you use newer versions, you can safely ignore the workaround, but the cause of the issue is still a nice example of how it only takes one small omission to break thread safety.

`MakeTypeLookupTable` uses double-checked locking—a performance optimization used when there are two possible code paths, and one does not require a lock if some condition is met. If the condition is not met, then it must be re-checked again after acquiring the lock to prevent concurrency issues.

The following example is simplified code from `TRealPackage.MakeTypeLookupTable`, used to initialize the `FNameToType` and `FTypeToName` fields that are used later on in other `TRealPackage` methods, including `FindType` to locate the requested data.

Thread-unsafe - incorrect code

```
procedure TRealPackage.MakeTypeLookupTable;

procedure DoMake;
...
begin
  TMonitor.Enter(Flock);
  try
    if FNameToType <> nil then
      Exit;

    FNameToType := TDictionary<string,PTypeInfo>.Create;
    FTypeToName := TDictionary<PTypeInfo,string>.Create;

    // populates FNameToType and FTypeToName
    ...
  finally
    TMonitor.Exit(Flock);
  end;
end;

begin
  if FNameToType <> nil then
    Exit;
  DoMake;
end;
```

If `FNameToType` is assigned, that means the lookup tables are already initialized and ready to use. Both `nil` checks—the one before the lock acquisition, and the one after—are correctly coded. The bug lies in the code that follows after the second `nil` check.

Namely, a freshly constructed dictionary is directly assigned to the `FNameToType` field, that is used as a flag saying that that lookup table is fully initialized and ready for use. But at that point, the dictionary is not yet populated with any data. Also, the second dictionary, `FTypeToName`, might not be even constructed yet. So it is perfectly possible that while one thread is in the process of initializing the dictionaries, another thread has already started using them because it successfully passed the `nil` check.

The solution for this problem is rather simple. It requires that the dictionary referenced by the `FNameToType` field is not assigned to the field before the initialization code (including the one populating the second lookup table) is fully completed. Instead of referencing the dictionary directly through a field, it will be referenced through a local reference, and the assignment to the field reference—which is an atomic operation, so thread-safe in this context—should be the last code that runs before releasing the lock.

Thread-safe - correct code

```
procedure TRealPackage.MakeTypeLookupTable;

procedure DoMake;
var
  LTable: TDictionay<string, PTypeInfo>;
begin
  TMonitor.Enter(Flock);
  try
    if FNameToType <> nil then
      Exit;

    LTable := TDictionay<string, PTypeInfo>.Create;
    FTypeToName := TDictionay<PTypeInfo, string>.Create;

    // populates FNameToType and FTypeToName
    ...
    FNameToType := LTable;
  finally
    TMonitor.Exit(Flock);
  end;
end;

begin
  if FNameToType <> nil then
    Exit;
  DoMake;
end;
```

If you are working with Delphi versions suffering from this bug, you can apply the above fix directly in the `System.Rtti` unit, or if that is not an option, you can use another workaround. Thread safety here is broken if multiple threads try to initialize the lookup table at the same time. If you trigger lookup table initialization before multiple threads start running, there will be no problems.

The simplest way to do this is by calling `FindType('')` on the RTTI context at some point before you start running threads.

As always, possible threading issues always lurk around. You need to be prepared for such possibilities and the fact that not all issues have yet been found and reported. At this time, there is another, still unresolved issue on ARM platforms regarding `TMethodImplementation`—and thus affecting `TVirtualInterface`—reported as RSP-17897.

Part 3. Core Frameworks

Chapter 16

Serialization

Serialization is the process of converting an object or other kind of data structure into a format that can be stored or transmitted, preserving its state. The opposite process, converting stored data back to an object or data structure, is called deserialization. (Though when the direction of conversion doesn't matter, the term *serialization* is used to refer to either of them.)

With serialization, we are entering the realm of more complex code, where assessing thread safety is not so simple, and where the contributing factors to unsafety start to add up. Any bugs or thread unsafety in other code used as building blocks also has a negative impact on the thread safety of the resulting code. Because of the complexity of the process, there is plenty of room for other bugs to hide, and even when some framework, library or even class is meant to be thread-safe, undetected and unfixed bugs can cause issues.

That all code can have bugs is important to keep in mind, not only when using the RTL and other Delphi-provided frameworks, but also any other 3rd-party library or your own code. Just because some code is supposed to be thread-safe, that doesn't mean it actually is.

Most of the non-UI related frameworks are single thread-safe. They are intended to be used in background threads—sometimes with additional constraints around acceptable usage—but they don't have thread safety built in, because thread safety always comes at a cost. This is not much different from the base classes where most of them are single thread-safe, too.

Except for trivial data, serialization is usually a time-consuming process, but at the same time, it is also very suitable for running in background threads. When dealing with the thread safety of the serialization process, there are two separate concerns. The first is the thread safety of the data that needs to be converted (regardless of the direction of conversion) and the resulting data, and the second is the thread safety of the serializers—instances that perform or otherwise participate in the process.

If the input data is not accessible from multiple threads during serialization, then it is thread-safe. Shared input data, whether it is an object or a storage medium, needs to be either read-only—immutable—or needs to be protected by a synchronization mechanism to be thread-safe.

Input data:

- not shared between threads
- read-only—immutable
- protected by a lock

Output data is usually a new instance returned as a result, and as such, it is thread-safe. However, it is also possible to reuse an existing instance and fill it with new data. In such cases, if the instance is shared between threads, it needs to be protected by a synchronization mechanism.

Output data:

- not shared between threads
- protected by a lock

The thread safety of the serializer depends on the serialization framework used, and on whether the process merely uses a simple function (class method) that does not rely on outside state, or requires a serializer instance. If an instance is required, thread safety depends on whether the state of the serializer instance changes during the serialization process.

Serializer:

- function, or class method
- stateless or immutable serializer instance
- stateful serializer instance and supporting instances
 - serializer with configuration
 - standalone configuration data
 - data holding intermediate processing state

Serialization methods are also often incorporated into the data in the form of `LoadFromXXX` and `SaveToXXX` or similar methods. Eventually, such methods will also call a serializer in the form of a function, or a stateless or stateful instance. The thread safety requirements in the context of the serializer don't change. But, if the serialization process is triggered from within the data instance, it may contain a hidden state change that is susceptible to race conditions.

While such hidden state changes are commonly triggered by data instance methods, they also might be triggered by outside serialization.

If **any state** of the data instance changes—even temporarily—such data instances don't satisfy the read-only requirement and must be protected by a **lock** during the serialization process if multiple threads have simultaneous access to them—other synchronization mechanisms can be used, but locks are the most suitable and therefore the most commonly used. Usually, such changes involve setting some state flags that mark the instance as being in reading or writing mode or similar.

16.1 Protecting the data

Because there are two aspects to protecting the serialization process, I'm going to treat them as separate problems, and start with examples covering data thread safety and assume that our serializer—the **Convert** function, taking input data as a parameter and returning the converted data—is thread-safe.

Protecting the data for serialization follows already well-established patterns, and is not any different from protecting the data in other situations. The following examples also provide usable templates for protecting data beyond serialization scenarios.

The data type used for serialization in the following examples:

```
type
{$M+}
TFoo = class
private
  FData: string;
published
  property Data: string read FData write FData;
end;
```

We can also serialize reference-counted classes with automatic memory management:

```
type
IFoo = interface
  function GetData: string;
  procedure SetData(const Value: string);
  property Data: string read GetData write SetData;
end;

{$M+}
TInterfacedFoo = class(TInterfacedObject, IFoo)
private
  FData: string;
published
  property Data: string read GetData write SetData;
end;

function TInterfacedFoo.GetData: string;
begin
  Result := FData;
end;
```

```
procedure TInterfacedFoo.SetData(const Value: string);
begin
  FData := Value;
end;
```

16.1.1 Unshared data

Unshared data is the simplest scenario. If the data *cannot* be simultaneously accessed by multiple threads, there is no need to protect it. In such scenarios, we need to pay attention to the *sharing* aspect and make sure that the data is really not shared between multiple threads.

The one and only requirement an unshared data type must satisfy is that it can be used in the context of a background thread. In other words, if the data type can only be used in the main thread, then even if unshared, the serialization of such a data type will not be thread-safe.

There are some exceptions to that rule—for instance, serializing data modules or non-visual components—but in such cases, we need to know that the serialization process itself will not trigger any code that needs to run in the main thread.

Data initialized in one thread and then passed to another for further handling—in this case, for serialization—is a typical example of data handover.

Thread-safe serialization with data handover

```
procedure Serialize;
var
  Foo: TFoo;
begin
  Foo := TFoo.Create;
  try
    Foo.Data := 'abc';
    TThread.CreateAnonymousThread(
      procedure
        var
          Converted: string;
        begin
          try
            Converted := Convert(Foo);
          finally
            Foo.Free;
          end;
          // do something with Converted data
        ...
      end).Start;
  end;
```

```

except
  Foo.Free;
  raise;
end;
end;

```

As usual, memory management complicates the code a bit. Using a reference-counted class would make the code cleaner:

```

procedure Serialize;
var
  Foo: IFoo;
begin
  Foo := TInterfacedFoo.Create;
  Foo.Data := 'abc';
  TThread.CreateAnonymousThread(
    procedure
    var
      Converted: string;
    begin
      Converted := Convert(Foo);
      // do something with Converted data
      ...
    end).Start;
end;

```

One of the more common mistakes with handovers is considering reference types passed as parameters to be thread-safe. If such a passed reference is still being used and modified after it is passed as a parameter, such code is not thread-safe.

The following `Serialize` procedure takes data as a parameter and uses it in a background thread. That procedure, on its own, is thread-safe. It is what we do with the data passed as parameter in the outer context that makes the difference.

```

procedure Serialize(const Foo: IFoo);
begin
  TThread.CreateAnonymousThread(
    procedure
    var Converted: string;
    begin
      Converted := Convert(Foo);
      // do something with Converted data
      ...
    end).Start;
end;

```

The following usage of the `Foo` instance in combination with the above `Serialize` procedure is not thread-safe. Two threads will have simultaneous access to the object, and since one of them also modifies its content, the instance is not being shared in read-only mode.

Thread-unsafe - incorrect code

```
procedure Work;
var
  Foo: IFoo;
begin
  Foo := TInterfacedFoo.Create;
  Foo.Data := 'abc';
  Serialize(Foo);
  Foo.Data := '123';
end;
```

The following example is just a variant of the instance handover pattern. We are just not handing it off directly to be captured by an anonymous method, but we are handing it over as a parameter. The only thing that matters in the handover pattern, regardless of how it is done, is that once the instance is handed over, it should not be used (modified) afterwards in the code that originally owned it.

Thread-safe correct code

```
procedure Work;
var
  Foo: IFoo;
begin
  Foo := TInterfacedFoo.Create;
  Foo.Data := 'abc';
  Serialize(Foo);
  // Foo is not modified after this point
  ...
end;
```

16.1.2 Read-only data

Read-only data can be shared among multiple threads. Because we are allowing multiple threads access to the data, we must ensure that the memory management of such data is correctly handled. If the data type is automatically managed, whether it is a value type or a reference-counted type, then we don't have to do anything in that regard. If the data type requires

manual memory management, we must ensure that the lifetime of such a data instance will be longer than its usage in threads. In other words: We can free such an instance only after all threads have finished working with the instance.

In the following example, a read-only instance is being handed over to two different threads, and still used from the original thread. Read-only data does not necessarily imply dealing with true immutable data, but rather, data that will only be read and never modified by the processing code. As long as all threads just read the data, such code is thread-safe. It is important to remember that thread safety implies that both the data and its reference must not be modified while multiple threads have access to them.

Thread-safe serialization with read-only data

```
var
  Foo: IFoo;
begin
  Foo := TInterfacedFoo.Create;
  Foo.Data := 'abc';

  TThread.CreateAnonymousThread(
    procedure
    var
      JSON: string;
    begin
      JSON := ConvertJSON(Foo);
    end).Start;

  TThread.CreateAnonymousThread(
    procedure
    var
      XML: string;
    begin
      XML := ConvertXML(Foo);
    end).Start;

  Writeln(Foo.Data);
end;
```

16.1.3 Mutable data protected by a synchronization mechanism

If we need to serialize data that is accessed by multiple threads and can also be modified by any of them, we need to protect such data with a synchronization mechanism. We need to be able to lock the complete data instance to preserve the consistency of the data.

For instance, having a thread-safe data type with several properties, where modifying each individual property is protected with an internal locking mechanism, will not be enough to ensure that the current contents of the instance will be serialized when serialization begins.

The following class is thread-safe in the sense that it allows you to access its instances from multiple threads and safely modify their contents.

Thread-safe data type

```
type
  IValues = interface
    function GetValue1: string;
    function GetValue2: string;
    procedure SetValue1(const Value: string);
    procedure SetValue2(const Value: string);
    property Value1: string read GetValue1 write SetValue1;
    property Value2: string read GetValue2 write SetValue2;
  end;

  TThreadValues = class(TInterfacedObject, IValues)
  private
    FLock: TCriticalSection;
    FValue1: string;
    FValue2: string;
    function GetValue1: string;
    function GetValue2: string;
    procedure SetValue1(const Value: string);
    procedure SetValue2(const Value: string);
  public
    constructor Create;
    destructor Destroy; override;
  published
    property Value1: string read GetValue1 write SetValue1;
    property Value2: string read GetValue2 write SetValue2;
  end;

  constructor TThreadValues.Create;
begin
  FLock := TCriticalSection.Create;
end;

destructor TThreadValues.Destroy;
begin
  FLock.Free;
end;
```

```
    inherited;
end;

function TThreadValues.GetValue1: string;
begin
  FLock.Enter;
  try
    Result := FValue1;
  finally
    FLock.Leave;
  end;
end;

function TThreadValues.GetValue2: string;
begin
  FLock.Enter;
  try
    Result := FValue2;
  finally
    FLock.Leave;
  end;
end;

procedure TThreadValues.SetValue1(const Value: string);
begin
  FLock.Enter;
  try
    FValue1 := Value;
  finally
    FLock.Leave;
  end;
end;

procedure TThreadValues.SetValue2(const Value: string);
begin
  FLock.Enter;
  try
    FValue2 := Value;
  finally
    FLock.Leave;
  end;
end;
```

However, that is not enough to protect such instances during the serialization process.

Thread-unsafe incorrect serialization

```
var
  Obj: IValues;
begin
  Obj := TThreadValues.Create;
  Obj.Value1 := 'abc';
  Obj.Value2 := '123';

  TThread.CreateAnonymousThread(
    procedure
    var
      XML: string;
    begin
      XML := ConvertXML(Obj);
    end).Start;

  Obj.Value1 := '000';
  Obj.Value2 := '444';
end;
```

You would expect the resulting XML to look like:

```
<obj>
  <value1>abc</value1>
  <value2>123</value2>
</obj>
```

You might get that result, but you also might get:

```
<obj>
  <value1>000</value1>
  <value2>123</value2>
</obj>
```

In such scenarios, we need a lock that we can apply in a broader context. This lock may be declared outside, in the context where object instances are being used, or we can expose a locking API contained inside the class. We can also use the **TMonitor** API. From a thread safety perspective, it is not important which mechanism you use, and choosing the most appropriate one may depend on other code.

Let's start with declaring the lock in the broader context, as this approach works with any kind of data, not only classes. We can also protect an otherwise thread-unsafe class with such an

external lock.

```
type
  TValues = class(TInterfacedObject, IValues)
  private
    FValue1: string;
    FValue2: string;
  published
    property Value1: string read FValue1 write FValue1;
    property Value2: string read FValue2 write FValue2;
  end;
```

Thread-safe serialization with memory leak

```
var
  Obj: IValues;
  Lock: TCriticalSection;
begin
  Lock := TCriticalSection.Create;
  // immediate initialization after object is constructed does not
  // have to be protected because at that point, the Obj instance is
  // accessible only from a single thread
  Obj := TThreadValues.Create; // or TValues.Create
  Obj.Value1 := 'abc';
  Obj.Value2 := '123';

  TThread.CreateAnonymousThread(
    procedure
    var
      XML: string;
    begin
      Lock.Enter;
      try
        XML := ConvertXML(Obj);
      finally
        Lock.Leave;
      end;
    end).Start;

  Lock.Enter;
  try
    Obj.Value1 := '000';
    Obj.Value2 := '444';
```

```
finally
  Lock.Leave;
end;
end;
```

We added a lock and we have made our serialization thread-safe, but unfortunately, this code still has some problems.

The first, glaring issue is that we have constructed the lock instance, but we haven't released it, and the above example leaks memory. The problem is that wherever we put that code, it would be the wrong place.

If we free the lock in the original thread, the background thread might still be using it.

```
...
  Obj.Value2 := '444';
finally
  Lock.Leave;
end;
Lock.Free;
...
```

If we free the lock in the background thread, the original thread might still be using it.

```
...
  XML := ConvertXML(Obj);
finally
  Lock.Leave;
end;
Lock.Free;
...
```

If we want to use code like in the previous example, we need to use a lock type with automatic memory management—a reference-counted class or record. Or we can use the **TMonitor** API. If the data instance we need to protect contains its own locking mechanism and exposes access to that lock, we can also use that lock to protect the consistency of the data in the broader context during serialization.

Thread-safe serialization with random XML result

```
var
  Obj: IValues;
begin
  Obj := TValues.Create;
  Obj.Value1 := 'abc';
  Obj.Value2 := '123';

  TThread.CreateAnonymousThread(
    procedure
      var
        XML: string;
      begin
        TMonitor.Enter(TObject(Obj));
        try
          XML := ConvertXML(Obj);
        finally
          TMonitor.Exit(TObject(Obj));
        end;
      end).Start;

  TMonitor.Enter(TObject(Obj));
  try
    Obj.Value1 := '000';
    Obj.Value2 := '444';
  finally
    TMonitor.Exit(TObject(Obj));
  end;
end;
```

The above code solves the lock memory leak, and protects the integrity of serialized data. You will no longer be able to get half-applied values in XML, like:

```
<obj>
  <value1>000</value1>
  <value2>123</value2>
</obj>
```

But, it is still possible that the resulting XML would contain

```
<obj>
  <value1>000</value1>
  <value2>444</value2>
</obj>
```

If you expected to see **abc** and **123**, then you still have a problem. While the serialization is protected from interleaved data, if you need the result to contain the first set of values, then you need to protect a much larger chunk of code, and prevent modification of the data until the serialization process is completed.

You happily move **TMonitor.Enter** out of the anonymous method to prevent the original thread from entering the lock while the background thread is still using it...

Thread-unsafe - incorrect code

```
var
  Obj: IValues;
begin
  Obj := TValues.Create;
  Obj.Value1 := 'abc';
  Obj.Value2 := '123';

  TMonitor.Enter(TObject(Obj));
  try
    TThread.CreateAnonymousThread(
      procedure
        var
          XML: string;
        begin
          try
            OutputDebugString('XML ENTER');
            XML := ConvertXML(Obj);
            Sleep(500);
            OutputDebugString('XML EXIT');
          finally
            TMonitor.Exit(TObject(Obj));
          end;
        end).Start;
  except
    TMonitor.Exit(TObject(Obj));
    raise;
  end;
```

```
Sleep(200);

TMonitor.Enter(TObject(Obj));
try
  OutputDebugString('MAIN ENTER');
  Obj.Value1 := '000';
  Obj.Value2 := '444';
  OutputDebugString('MAIN EXIT');
finally
  TMonitor.Exit(TObject(Obj));
end;
end;
```

You run the code, expecting the debug output to look like this:

```
XML ENTER
XML EXIT
MAIN ENTER
MAIN EXIT
```

But to your surprise, you get the following:

```
XML ENTER
MAIN ENTER
MAIN EXIT
XML EXIT
```

What on Earth happened here? How could you enter **MAIN** when you haven't exited **XML** yet?

Well, the problem is in the used synchronization mechanism, **TMonitor**, that protects access to the threads like other locking mechanisms. If the same thread requests access to the reentrant lock multiple times, it will be granted. In the previous example, both calls to **TMonitor.Enter** were made from the same thread.

The same would happen if you used some other kinds of locks, like, for instance, **TCriticalSection**. However, not all locks behave the same. The situation is a bit different for non-reentrant locks. But that does not mean that non-reentrant locks are the solution. If you had used **TLightweightMREW**, which is non-reentrant for write access, the above code would work on Windows, but it would cause an exception on other platforms. And on Windows it would work, not because it is designed to be used in such a way, but more as a side-effect.

There is another aspect to consider. Even if you could use locks to protect code across threads, that would still be a bad option, because this kind of approach to serialization does not scale well. Our example code is simple enough, and it can be successfully protected. But we didn't

achieve much with it anyway. We blocked all other processing while serialization is underway. Using an additional thread in such a scenario is an exercise in futility.

The real problem in such code would not be the synchronization mechanisms and their behavior, but the code logic and architecture. If we need to preserve the value of some mutable data at some point in time, we cannot expect that we will be able to successfully protect that data across threads. We can only protect it from within a single thread—if we need to preserve data originating in one thread while we are handing it over to another thread, we will have a hard time making that work properly using locking mechanisms.

Depending on the data in question, one of the solutions is making a copy of the data and then handing over that unshared copy to another thread for processing, while other threads can go on working with the original data. Of course, we will need to lock the data while we are making a copy, but that lock would be confined to a single thread, as it should be.

If, for any reason (make sure it is a really good one), the data is too complex or resource-intensive, we can still temporarily freeze the data across threads. The solution is not a standard lock, but a flag that we can flip in a thread-safe manner. Such a flag will tell us whether it is safe for other threads to access data or not. We can make our own with the **TInterlocked** API, or we can use a binary semaphore—a **TSemaphore** with a count of 1.

TSemaphore can be used across threads. But there are downsides to this approach. **TSemaphore** is not reentrant, which makes it less suitable in more complex code where the code logic may end up reacquiring the same semaphore from the same thread and causing a deadlock, unless there is another background thread running that will release the semaphore. Precisely what makes a semaphore suitable in the previous, very specific and narrow scenario, makes it extremely unsuitable to be used in others. If you do opt for a semaphore, just like with locks, you need to protect the data in all places of access.

More often than not, the need to start protecting data in one thread and extend that protection to another indicates that you need to redesign the code and rethink your approach, as you are protecting the data in too broad a context, which can be both detrimental to performance and increase the chance of thread contention or deadlocking.

When it comes to protecting the output data, there is not much difference in patterns from protecting the input data. If the result of serialization is returned as the result of a function, then, being unshared between threads, it does not require any additional protection, unless it will be assigned to an otherwise shared variable. In such a case, or if it is passed as a parameter and the passed variable can be accessed by multiple threads, then it must be protected by some mechanism.

If there is a need for some synchronization mechanism, then to minimize the time the code spends under protection, the commonly used approach is to use a temporary variable which doesn't require protection to store the result of serialization and then assign that variable to the desired one, protecting only the assignment operation, rather than the whole serialization process.

16.2 Protecting the serializer

Serialization itself, as a complex process, can involve using configuration data or intermediate results besides the original data that is being serialized. The serializer and the other supporting instances needed for the serialization process also provide a good example of how different functionality requires different thread protection mechanisms, and how not all of them are always applicable and suitable for use in every situation.

The main distinction to be made between various forms of data used for the serialization process itself, is between reusable configuration data, and data holding some intermediate state during a single serialization call that cannot be reused for another call.

Reusable configuration data is the data that can be shared among multiple threads, and as such, should and can be protected by commonly used mechanisms—immutability or locks.

Mutable intermediate state is commonly not reusable (does not support resetting to initial state) even in a single thread, and as such, is not shareable between threads. Even if such intermediate state has the ability to be reset to the initial state, adding locks around it would be an exercise in futility, and wouldn't accomplish anything because such an instance must be locked for the duration of the whole process. That would cause serious thread contention, and would prevent parallelizing data serialization. Because of that, intermediate state instances are freshly constructed for each serialization call, and discarded on exit.

Because serialization as a process can be time-consuming, and it is generally suitable for running in background threads, serialization frameworks are usually designed to be thread-safe and allow serialization to just work without the need for additional protection from the consumer's side, beyond making sure that the input and output data are thread-safe. While you could find frameworks that are not thread-safe, those are more an exception than the norm. However, beware of bugs in thread-safe frameworks.

If the framework is not thread-safe, it might be possible to use it in background threads, but also, it might not. Without knowing the exact code of the framework, it is impossible to say whether it can be used in multithreaded scenarios, or what is required to make it thread-safe.

16.2.1 Function, procedure or class method

If the serialization is contained within a global function, procedure or class method that doesn't use any global unprotected state, then the serialization process is thread-safe as long as all of the data passed in and out is also thread-safe. If the data is not thread-safe, then protecting the data is the only way to fix such code.

16.2.2 Stateless or immutable serializer instance

If the serializer comes in form of a stateless serializer instance, that does not hold any state, or is stateful and immutable once it is constructed, then the only thread safety consideration is

around that instance reference. If the reference—usually coming in the form of a singleton—is thread-safe, then it can be safely accessed from multiple threads. The thread safety of such instances has been covered in the *Class fields, singletons, and default instances* chapter.

Of course, any data or configuration that is passed as parameter to any kind of serialization methods also must be thread-safe in order to have a thread-safe serialization process.

16.2.3 Stateful serializer instance

If the serializer instance or other supporting instances stores some intermediate state that is modified during the serialization process, even just temporarily, then such instances cannot be simultaneously shared between threads.

A stateful serializer that holds only configuration data, or standalone configuration instances that will be only read and never modified during the serialization process, can be safely shared among multiple threads without any additional protection mechanism. The only protection they need would be if they are used as lazily initialized singletons, where the initialization sequence needs to be thread-safe.

But, such instances fall into the same category as **TFormatSettings**—if you need differently configured instances for different serialization scenarios, then you either have to construct and configure a new one every time when needed, or each different configuration should be stored as a separate singleton.

Commonly, serialization frameworks that need intermediate state or have stateful serializer instances also implement thread-safe global functions or class methods that construct temporary stateful instances as required by the serialization process every time such a method is called. From a thread safety perspective, if we look at those methods as black boxes, there is no difference between using such a function or class method, versus some other function or class method that doesn't require any temporary serializer instances to complete its work.

The main difference is that, in more complex serialization frameworks, developers have access to the serializer and other supporting classes required by the serialization process, and they can write their own API directly using those stateful classes. In such cases, the custom API must follow the established rules, and any deviation from expected usage—sharing between threads, or even reusing instances in the same thread—can cause issues.

16.3 Net encoding classes

Thread-safe

Classes:

- **TNetEncoding** - base abstract class

- **TCustomBase64Encoding**, **TBase64Encoding**, **TBase64StringEncoding**, **TBase64URLEncoding**
- **THTMLEncoding**
- **TURLEncoding**

Singletons:

- **TNetEncoding.Base64**
- **TNetEncoding.Base64String**
- **TNetEncoding.Base64URL**
- **TNetEncoding.HTML**
- **TNetEncoding.URL**

The encoding classes from **System.NetEncoding** are thread-safe, as either they are stateless, or they are immutable and can only be configured through the constructor. **TNetEncoding** also provides a number of thread-safe, lazily initialized singletons.

Assuming that the input data and result are thread-safe (not shared or are otherwise protected), the usage of any of the provided singletons is simple and always thread-safe:

```
var  
  Data: string;  
  Result: string;  
begin  
  Data := 'Some data';  
  Result := TNetEncoding.Base64.Encode(Data);  
end;
```

Thread-safe serialization

```
begin  
  TThread.CreateAnonymousThread(  
    procedure  
    var  
      Data1: string;  
      Result1: string;  
    begin  
      Data1 := 'First';  
      Result1 := TNetEncoding.Base64.Encode(Data1);  
    end).Start;  
  
  TThread.CreateAnonymousThread(  
    procedure  
    var
```

```
    Data2: string;
    Result2: string;
begin
    Data2 := 'Second';
    Result2 := TNetEncoding.Base64.Encode(Data2);
end).Start;
end;
```

If you need to use a differently configured instance, then you either need to make your own thread-safe singleton, or construct a local instance when needed. Because it is immutable, it can be safely shared across multiple threads as long as it is initialized only once in a thread-safe manner—either constructed before threads start running, or initialized with a thread-safe lazy initialization pattern.

Thread-safe serialization

```
var
  Encoding: TNetEncoding;
begin
  Encoding := TBase64Encoding.Create(64, #10);

  TThread.CreateAnonymousThread(
    procedure
    var
      Data1: string;
      Result1: string;
    begin
      Data1 := 'First';
      Result1 := Encoding.Encode(Data1);
    end).Start;

  TThread.CreateAnonymousThread(
    procedure
    var
      Data2: string;
      Result2: string;
    begin
      Data2 := 'Second';
      Result2 := Encoding.Encode(Data2);
    end).Start;
end;
```

16.4 JSON

Single thread-safe - some conditions apply

- classes from `System.Json` namespace
- JSON related classes from `REST` namespace

Thread-safe - some conditions apply

- JSON related singletons from `System.Json` and `REST` namespace
- JSON related class methods from `System.Json` and `REST` namespace

The JSON classes in Delphi-provided JSON frameworks fall into two categories. One are the JSON data container classes declared in the `System.Json` unit, and the rest of the classes belong to the serialization framework itself.

JSON serialization, specifically using the class methods from `System.Json.TJsonValue` and `REST.Json.TJson`, is thread-safe, whether you are serializing to or from JSON data objects. Of course, thread safety also depends on the thread safety of the input, as well as the output data involved in the process. If that data is not thread-safe, then the serialization process will not be thread-safe either.

The rest of the JSON-related classes, except shared singletons that are also thread-safe, are single-thread-safe. If you individually use any of those single-thread-safe classes, you can safely use them in the context of a background thread, but you cannot safely share them between threads.

There is a significant difference between how you can use JSON data classes and serialization-related classes. Both types contain mutable state, but there is a difference in acceptable usage.

Delphi-provided JSON serialization uses stateful instances for the serialization process. Serialization-related classes like converters, interceptors, readers, writers, and serializers, are *use once and discard* types, as they hold intermediate state used and modified during the serialization process.

Some of those classes are meant to be extended to provide custom behavior, but even in such cases, you don't generally need to deal with them during the serialization process as such, as they will be used by the public JSON API. If you are using them directly, you should follow the usage pattern from the public API: Construct new instances as you need them for serializing every new piece of data.

When you use thread-safe class methods from `REST.Json.TJson` or `System.Json.TJsonValue`, such temporary serialization instances will be constructed locally, and used only once on each method call.

The following example shows the code required for serializing data object to a JSON string, and constructing a new data object instance from that JSON string using thread-safe class methods:

Thread-safe serialization

```
var
  Obj: TFoo;
  NewFoo: TFoo;
  JSON: string;
begin
  Obj := TFoo.Create;
  Obj.Data := 'abc';
  // Object to JSON string
  JSON := TJson.ObjectToJsonString(Obj);

  // JSON string to new object
  NewFoo := TJson.JsonToObject<TFoo>(JSON);
  ...

```

If you want to use JSON data classes for more than mere serialization, you can use them just like you would use any other single-thread-safe data classes.

Note: Parsing JSON strings containing floating-point numbers on the 64-bit Windows platform is not thread-safe, because it calls the **StrToFloat** function, which is affected by the FPCR issue. See: *Floating-point control register* chapter.

Generally, if you want to check whether some code suffers from this particular issue with floating-point parsing on 64-bit Windows, the easiest way to do so, is putting a breakpoint inside **InternalTextToExtended** function in **System.SysUtils**. If you land there during a debugging session, you will immediately know that that code is not thread-safe.

In other words, if you want to use such code in background threads, or even in the main thread in multithreaded applications, without experiencing random floating-point-related problems, you will need to patch the RTL.

16.5 XML

Thread-unsafe - main thread only

- `Xml` namespace

When it comes to the thread safety of the default Delphi-provided XML serialization, it is a bit more complicated than the out-of-the-box JSON serialization. Some configuration-related parts of XML handling that could be thread-safe are not, and you must ensure not only that you modify those unsafe parts in the context of the main thread, but you also must ensure that, at that point, you don't have any running background threads that are accessing those.

Ensuring that the configuration is fully set up before you start background processing is rather simple, but unfortunately, there are some other issues preventing thread-safe parsing of XML in background threads using the framework in the `Xml` namespace. However, that does not mean that those issues cannot be fixed and patched. Of course, you can also use some other thread-safe XML framework, preferably one without bugs.

Commonly, just like JSON frameworks have data classes that can hold a JSON object structure, XML frameworks also have data classes that can hold an XML structure. Those classes are commonly single-thread-safe and you can handle them just like any other single-thread-safe data classes. The Delphi-provided XML framework uses interfaces declared in `Xml.xmldom` and `Xml.XMLIntf` to represent the XML document structure. The thread safety of code using those interfaces depends on the particular class implementations. Implementations provided in the Delphi RTL are single-thread-safe (if we ignore particular bugs), with the exception of `TXMLDocument`, which is a `TComponent`-based class. However, even `TXMLDocument` can be safely used in background threads if you don't use any of `TComponent`'s thread-unsafe features. See: *Components* chapter.

The `Xml.xmldom` unit declares standard interfaces conforming to the W3C Document Object Model (DOM) Level 2 Specification. The `Xml.XMLIntf` interfaces are simplified to enable easier handling of common tasks, and are also used by the XML Data Binding Wizard. If needed, you can retrieve standard DOM interfaces from them.

Note: Parsing floating-point numbers from XML suffers from the same FPCR issue on the 64-bit Windows platform as parsing them from JSON. See: *Floating-point control register* chapter.

Problematic code calling `StrToFloat` can be found in:

- `Xml.xmlutil.XmlStrToFloat`
- `Xml.xmlutil.XmlStrToFloatExt`

The affected procedures are generally called by code generated with the XML Data Binding wizard when dealing with floating-point data types.

If you are using other XML frameworks, and they use floating-point string parsing from `System.SysUtils`, they will also be thread-unsafe on the 64-bit Windows platform.

16.5.1 Guide for thread-safe XML serialization

If you decide to patch the bugs in the `Xml` namespace, then you can also use the Delphi-provided XML framework in a thread-safe manner, if you follow the below guidelines.

Thread-unsafe - main thread only

- DOM vendor handling from the `Xml.xmldom` unit
 - `DOMVendors`
 - `DefaultDOMVendor`
 - `RegisterDOMVendor`
 - `UnRegisterDOMVendor`
- XML schema translator registration from the `Xml.XMLSchema` unit
 - `TranslatorList`
 - `RegisterSchemaTranslator`
 - `UnRegisterSchemaTranslator`

`TDOMVendorList` holds a list of available DOM vendors. The list itself is not thread-safe, nor is the global variable `DOMVendors` that holds a reference to that list. Also, the global `DefaultDOMVendor` string variable is not thread-safe.

DOM handling does not represent an issue for XML processing if you don't modify the vendors list (register/unregister vendors)—or the default vendor—after you start processing XML in the background. All those globals will only be read during processing.

The schema `TranslatorList` is also not thread-safe. Registering/unregistering translators must be done before threads start running. If that part is honored, then `TranslatorList` will only be read in schema handling code, and therefore thread-safe.

Thread-unsafe - bug

- `Xml.xmldom.CurrentDOMVendor`
- `Xml.xmldom.GetDOMVendor`
- `Xml.xmldom.GetDOM` - calls `GetDOMVendor`
- `Xml.omnixmldom.TODOMImplementationFactory.DOMImplementation`
- `Xml.adomxmldom.TODOMImplementationFactory.DOMImplementation`

The above parts contain bugs that require fixing in order to safely use the Delphi RTL XML framework.

The first part is related to the `CurrentDOMVendor` global, which is assigned within the `GetDOMVendor` function called when the XML document is being parsed. This global serves

no purpose and can be completely removed. The rest of the `GetDOMVendor` function will be thread-safe as long as you don't modify the vendor list while background threads are running.

The next bug is in `TDOMImplementationFactory.DOMImplementation` factory functions, which lazily initialize the DOM implementation singleton in a thread-unsafe manner. You can solve that bug by explicitly calling that factory function before you start background XML processing, so that when multiple threads start using it, the singleton will be initialized. Another way to fix it is to implement thread-safe lazy initialization of such a singleton following the example from the *Class fields, singletons, and default instances* chapter.

Single thread-safe

- XML-DOM-implementing classes
- XML-schema-implementing classes
- `TXMLDocument`, as long as thread-unsafe features from `TComponent` are avoided

Serializing XML is not too different from serializing JSON. You need to ensure the thread safety of input and output data, but for the rest, the serialization process itself is thread-safe—more precisely, single-thread-safe like it has to be, and will be performed from the context of a single thread. Even if you are using other XML or JSON frameworks, or you are serializing to and from other formats, all code will roughly follow the same pattern.

Compared to JSON, there is one difference in code required for Windows platforms if you are using MS DOM-based handling backed by the COM. You need to always initialize the COM in background threads.

Thread-safe XML document parsing with COM initialization

```
var
  XML: string;
begin
  XML := '<?xml version="1.0"?><doc>...</doc>';
  TThread.CreateAnonymousThread(
    procedure
    var
      Doc: IDOMDocument;
    begin
      {$IFDEF MSWINDOWS}
      CoInitialize(nil);
      try
        {$ENDIF}
        Doc := LoadDocFromString(XML);
      ...
    end
  );
end;
```

```
{$IFDEF MSWINDOWS}
finally
  CoUninitialize;
end;
{$ENDIF}
end).Start;
end;
```

When it comes to using `TXMLDocument`, the situation is not just more complicated because it is a `TComponent` descendant class, but also because it has dual memory management, depending on the parameter passed to the constructor.

If the passed owner is `nil`, then the `TXMLDocument` instance's memory will be automatically managed by ARC, and it must be stored in an interface reference declared as `IXMLDocument`—not doing that is a very common mistake. Otherwise, it will behave just like a regular component with manual memory management.

Having an owner is the source of thread unsafety. Using an unowned `TXMLDocument` is the simplest way to avoid problems in background threads.

Thread-safe XML document parsing with `TXMLDocument`

```
var
  XML: string;
begin
  XML := '<?xml version="1.0"?><doc>...</num></doc>';
  TThread.CreateAnonymousThread(
    procedure
    var
      Doc: IXMLDocument;
    begin
      {$IFDEF MSWINDOWS}
      CoInitialize(nil);
      try
        {$ENDIF}
        Doc := TXMLDocument.Create(nil);
        Doc.LoadFromXML(XML);
        ...
      {$IFDEF MSWINDOWS}
      finally
        CoUninitialize;
      end;
      {$ENDIF}
    end).Start;
  end;
```

Chapter 17

System.Net

Single-thread-safe - specific conditions may apply

After serialization, retrieving data from the Internet is another common programming task suitable for running in background threads—on some platforms, like Android, the OS actually forbids you to perform such tasks from the main thread. Retrieving data from the Internet can take quite some time, leaving the application in an unresponsive state. Even if the data is fairly small and can be transferred at high speed under a good connection, the Internet is anything but predictable: From slow connections to slow servers, there are many potential bottlenecks and points of failure.

For a long time, the most commonly used Delphi library for handling Internet-related tasks with a variety of Internet protocols was the open-source Indy Project, which has been shipped with Delphi since version 6. Working with Indy components is covered in *Indy* chapter.

After the introduction of multiple platforms, the Delphi RTL was extended with its own framework in the **System.Net** namespace, covering the HTTP protocol. One of its most distinguishing feature is implementation of TLS/SSL connections through OS-provided APIs. This makes it simpler to use than Indy, which uses OpenSSL or requires you to provide your own OS-specific handling. There is another significant difference—Indy is built on top on the thread-unsafe **TComponent**, which complicates usage rules, whereas **System.Net** is based on **TObject**.

Besides the difference in protocol variety, the basic code workflow for both frameworks doesn't differ too much. The reason for this lies in the stateful nature of network communication, where supporting classes are single-thread-safe because they hold intermediate state which changes while the classes are being used to perform their functionality.

You can reuse an object instance in the same thread, making a new request after the previous one is completed, but you cannot share it between multiple threads. Even though reuse in the same thread is possible, it is more common to recreate the instance anyway, to prevent potential bugs with stale state.

Much like the serializers and their supporting classes, the communication classes firmly establish the pattern of having operational classes that cannot be meaningfully shared among threads. You construct the instances when needed, to perform some action, and destroy them after the action is completed. If it is not shared, it is thread-safe—provided that the class functionality doesn't otherwise access any global unprotected data in the process.

The only data you may want to share between threads will be the input data, configurations, and output data. Protecting those follows the same principles as protecting data for the serialization process. You will deal with either unshared data, read-only data, or shared data that requires protection by some synchronization mechanism.

The following example shows a simple HTTP GET request running in a background thread. The input data—a URL string—is read-only, and is handed over to the anonymous background thread. The result is a local instance of a reference-counted class, so we don't have to manage its memory—it can be processed further in the background thread or passed on to the main thread. Because the result of an HTTP request is completely independent from the HTTP client, we can safely free the client as soon as the request is completed.

Thread-safe HTTP request

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  URL: string;
begin
  URL := 'http://...';
  TThread.CreateAnonymousThread(
    procedure
    var
      Client: THTTPClient;
      Response: IHTTPResponse;
    begin
      Client := THTTPClient.Create;
      try
        Response := Client.Get(URL);
      finally
        Client.Free;
      end;
      TThread.Queue(nil,
        procedure
        begin
          Memo.Lines.Add(Response.ContentAsString);
        end);
    end).Start;
end;
```

The **THTTPClient** class has various event handlers which, if attached, will run in the context of the calling thread. If you call the **Get** method from the background thread, the handlers will also run in that background thread. This is important to keep in mind in order to apply the appropriate thread safety measures. For instance, if you are using those handlers to show progress to the user, you need to synchronize any code using the UI with the main thread.

Thread-safe HTTP progress update

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  URL: string;
begin
  URL := 'http://...';
  TThread.CreateAnonymousThread(
    procedure
    var
      Client: THTTPClient;
      Response: IHTTPResponse;
    begin
      Client := THTTPClient.Create;
      try
        Client.OnReceiveData := HTTPRequestReceiveData;
        Response := Client.Get(URL);
      finally
        Client.Free;
      end;
      TThread.Queue(nil,
        procedure
        begin
          Memo.Lines.Add(Response.ContentAsString);
        end);
    end).Start;
end;

procedure TMainForm.HTTPRequestReceiveData(const Sender: TObject;
  AContentLength, AReadCount: Int64; var AAabort: Boolean);
begin
  TThread.Queue(nil,
    procedure
    begin
      Memo.Lines.Add(Format('Received: %d of %d', [AReadCount, AContentLength]));
    end);
end;
```

HTTP requests can take some time to finish, and you may want to give users the ability to cancel such a request. You can do that by setting **AAbort** to **True**.

```
type
  TMainForm = class(TForm)
  private
    HTTPCanceled: Boolean;
  ...
end;

procedure TMainForm.CancelButtonClick(Sender: TObject);
begin
  HTTPCanceled := True;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
var
  URL: string;
begin
  HTTPCanceled := False;
  URL := 'http://...';
  TThread.CreateAnonymousThread(
  ...

procedure TMainForm.HTTPRequestReceiveData(const Sender: TObject;
  AContentLength, AReadCount: Int64; var AAbort: Boolean);
begin
  AAbort := HTTPCanceled;
  ...
end;
```

The previous examples covered usage of the HTTP GET command. Using other commands follows the same principles and thread safety patterns.

Besides **THTTPClient**, **System.Net** also has a **TComponent**-based wrapper around it in the form of **TNetHTTPClient**, which can be used at design time like any other non-visual component, which complicates its usage in background threads. To solve the conflict between drag-and-drop design and running tasks in the background, **TNetHTTPClient** also supports asynchronous mode.

The core implementation of the asynchronous mode goes back to the wrapped **THTTPClient** and its ancestor, the **TURLClient** class. If you want, you can use asynchronous mode with those classes, but it requires more boilerplate code to use.

I will give a short overview of both modes in this chapter, with a more detailed coverage of the general aspects of asynchronous mode and its downsides in the *Asynchronous Programming Library* chapter.

Because **TNetHTTPClient** is a descendant of **TComponent**, we need to construct it without an owner in order to safely use it in background threads. The functionality **TNetHTTPClient** provides is built on top of the **THTTPClient** class and there are no significant reasons why you would need to use **TNetHTTPClient** over **THTTPClient**, but if you do want to use it, the code flow will be very similar. The main functional difference is that **TNetHTTPClient** allows you to choose whether an event handler will run synchronized on the main thread, or in the context of the calling thread.

Thread-safe HTTP request

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  URL: string;
begin
  URL := 'http://...';
  TThread.CreateAnonymousThread(
    procedure
    var
      Client: TNetHTTPClient;
      Response: IHTTPResponse;
    begin
      Client := TNetHTTPClient.Create(nil);
      try
        Client.OnReceiveData := HTTPRequestReceiveData;
        // default value is True
        Client.SynchronizeEvents := False;
        Response := Client.Get(URL);
      finally
        Client.Free;
      end;
      TThread.Queue(nil,
        procedure
        begin
          Memo.Lines.Add(Response.ContentAsString);
        end);
    end).Start;
end;
```

When it comes to using **TNetHTTPClient** in asynchronous mode, we need to use an additional event handler, **OnRequestCompleted**, to let us know when the request has completed.

While it is possible to use **TNetHTTPClient** from a background thread in asynchronous mode, such usage doesn't make much sense. Asynchronous mode will start the request through a **TTask** in another background thread. Because we are already running that code in a background thread, adding another one accomplishes nothing except complicating memory management and our

code.

Because `Get` is no longer a blocking call, we cannot call `Free` like we would in synchronous code. We could free the client in the `OnRequestCompleted` event handler, but again, using asynchronous mode in a background thread is just an unnecessary complication.

Thread-unsafe - incorrect code

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  URL: string;
begin
  URL := 'http://...';
  TThread.CreateAnonymousThread(
    procedure
    var
      Client: TNetHTTPClient;
    begin
      Client := TNetHTTPClient.Create(nil);
      try
        Client.OnReceiveData := HTTPRequestReceiveData;
        Client.OnRequestCompleted := HTTPRequestRequestCompleted;
        Client.Asynchronous := True;
        // default value is True
        Client.SynchronizeEvents := False;
        Client.Get(URL);
      finally
        Client.Free; // it is wrong to free Client here
      end;
    end).Start;
end;

procedure TMainForm.HTTPRequestRequestCompleted(const Sender: TObject;
  const AResponse: IHTTPResponse);
begin
  // in cases where the SynchronizeEvents is True
  // synchronization with the main thread is not needed
  TThread.Queue(nil,
    procedure
    begin
      Memo.Lines.Add(AResponse.ContentAsString);
    end);
end;
```

If needed, you can also construct and configure `THTTPClient` and `TNetHTTPClient` instances in the main thread, and then hand them over to the background thread to perform the request. The usual rules apply—after you hand the instance over, you are not allowed to do anything with it in the original thread while the background thread is running. Handing over complicates memory management, and constructing instances in the background thread is preferred, as it makes the code cleaner and easier to follow.

Chapter 18

Asynchronous Programming Library

The Asynchronous Programming Library (APL) is a relatively new addition to the Delphi frameworks. It is consistent with the RAD approach and easy prototyping, but unconstrained use can easily lead you into programming hell.

The basic idea behind asynchronous execution provided by the APL is using a pair of methods: `BeginXXX` (`BeginOperation`), which is used to initiate an asynchronous task and returns an `IAsyncResult` interface, which can be used to query the state of the task or wait for its completion; and `EndXXX`, used to retrieve the result of that asynchronous task. If the task is already completed, `EndXXX` will return immediately. Otherwise, it will block until the task completes.

The code initiating the task in `BeginOperation` can run the task synchronously, asynchronously in the main thread—using `TThread.Queue`—or in a background thread. Obviously, running a task synchronously doesn't make much sense, but it is a possibility.

When you will call `EndOperation` depends on the particular scenario, but commonly there is an additional completion handler that runs at the end of the scheduled operation, giving us the chance to call `EndOperation` and retrieve the result immediately instead of polling for it.

Now, you may wonder how this is different from having a simple completion handler, where you don't have to call `EndOperation` and you can just retrieve the result without jumping through hoops. The difference is in flexibility, because code that uses such an API has the ability to call `EndOperation` at a different point, blocking execution until the result can be retrieved (this is similar to retrieving a future value) instead of having to act in a callback to continue some other related task—this is less important for a single task, but it can be critical for sequential processing, where you can easily end up in *callback hell*.

There are different ways you can pair calls to `BeginOperation` and `EndOperation`, and the following pseudocode of an asynchronous `TFoo` class and its usage is just one of them:

```
type
  TCompletion = procedure of object;

function TFoo.BeginOperation(OnCompleted: TCompletion): IAsyncResult;
begin
  Result := TFooAsyncResult.Create;
  TTask.Run(
    procedure
    begin
      DoOperation;
      if Assigned(OnCompleted) then
        OnCompleted;
    end);
end;

function TFoo.EndOperation(const AsyncResult: IAsyncResult): TFooResult;
begin
  AsyncResult.AsyncWaitEvent.WaitFor;
  Result := TFooAsyncResult(AsyncResult).Value;
end;

procedure TFoo.DoOperation;
begin
  ...
end;
```

```
TMainForm = class(TForm)
private
  FAsyncResult: IAsyncResult;
  ...
end;

procedure TMainForm.OnCompleted;
var
  Result: TFooResult;
begin
  Result := Foo.EndOperation(FAsyncResult);
end;

procedure TMainForm.StartBtnClick(Sender: TObject);
begin
  FAsyncResult := Foo.BeginOperation(OnCompleted);
end;
```

In order to appropriately use the APL or similar patterns and avoid pitfalls, we need to better understand its advantages and disadvantages.

APL advantages:

- easy to use in the context of RAD drag-and-drop design and prototyping - you can add a background operation for some commonly-used long running tasks with a few clicks, which significantly improves the speed of prototyping, as well as the quality of the final result, because long-running operations will not block the UI and make the application unresponsive
- useful in top-level UI classes for running asynchronous operations which are tightly coupled with the UI and its lifecycle

APL disadvantages:

- implements functionality on the wrong level of abstraction
 - violates the *Single Responsibility Principle*
 - contributes to code bloat
 - excessively complicates implementing classes
- complicated exception handling
- hard to debug except in direct top-level implementations
- tight coupling between the UI and asynchronous code
- not suitable for running multiple operations in parallel or serially (in a loop)

18.1 Wrong level of abstraction

The problem with this kind of pattern is that it is often implemented on the wrong level of abstraction and all other issues derive from that one. The ability to run some operation asynchronously is generally not the primary responsibility of some class, and adding such functionality within those classes violates the *Single Responsibility Principle*.

This kind of functionality is more appropriate in some very low-level code, or classes that interact with particular asynchronous OS APIs—like asynchronous file operations—but not as a general multithreading pattern.

Another place where such a pattern works is in final top-level classes that need to run some parts of their functionality asynchronously, where this behavior is an integral part of the class, not some optional feature.

For other classes, even when it makes sense to implement such asynchronous extensions of the class, they would be better suited for class helpers (extension methods) rather than the class

itself, but the issue with Delphi class helpers is that you can have only one helper in scope. The inability to add multiple helpers forces an overly bloated design, where every tangentially related behavior is implemented from within, and not as an extension that can be used only if needed.

Of course, you don't have to use such a feature just because it exists, but in combination with enhanced RTTI, any code that is part of the public API (and some more) will be linked in the final application regardless (unless you are using runtime packages). Helpers would prevent that unnecessary bloat.

A similar pattern influencing Delphi's implementation exists in .NET, using the `IAsyncResult` interface with a `BeginOperationName`/`EndOperationName` method pair, and is called the *Asynchronous Programming Model* (APM).

The bloat aspect is less of an issue in .NET which is bloated by design, but small deployment size was always one of Delphi's superpowers, and unnecessary bloat diminishes that power. This is something to think about if you decide to add the ability to make asynchronous calls to your own libraries, where such an ability is optional and might not be used at all.

Additionally, compared to .NET, Delphi's APL implementation is not seamlessly integrated with other asynchronous libraries, namely the *Parallel Programming Library*, which makes its core parts less reusable.

The APL and its ability to execute asynchronous operations is also implemented on the `TComponent` level with the `BeginInvoke` and `EndInvoke` methods and supporting types. One of the issues with such generalization is that it is not easily reusable if you need to support multiple operations. Adjusting behavior through inheritance is very rigid, and such an implementation either needs to predict all possible use cases—which increases complexity—or it will be usable in very narrow scenarios.

This is clearly visible in the implementations of `TNetHTTPClient`, `THTTPClient`, and other classes in the `System.Net` namespace, which don't reuse building blocks from `TComponent`. `THTTPClient` cannot use them because it is not `TComponent`-based, and `TNetHTTPClient` is not reusing them because it wraps the asynchronous functionality implemented in `THTTPClient`.

Having said that, code bloat on its own is the least of the problems with APL. Another, more serious issue with adding asynchronicity on the wrong level of abstraction is collision with the base functionality provided by other classes. Instead of having a clean and focused class that is easy to implement and maintain, such classes tend to explode. The problem becomes pretty evident if you look at the asynchronous classes in the `System.Net` namespace.

For instance, for each HTTP command, `THTTPClient` implements not only one synchronous method, but also additional `BeginXXX` methods, resulting in a total of 57 additional methods in its public API. Fortunately, there are only two `EndAsyncHTTP` methods, as all those different commands can be completed by the same method.

The number of methods is not a problem on its own, but rather the fact that this is just duplicating functionality that the end user's code can easily achieve by using the PPL and tasks directly. Also, code using tasks would be no more complicated than the one using asynchronous calls.

18.2 Complicated exception handling

With the APL and similar patterns that add additional layers of code along with a scattered workflow, you can lose exception details, as exceptions will be caught and possibly handled in code outside your reach. Whether or not this will happen depends on the particular class and pattern implementation. The ability to get all exception details for one class does not necessarily mean you will be able to get them for another.

If you need to know the exact point and cause of failure, using the APL and similar asynchronous functionality is not an option if the particular classes used will swallow the exception details. The main problem here is not that you will choose not to use the APL in the first place in cases where having precise exception details are necessary and you cannot retrieve them, but that you might choose to use the APL, and then find out you are not getting proper information long after that code ends up in production.

Even in cases where you can get complete information and the exact exception, depending on the particular class implementation, you might still have to write more code to get that information compared to writing simple `try...except` blocks around synchronous code.

For instance, if you are using `TNetHTTPClient`, you will need to attach three event handlers to handle all code paths in asynchronous mode.

```
var
  Client: TNetHTTPClient;
begin
  Client := TNetHTTPClient.Create(Self);
  Client.Asynchronous := True;
  Client.OnRequestCompleted := HTTPRequestRequestCompleted;
  Client.OnRequestError := HTTPRequestRequestError;
  Client.OnRequestException := HTTPRequestRequestException;
end;
```

18.3 Difficult to debug

Code using asynchronous classes, most notably when the pattern is buried under an additional layer—where the code does not directly invoke the `BeginOperation` and `EndOperation` methods—is hard to debug. This problem is more visible in classes that support more than one asynchronous operation and have different overloads for invoking them. `THTTPClient`, and especially `TNetHTTPClient`, are good examples of hard-to-debug code in asynchronous mode.

The main issue here is that you cannot easily find the exact places where you need to put breakpoints for inspecting the asynchronous code you’re interested in. Even when you do invoke `BeginOperation` directly, this method is just the starting point, and the actual code of interest will run asynchronously from somewhere within that method, and it can still be hard

to place breakpoints in appropriate places, minimizing the time you need to spend debugging and gathering information.

You can easily test the debugging experience with the following examples.

Synchronous HTTP request using **THTTPClient**:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  TTask.Run(
    procedure
    var
      Client: THTTPClient;
      Response: IHTTPResponse;
    begin
      Client := THTTPClient.Create;
      try
        Response := Client.Get('http://...');

      finally
        Client.Free;
      end;
      TThread.Synchronize(nil,
        procedure
        begin
          Memo.Lines.Add(Response.ContentAsString);
        end);
    end);
end;
```

Asynchronous HTTP request, using a component dropped on a form:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  Client.Asynchronous := True;
  Client.SynchronizeEvents := True;
  Client.OnRequestCompleted := HTTPRequestRequestCompleted;
  Client.Get('http://...');

procedure TMainForm.HTTPRequestRequestCompleted(const Sender: TObject;
  const AResponse: IHTTPResponse);
begin
  Memo.Lines.Add(AResponse.ContentAsString);
end;
```

In the first example you can easily follow code through the debugger: From the point you issue the GET request, until the request is successfully or unsuccessfully completed. With asynchronous code, this is not as easy. You can place a breakpoint on the line calling the `Get` method, but you will have to carefully step into that code, figuring out at runtime where you need to put an additional breakpoint in your asynchronous code so that you can debug the actual code issuing the GET request.

Since asynchronous support is baked into `THTTPClient`, you can also directly use that class to make asynchronous HTTP requests. However, you will have to write significantly more code for such requests compared to using synchronous calls in background threads or using asynchronous mode through `TNetHTTPClient`.

This example shows how much complexity is buried behind the seemingly *simple* asynchronous mode, and will also give you some additional insight on what you can expect when debugging such code:

```
TMMainForm = class(TForm)
...
public
  FAsyncResult: IAsyncResult;
  FClient: THTTPClient;
  FDest: TMemoryStream;
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  procedure CompleteRequest(const AsyncResult: IAsyncResult);
end;

constructor TMMainForm.Create(AOwner: TComponent);
begin
  inherited;
  FClient := THTTPClient.Create;
  FClient.OnReceiveData := HTTPRequestReceiveData;
  FDest := TMemoryStream.Create;
end;

destructor TMMainForm.Destroy;
begin
  if Assigned(FAsyncResult) then
  begin
    FAsyncResult.Cancel;
    FAsyncResult.AsyncWaitEvent.WaitFor;
    FAsyncResult := nil;
  end;
  FClient.Free;
  FDest.Free;
end;
```

```
    inherited;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
var
  URL: string;
begin
  if Assigned(FAsyncResult) then
    Exit;
  URL := 'http://...';
  FDest.Clear;
  FAsyncResult := FClient.BeginGet(CompleteRequest, URL, FDest);
end;

procedure TMainForm.CancelButtonClick(Sender: TObject);
begin
  if Assigned(FAsyncResult) then
    FAsyncResult.Cancel;
end;

procedure TMainForm.CompleteRequest(const AsyncResult: IAsyncResult);
var
  Response: IHttpResponse;
begin
  Response := THTTPClient.EndAsyncHTTP(AsyncResult);
  TThread.Synchronize(nil,
    procedure
    begin
      if Assigned(FAsyncResult) then
        begin
          if FAsyncResult.IsCancelled then
            Memo.Lines.Add('Cancelled')
          else
            Memo.Lines.Add(Response.StatusText);
          FAsyncResult := nil;
        end;
    end);
end;
```

18.4 Tight coupling between the UI and asynchronous code

One of the problems with the APL is that it creates a tight coupling between the UI and asynchronous code. This is by design, because the whole purpose of the APL is simplifying asynchronous execution that will not block the UI, and it can help developers avoid dealing with at least some aspects of thread safety.

Again, this is extremely useful for quick prototyping and getting as much functionality as you can with minimal effort, but there is also a price to pay, and accomplishing smooth behavior in all situations is extremely hard with an APL-based approach.

Losing control over exception handling is not the only place where one can lose critical control with the APL. Fast shutdown is another one. Because of tight coupling, asynchronous code lifecycle will match UI lifecycle. When a part of the UI owning asynchronous components is destroyed, that will trigger destruction of those components, too.

In order to have a clean shutdown, any asynchronous operation that is running at that moment has to be cancelled and waited for before the UI can be dismantled. Because destruction of the UI always happens on the main thread, that means waiting for asynchronous operations will also happen on the main thread. If an asynchronous operation cannot be terminated quickly, this will cause an unresponsive UI.

Internet communication falls into the category of such *unpredictable* operations, and while you may be able to quickly cancel such operations without causing issues during shutdown, the ability to do so depends on many factors, and there is no guarantee that your application will always behave properly in such situations.

Placing APL components in long-lived data modules with a different lifecycle can mitigate some of the problems, but it can also complicate other code logic.

While you can have similar shutdown problems in code that uses threads or tasks, it is much easier to fully decouple the UI from background operations and leave them running until they are properly terminated without waiting for their completion on the UI. One of the clear advantages of threads and tasks is that you can have supporting classes constructed locally, without the need for holding too many references in a broader scope, which complicates the memory management of the whole process, as well as shutdown and cleanup, and forces you to do all that in the context of the main thread.

For instance, if you are using tasks, you only need to store the task interface's reference. Since it is automatically managed, you don't have to worry about its memory management, and you don't have to wait for its completion during UI cleanup either.

18.5 Not suitable for running multiple operations

Another disadvantage of the APL is that it is suitable for running one asynchronous operation at a time in some code. The code required for running multiple operations in parallel, or even

serially one by one, would be significantly more complicated than simply running synchronous operations from background thread(s).

For instance, if you have a form with a search box that triggers an asynchronous search operation which is restarted each time the search string changes, then using the `BeginXX/EndXX` asynchronous pattern is appropriate, as you need to keep a single `IAsyncResult` instance along with single instances of other classes needed to run search code logic.

However, if you need to download multiple files from some list, then using asynchronous calls from `THTTPClient` would require maintaining a collection of HTTP clients, their associated asynchronous results and streams, and coordinating all of them in a monstrous pile of scattered code. While using asynchronous calls from `TNetHTTPClient`—which hides some of the complexity—would be a simpler solution, it is still more complex than using any other option—a single background thread or task with a loop, running multiple tasks, or using `TParallel.For`.

How to parallelize background work, as well as choosing the most appropriate solution, has been explained in the *Parallel collection processing* chapter. Downloading multiple files is just a variant of such processing. This is a task that comes up often enough in real-life scenarios, and showing more specific examples can help you understand how to parallelize other common tasks.

The examples show a basic parallelization workflow, and don't fully handle other requirements like cancelling or tracking progress. Those parts can be easily added from other examples showing how to solve those specific requirements.

Because using asynchronous calls with `TNetHTTPClient` is a suboptimal solution to the problem, none of the examples show how to do that. This is another good example of an algorithm that cannot be easily parallelized, and in such situations it is always more productive to change approach than try to brute-force an inappropriate solution. Yes, if you try hard enough, you can get *working* code out of almost anything, but that does not mean you should do that. More complicated solutions usually have more bugs, and are hard to understand and therefore change and maintain.

Moving asynchronous code that is more tightly coupled with the UI to background threads or tasks also requires decoupling the input and output data from the UI. When `TNetHTTPClient` is used in asynchronous mode, you can freely populate all input data directly from UI controls without any intermediate data containers. For instance, you can directly use an edit control to specify download URL, or in the case of multiple downloads, some list control.

Because we cannot use UI controls in background threads without synchronizing their access with the main thread, we need to store all needed input data in some UI-independent storage. This can be a simple string or record array, or even a collection of object instances for more complex data. We also need to handle storage for output data.

In the following download examples, all required input and output data will be represented by a record, stored in a dynamic array:

```
type
  TDownloadData = record
    // input
    URL: string;
    // output
    FileName: string;
  end;
```

Thread-safe download of multiple files using a single task with a loop

```
var
  List: array of TDownloadData;
begin
  // populate List before processing
  ...
  TTask.Run(
    procedure
      var
        Data: TDownloadData;
        Client: THTTPClient;
        Response: IHTTPResponse;
        Stream: TFileStream;
      begin
        for Data in List do
          begin
            try
              Client := nil;
              Stream := TFileStream.Create(Data.FileName,
                fmCreate or fmShareExclusive);
              try
                Client := THTTPClient.Create;
                Response := Client.Get(Data.URL, Stream);
                // handle Response if needed
                ...
              finally
                Stream.Free;
                Client.Free;
              end;
            except
              // handle exceptions
              ...
            end;
          end;
      end;
```

```
TThread.Synchronize(nil,
  procedure
  begin
    // handle status for each file
  end);
end;
end;
```

Thread-safe download of multiple files using multiple tasks

```
procedure DownloadTask(const URL, FileName: string);
begin
  TTask.Run(
    procedure
    var
      Client: THTTPClient;
      Response: IHTTPResponse;
      Stream: TFileStream;
    begin
      try
        Client := nil;
        Stream := TFileStream.Create(FileName, fmCreate or fmShareExclusive);
        try
          Client := THTTPClient.Create;
          Response := Client.Get(URL, Stream);
          // handle Response if needed
        ...
      finally
        Stream.Free;
        Client.Free;
      end;
    except
      // handle exceptions
    end;
    TThread.Synchronize(nil,
      procedure
      begin
        // handle status for each file
      end);
    end);
  end;
```

```
var
  List: array of TDownloadData;
  Data: TDownloadData;
begin
  // populate List before processing
  ...
  for Data in List do
    DownloadTask(Data.URL, Data.FileName);
end;
```


Chapter 19

Indy

Single-thread-safe - specific conditions apply

Following RAD principles, the Indy classes are also built on top of the thread-unsafe `TComponent`, but they can be safely used in background threads, provided that the components are not simultaneously used from multiple threads, and that other unsafe parts of `TComponent` are avoided.

You can construct the component you need in a background thread when you need it, following the same pattern as with `System.Net` classes. This approach is commonly used with Indy client components which usually perform some short-lived operations. Server components have a different lifecycle. If you are making repeated requests, you can also reuse client components, as long as you don't start a new request in a different thread before the previous one is finished.

If you are using components that are dropped on a form or data module, you need to make sure such a container is not released while the component is doing some work in a background thread, and the component shouldn't be used in any other way from the main thread during that time.

Similarly to `System.Net`, Indy also has a number of event handlers that can be attached to monitor progress or other significant events during an operation. If the operation is called from a background thread, then those events will fire in the context of that background thread, and any logic that needs to run in the context of the main thread, like updating progress on the UI, needs to be synchronized.

Indy components use blocking calls, and that makes using them rather simple and straightforward, as you don't have to scatter code around, and all code logic can be easily followed. To keep the application responsive during such blocking calls made from the main thread, Indy has the `TIdeAntiFreeze` component that, when dropped on the form or data module, keeps pumping the Windows message queue. However, this component is neither needed nor suitable for use when Indy operations are being called from background threads, and you **must** remove it.

Thread-safe HTTP request

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  URL: string;
begin
  URL := 'http://...';
  TThread.CreateAnonymousThread(
    procedure
    var
      Client: TIIdHTTP;
      Response: string;
    begin
      Client := TIIdHTTP.Create(nil);
      try
        Client.OnWorkBegin := HTTPWorkBegin;
        Client.OnWork := HTTPWork;
        Client.OnWorkEnd := HTTPWorkEnd;
        Response := Client.Get(URL);
      finally
        Client.Free;
      end;
      TThread.Queue(nil,
        procedure
        begin
          Memo.Lines.Add('HTTP Response: ' + Response);
        end);
    end).Start;
end;

procedure TMainForm.HTTPWorkBegin(ASender: TObject; AWorkMode: TWorkMode;
  AWorkCountMax: Int64);
begin
  TThread.Queue(nil,
    procedure
    begin
      Memo.Lines.Add('HTTP Work Begin ' + AWorkCountMax.ToString());
    end);
end;

procedure TMainForm.HTTPWork(ASender: TObject; AWorkMode: TWorkMode;
  AWorkCount: Int64);
begin
```

```
TThread.Queue(nil,
  procedure
  begin
    Memo.Lines.Add('HTTP Work ' + AWorkCount.ToString);
  end);
end;

procedure TMainForm.HTTPWorkEnd(ASender: TObject; AWorkMode: TWorkMode);
begin
  TThread.Queue(nil,
  procedure
  begin
    Memo.Lines.Add('HTTP Work End');
  end);
end;
```

Regardless of which Indy client component you use, the above pattern is an appropriate one. Of course, for input and output data that might be shared among multiple threads, the base thread safety rules are still valid—if you share the data, you either need to use additional protection mechanisms, or such data must be used as read-only in all threads during the time multiple threads have access to the data.

Unlike discardable client components, Indy server components are meant to be used as long-lived instances. They are also single-thread safe in the sense that you cannot share the same server component between multiple threads, nor do you need to do so, because servers spawn their own background threads to perform operations, and they manage those threads in a thread-safe manner.

Because server components handle their own threads, they are commonly constructed in the main thread, usually as part of a data module, where you can take advantage of RAD design. However, some of the server event handlers attached in the main thread will run from the context of the background threads. Any code in such event handlers can be run simultaneously from different background threads, and you must use synchronization or locking mechanisms to protect any shared data you use in those. Also, any code that can only run in the context of the main thread must be synchronized.

Chapter 20

REST

Single-thread-safe - specific conditions apply

Delphi's REST framework provides another set of examples where the thread-unsafe **TComponent** is used as a base for performing operations that are suitable for running in background threads. More than suitable, in fact—it is highly recommended.

Unlike the framework in the **System.Net** namespace, the REST framework is implemented directly in **TComponent** descendant classes, which complicates their handling in background threads. Being descended from **TComponent** is not a major obstacle, as we could see in the examples from the Indy framework, but the REST client is not just built on top of **TComponent**, it is built on top of LiveBindings classes, which pose a much greater threat to thread safety.

However, if the LiveBindings feature is not used for mapping REST data, then the REST components follow the same pattern as **TNetHTTPClient** or Indy components: You can use components constructed in the main thread or those placed on forms or data modules, or you can construct and configure them in a background thread. In the event of a handover, you need to ensure their owner is not destroyed while the background thread is using them, and make sure not to simultaneously access those components from multiple threads, as they are not thread-safe.

If you want to run multiple requests from multiple threads, you need to have dedicated REST components for each thread. Usually, the best way to deal with such scenarios is constructing components in background threads to prevent accidental misuse.

Keep in mind that all components required to perform REST requests are stateful and cannot be shared. For instance, you cannot share an authorization component between REST clients that work from different threads.

Using a **TRESTClient** as the owner of other required components simplifies memory management, because freeing the client will automatically free all the other components. Because of that, we need to be careful about those components' lifetimes. In the following example, we must

use `TThread.Synchronize`, because using `TThread.Queue` would cause us to access the `Response` component after it was destroyed:

Thread-safe REST request

```
TThread.CreateAnonymousThread()
procedure
var
  Client: TRESTClient;
  Request: TRESTRequest;
  Response: TRESTResponse;
  Auth: THTTPBasicAuthenticator;
begin
  Client := TRESTClient.Create(nil);
  try
    Auth := THTTPBasicAuthenticator.Create(Client);
    Request := TRESTRequest.Create(Client);
    Response := TRESTResponse.Create(Client);
    Auth.Username := ...;
    Auth.Password := ...;
    Client.BaseURL := ...;
    Request.Resource := ...;
    Request.Client := Client;
    Request.Response := Response;
    Request.Execute;
    TThread.Synchronize(nil,
      procedure
        begin
          Memo.Lines.Add(Response.Content);
        end);
    finally
      Client.Free;
    end;
  end).Start;
```

The previous example doesn't have any events attached. If you attach them, depending on their code, you may want to automatically synchronize their execution with the main thread. You can do that by setting the `SynchronizedEvents` property in the `TRESTClient` and `TRESTRequest` classes to `True`.

If you are using LiveBindings in combination with the REST framework, you need to pay special attention and avoid triggering the LiveBindings mechanism from a background thread. That means you will need to use `SynchronizedEvents` and configuring bindings also needs to be done in the context of the main thread.

In some simple scenarios, you can get away with the main thread requirement for LiveBindings configuration, but you need to carefully inspect such code, making sure that it does not access any of the LiveBinding singletons—global or class variables which are not thread-safe and cannot be used from background threads.

For asynchronous execution you can also use the `ExecuteAsync` method in the `TRESTRequest` class, which returns a custom `TRESTExecutionThread` thread. While that custom thread is running, you must not start another request with the same request instance, nor modify any of the other connected REST components. This is not too different from handing a request variable over to a thread or a task.

If you use such asynchronous execution, you need to pay attention to the lifetime of REST components, to avoid their destruction before the thread has finished running. This behavior is different to the behavior of APL components, which hold an `IAsyncResult` reference and use it to wait for completion before the component is destroyed. The REST request does not hold reference to the constructed thread, so you will have to manage lifetime and cleanup yourself.

Constructing components in the main thread and using `ExecuteAsync` also allows for the safe usage of LiveBindings, provided that you pass `True` to the `ASynchronized` parameter.

```
function TRESTRequest.ExecuteAsync(ACompletionHandler: TCompletionHandler = nil;
  ASynchronized: Boolean = True; AFreeThread: Boolean = True;
  ACompletionHandlerWithError: TCompletionHandlerWithError = nil): TRESTExecutionThread;
```

Thread-safe asynchronous REST request

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  RESTRequest.ExecuteAsync(
    procedure
    begin
      Memo.Lines.Add(RESTResponse.Content);
    end,
    True, True,
    procedure(Error: TObject)
    begin
      Memo.Lines.Add(Exception(Error).Message);
    end);
end;
```

If you use an auto-destroying thread (pass `True` to `AFreeThread` parameter), you must not store the returned thread instance, as it is not thread-safe to use it. If you need more control over the thread by using the returned thread instance, you will have to free it yourself. Because

completion handlers run from the thread's **Execute** method, you will need to queue the releasing of the thread. If you use synchronized handlers, that also means using **ForceQueue** to avoid deadlocks, as **Queue** would just try to run the code without queueing if it detected that it was being called from the context of the main thread.

Thread-safe asynchronous REST request

```
TMainForm = class(TForm)
private
  FThread: TThread;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
begin
  if Assigned(FThread) then
    Exit;

  FThread := RESTRequest.ExecuteAsync(
    procedure
    begin
      Memo.Lines.Add(RESTResponse.Content);
      TThread.ForceQueue(nil,
        procedure
        begin
          FreeAndNil(FThread);
        end);
    end,
    True, False,
    procedure(Error: TObject)
    begin
      Memo.Lines.Add(Exception(Error).Message);
      TThread.ForceQueue(nil,
        procedure
        begin
          FreeAndNil(FThread);
        end);
    end);
end;
```

Chapter 21

Regular expressions

Single-thread-safe

Support for regular expressions in the Delphi RTL is built as a wrapper on top of the PCRE library. To determine the thread safety and appropriate usage of that implementation, we need to start by determining the thread safety of the underlying PCRE library.

The PCRE documentation on multithreading states:

The PCRE functions can be used in multi-threading applications, with the proviso that the memory management functions pointed to by `pcre_malloc`, `pcre_free`, `pcre_stack_malloc`, and `pcre_stack_free`, and the callout and stack-checking functions pointed to by `pcre_callout` and `pcre_stack_guard`, are shared by all threads.

The compiled form of a regular expression is not altered during matching, so the same compiled pattern can safely be used by several threads at once.

If the just-in-time optimization feature is being used, it needs separate memory stack areas for each thread. See the `pcrejit` documentation for more details.

The Delphi wrapper handles the first requirement, and the JIT option is not supported by Delphi out of the box. We already know that calling functions is thread-safe as long as those functions don't access any unprotected global state, and as long as the passed parameters are also thread-safe—either unshared, read-only, or protected by additional mechanisms. The PCRE documentation extends those basic thread safety rules to the compiled pattern, which can be safely shared across multiple threads without additional protection, because the PCRE functions will not modify that pattern in any way.

The above is valid for calling the raw PCRE API. Now we can explore the thread safety of the Delphi implementation. Similarly to serialization frameworks or Internet connectivity classes, the classes and other supporting types for processing regular expressions are stateful, and there-

fore single-thread-safe. You cannot share instances of those types between multiple threads. There are also a number of class functions that are thread-safe, because they construct all necessary instances as temporary local variables.

Single-thread-safe

- **TPerlRegEx**
- **TRegEx**
- other types and classes from the **System.RegularExpressionsXXX** namespace

Thread-safe

- class functions from **TPerlRegEx** and **TRegEx**

One specific part that is thread-safe in the PCRE library which you might want to use is sharing compiled patterns between threads. However, the RegEx pattern is encapsulated within the **TPerlRegEx** class along with other thread-unsafe data used during processing, so you cannot take advantage of this one thread-safe piece and share that piece of data between threads. If you want to do so, you would have to create your own wrapper types and functionality around the exposed PCRE API from **System.RegularExpressionsAPI** and separate the pattern from the rest of the data.

Thread-unsafe - incorrect code

```
var
  Reg: TRegEx; // shared instance is unsafe
begin
  Reg := TRegEx.Create('ab*c');
  TThread.CreateAnonymousThread(
    procedure
    begin
      if Reg.IsMatch('abcfoo 123 abac458') then
        ...
    end).Start;
  TThread.CreateAnonymousThread(
    procedure
    begin
      if Reg.IsMatch('foo') then
        ...
    end).Start;
end;
```

Thread-safe correct code

In contrast to the previous thread-unsafe example, where the unsafety is caused by the shared **Reg** instance, each thread in the following thread-safe example has its own locally declared and constructed **TRegEx** instance:

```
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      Reg: TRegEx; // thread local instance is safe
    begin
      Reg := TRegEx.Create('ab*c');
      if Reg.IsMatch('abcfoo 123 abac458') then
        ...
    end).Start;

  TThread.CreateAnonymousThread(
    procedure
    var
      Reg: TRegEx; // thread local instance is safe
    begin
      Reg := TRegEx.Create('ab*c');
      if Reg.IsMatch('foo') then
        ...
    end).Start;
end;
```

Handing the regex instance over to another thread is thread-safe, as long as multiple threads don't use the instance simultaneously:

```
var
  Reg: TRegEx;
begin
  Reg := TRegEx.Create('ab*c');
  TThread.CreateAnonymousThread(
    procedure
    begin
      if Reg.IsMatch('abcfoo 123 abac458') then
        ...
    end).Start;
  // Reg must not be used after this point
end;
```

Thread-safe usage of class functions

Using the **TRegEx** class functions is thread-safe, provided that the passed parameters and result variables follow the established thread safety rules, being either: unshared, read-only, or protected by additional mechanisms.

```
var
  a: TArray<string>;
begin
  a := TRegEx.Split('abcfoo 123 abac458', 'ab*c');
end;
```

Part 4. Visual Frameworks - VCL and FMX

Chapter 22

Visual framework components

The general rules for determining whether a type is thread-safe in visual frameworks like VCL and FMX can be summed up in few questions:

- Does the class (or type) represent or wrap a visual control?
- Does the class interact with a visual control?
- Is the class part of a visual class hierarchy?

If the answer to any of the above is ‘yes’, then the type is not thread-safe, and can only be used in the context of the main thread. Even if the answer is ‘no’, that doesn’t mean that such a type is thread-safe, as visual frameworks are not thread-safe by design, and very few related types have any thread safety features built in. Basically, you should start from the assumption that any type belonging to visual frameworks is not thread-safe.

This part of the book covers the thread safety of the classes that can only be used in the context of the main thread. This may sound a bit confusing, because if a class can only be used in the main thread, there is not much else to be said about that class, and any access to instances of those classes must happen from the context of the main thread. Either by being used directly from code that runs in the main thread, or synchronized with the main thread using the `TThread.Queue` or `TThread.Synchronize` methods.

But, just like some `TComponent`-based classes can, under specific conditions, be safely used in the context of a background thread, some of the classes and types belonging to visual frameworks can be used in background threads, too. And this is what this part of the book is all about. Finding which parts of the visual frameworks can be used from background threads, and under which conditions. This will also include workarounds for some classes that could be thread-safe, but are not thread-safe because of some implementation details, as well as code that has additional requirements in order to work properly in background threads, like using the Windows API with thread affinity.

Besides non-visual components that can be used in background threads under some circumstances, there are also some components and classes that are too tightly coupled with the main thread, and just like visual controls, cannot be used from background threads.

Some types are single-thread-safe on their own. Some are not and can only be used in the main thread, and some can only be used in background threads under very specific conditions. Which category each type belongs to can be determined by inspecting their implementation or reading the documentation. Beware of bugs, though, because even though some parts of the visual frameworks are documented as being thread-safe, not all documentation is aligned with reality. Because those thread-safe parts need to interact with the thread-unsafe parts of the visual frameworks, it is also rather easy to make mistakes in your own code and use those types in a thread-unsafe way.

As one of the base classes in RTL, the general aspects of `TComponent`'s thread safety and the conditions under which a `TComponent`-based class can be used in background threads has been covered in the *Components* chapter. Part 3 of the book also covered some more specific non-visual `TComponent`-based classes and their thread-safe usage.

Thread-unsafe - main thread only

- LiveBindings
- `VCL.Controls.TControl`
- `FMX.Types.TFmxObject`
- other non-visual classes that are tightly coupled with the visual frameworks, like `TMenuItem`, `TMenu`, `TAction`, and similar
- global instances like `Application`, `Screen`

Chapter 23

LiveBindings

Thread-unsafe - main thread only

The primary purpose of LiveBindings is providing a mapping and updating mechanism between data and visual controls. Because visual controls are not thread-safe, using LiveBindings is not thread-safe either.

However, the ability to create a binding is not restricted to any base class, and you can create bindings between any two classes you like. That raises the question of whether you can create and use bindings between such classes in background threads. LiveBindings classes are generally single-thread-safe (including the ones that descend from `TComponent` and its thread safety considerations), so using LiveBindings to map between non-visual classes that all belong to the same background thread is probably possible.

I am saying probably, because the LiveBindings framework also contains plenty of thread-unsafe singletons that are used from otherwise seemingly (as far as declarations are concerned) single-thread-safe classes. Because the code paths that may touch those singletons in a thread-unsafe manner span across multiple method calls and classes, it can be hard to determine whether there is a fully thread-safe code path, and what the basic requirements are to follow it.

In other words, if LiveBindings as a framework had been designed for usage in background threads, then those singletons should be thread-safe, too. They are not, and when all of the above is taken into account, it is safe to say that LiveBindings are meant to be used only in the context of the main thread.

As thread safety is very code- and context-sensitive, it is always possible to achieve thread safety in some very specific code, but I will not investigate those possibilities here, because the potential benefits are minimal, and the risks of stepping into unsafe code are too great.

Besides LiveBindings, any other data binding framework or code used to populate visual controls will not be thread-safe in those parts where it accesses visual controls. Whether other parts of those frameworks or code that are not directly interacting with visual controls are thread-safe depends on the specific framework or code. Determining the thread safety of such frameworks can be a daunting task, but if you need to do that regardless, there are a few things to keep in mind:

- Read the framework documentation. If there is no mention of thread-safe usage and examples, treat the framework as thread-unsafe—you will have to make a detailed inspection of the framework code in order to determine whether you can use it safely and under which circumstances. If the framework is thread-safe, any deviation from documented usage can result in thread-unsafe code—even in thread-safe classes and frameworks, not everything should and can be fully thread-safe.
- Most classes in binding frameworks, but also in other kind of frameworks, will be single-thread-safe, and as long as you don't share their instances between threads, you will be able to use those safely.
- Most of those single-thread-safe classes cannot be shared between threads in any meaningful way, and trying to make them thread-safe is an exercise in futility.
- Singletons and other shared classes must be thread-safe, or must only be used in read-only mode when shared among threads. If such classes are not thread-safe, this is usually the best signal that the framework is not thread-safe, or can be used in background threads only under very specific circumstances.
- Pay attention to the code that accesses singletons, especially on whether modifying operations can be easily triggered from background threads. If this is the case, it will be very hard to safely use those classes in background threads.
- Any interaction with visual controls or non-visual components that are part of the visual hierarchy must be performed from the context of the main thread. There are no exceptions to this rule.
- The guidelines and coding patterns throughout the book, especially those mentioned in *Part 3*, and more specifically the *Serialization* chapter, are universal and can easily be followed to determine and ensure thread safety in broad scenarios, as well as for data binding.

Chapter 24

VCL and FMX controls

Thread-unsafe - main thread only

Visual VCL controls, the descendants of `VCL.Controls.TControl`, can only be used in the context of the main thread, and this restriction includes component streaming. They can only be streamed in the main thread. This is also valid for components and classes that control some aspects of visual controls or wrap them, but are not direct descendants of the `TControl` class, like `TMenuItem`, `TMenu`, `TAction`, `TActionList`, and similar classes.

In FireMonkey, `FMX.Types.TFmxObject` is the base class for FireMonkey controls and other non-visual components that are tightly coupled with the visual framework. Using any descendant of `TFmxObject` in background threads is not safe.

Additionally, regardless of its ancestry, any class that implements `IFreeNotification` is also not thread-safe, due to the inner workings of FireMonkey's specific free notification system. However, as with the `TComponent` notification system, if your code doesn't trigger that notification system across thread boundaries, then that part alone will not cause concurrency issues.

If you think you can workaround the thread safety restrictions on visual components and controls, you might get some small piece of code that seemingly works, but this will be just an illusion. Visual controls are simply not designed to be accessed from background threads, and no amount of wishful thinking will change that fact.

The global singletons `Application` and `Screen` are also not thread-safe. Not only are those singletons directly exposed as global variables that can be freely written to, but the `TScreen` and `TApplication` classes in both VCL and FMX are not thread-safe either.

The application and screen globals in VCL are initialized and finalized in the `Vcl.Controls` unit's initialization section.

Besides the `Application` and `Screen` globals, FireMonkey also has additional platform-specific globals that represent a given platform and its services. The application and screen globals

are constructed and destroyed inside the platform global instance, which is initialized in the `FMX.Platform` unit's initialization section. Since the platform globals are reference-counted instances, their memory will be automatically managed, and they will be destroyed after the finalization section of a unit where they are declared runs.

The `Application` instance has a few properties you might be tempted to use in background threads, like the `Terminated` property or the `Handle` property. This is not advisable, not because you will not be able to use `Terminated` and similar properties in a thread-safe manner—assignment is an atomic operation for those simple types—but because there is no guarantee that the `Application` instance will be valid if you try to access it during application shutdown.

If you can guarantee that threads accessing any of those properties will completely stop running before the global instance is destroyed, then you can safely read such properties as long as your code logic is valid, even if the value of the property changes immediately after you have read it and before you could do anything else with the retrieved value. However, when you satisfy this requirement, using `Application.Terminated` in a background thread will not give you any useful actionable information.

Sometimes it is not thread safety alone that will prevent you from using some value, but the desired functionality. The need to access global variables belonging to visual frameworks is also a pretty good signal that your code is doing something wrong.

Chapter 25

Interactions with OS APIs and frameworks

Some of the non-visual components, classes, and routines interact with various OS APIs and frameworks. Some of those can be used in background threads, some have specific usage rules, and some are mostly used in GUI contexts, and even though they are more universal, the GUI connection imposes additional limitations. Sometimes those rules and limitations stem directly from implementation of the API or of the Delphi wrapper class

The thread safety aspects of OS APIs and frameworks are commonly well-documented by OS vendors, and if you have any doubts about some particular use case, the official documentation will most certainly give you the answers you seek. Some of the more commonly used non-visual APIs and their implementing Delphi classes, like files and sockets (networking), were covered in previous chapters. We already know that visual controls and related components are not thread-safe, graphics processing is covered in *Part 5*, which leaves us some non-visual APIs that are commonly used in a GUI context, even though they are not strictly GUI-related, and some that have a deeper connection with the GUI than it seems at first glance.

Because Delphi originated on the Windows platform and it is its most commonly used one, I will cover interacting with Windows in more detail. When it comes to other platforms, there are similar concepts in play, though they don't have to necessarily involve all the similar OS APIs and entities in the same way. In this context, "similarity of concepts" means that you can read the OS documentation, and when you know what the acceptable usage is, you will be able to translate that into appropriate Delphi code following the thread-safety patterns that have been covered across the book, and more specifically in the following chapters that deal with the Windows API.

25.1 Window allocation and de-allocation

In the context of the Windows OS, a window as an entity is the basic building block of a GUI. However, windows can also be used in background threads. While the OS allows using any kind of window in a background thread as long as some rules are followed, such usage is not permitted in the context of Delphi visual controls, and using windows in background threads is restricted to non-visual windows or sending messages to visual window handles, which is supported by the thread-safe OS-provided window messaging system.

When it comes to using windows in background threads, there are two thread safety aspects: The first is window thread affinity, and the second is the thread safety of the window allocation and de-allocation process.

Window thread affinity means that the thread where a window is created *owns* the window. Any changes to window parameters (state) must be done from that thread. Also, the window will belong to a message queue associated with that thread. More precisely, creating a window in a thread will automatically construct a message queue for that thread.

Constructing a message queue also obligates the developer to process messages in that queue.

Because of that, you cannot create a window in one thread and hand it over for usage to another one. This is also one of the reasons why visual controls and related components are restricted to the main thread. Even constructing them in a background thread is not possible because you can easily trip over code that triggers the creation of a window handle and it is game over when that happens in the *wrong* thread.

What about creating a window in a background thread on purpose?

Creating a window in a background thread and using it there is a valid option, and there are situations where you will want to do that. There are two routines in `System.Classes` that simplify the process: `AllocateHWnd` and `DeallocateHWnd`. Because they are not part of the VCL namespace and they don't seem in any way related to any visual class, and because Windows allows constructing windows in background threads, one can easily make a mistake and think that those two routines are thread-safe. They are not.

The main source of unsafety comes from two other routines: `MakeObjectInstance` and `FreeObjectInstance`, called from within, that are not thread-safe because they handle global variables in a thread-unsafe manner. Because of that, all those routines can only be used from the context of the main thread. And that detail is our problem. In order to create and destroy a window that belongs to a thread, we need to call `CreateWindowEx` and `DestroyWindow` from that thread, but we cannot do that because `AllocateHWnd` and `DeallocateHWnd` are tied to the main thread.

Another piece of thread-unsafe code in the `AllocateHWnd` function is registering the window class. That one is less critical, and will not cause troubles if the window class has already been registered. Still, that code belongs to the main thread.

Thread-unsafe - main thread only

```
function AllocateHWnd(const AMethod: TWndMethod): HWnd;
procedure DeAllocateHWnd(Wnd: HWnd);

function MakeObjectInstance(const AMethod: TWndMethod): Pointer;
procedure FreeObjectInstance(0bjectInstance: Pointer);
```

There is a fairly simple solution to the above issue that will require a bit of copy-pasting and tweaking code. If we synchronize the parts of the code that need to run in the main thread, and leave out the code that needs to run in the context of the background thread, we can solve our problems.

Thread-safe

```
function SafeAllocateHWnd(const AMethod: TWndMethod): HWnd;
var MethodPtr: IntPtr;
...
begin
  TThread.Synchronize(nil,
    procedure
      begin
        // register window class
        ...
        // create method instance
        MethodPtr := IntPtr(MakeObjectInstance(AMethod));
      end);
  Result := CreateWindowEx(...);
  if Assigned(AMethod) then
    SetWindowLongPtr(Result, GWL_WNDPROC, MethodPtr);
end;

procedure SafeDeAllocateHWnd(Wnd: HWnd);
begin
  ...
  if Instance <> @DefWindowProc then
    TThread.Synchronize(nil,
      procedure
        begin
          FreeObjectInstance(Instance);
        end);
end;
```

25.2 Windows messaging

Sending and posting messages in Windows is thread-safe and can be done from any thread. If your code's functionality depends on the window receiving the message, you will have to make sure that the window handle is valid when you send the message, but sending a message to an invalid handle will not cause any other issues.

SendMessage and **SendMessageTimeout** are blocking calls, and they will not return until either the message is processed by a receiving window, or the message times out. When you send messages across threads, the message is not processed immediately, but only when the receiving thread invokes message retrieval code. In other words, the receiving thread will not be interrupted in the middle of some work.

The rest of the messaging API (both send and post) will put a message in a queue and return immediately. **PostMessage** will post the message to a queue belonging to the thread in which the specified window is created, and **PostThreadMessage** will post the message to a queue belonging to the specified thread.

Generally, OS messaging systems are thread-safe, and they allow messaging across threads. However, you still need to take care of the thread-safety of the code that runs when the message is received, and that any potentially shared data is protected. For data protection, the general thread safety rules apply, and if you are concerned about the particular behavior of some OS API you're calling, the official documentation is the best place to find out all the ins and outs.

Using OS messaging systems for thread-safe messaging between threads is a common practice. But because such messaging is tightly coupled with the OS API, which varies greatly between platforms, it is not practical for cross-platform development. In such cases, using a platform-independent messaging system is warranted. See: The *Event bus* chapter.

If you only need to support a single platform, then using the OS messaging system will do just fine, unless you require other custom features that are not available in the OS out of the box.

In Windows, you can safely communicate with the main thread by sending messages directly to some window handle or the main thread itself. If you are sending messages to window handles captured by the anonymous method variable capturing mechanism, the handle value will be valid even if the window is recreated while the thread is running—be aware that there is always a very small timeframe during window recreation when the handle can become invalid. However, if you use custom threads where the handle is passed as a parameter, or your code cannot tolerate even the slightest chance of using an invalid handle, then you can only use handles that are not subject to change.

Any of those messages will be processed in the context of the receiving thread, not the thread that sent the message.

```
TThread.CreateAnonymousThread(
  procedure
  begin
    SendMessage(MainForm.Handle, WM_USER, 0, 0);
    PostMessage(MainForm.Handle, WM_USER, 0, 0);
    PostThreadMessage(MainThreadID, WM_USER, 0, 0);
  end).Start;
```

With thread-safe window handle allocation/de-allocation, you can send messages in the other direction—from the main thread to the background, or between two background threads.

The following example shows a custom thread class with an associated window handle, doing some work in addition to the obligatory message pumping:

```
type
  TWindowThread = class(TThread)
protected
  FWnd: HWND;
  procedure WndProc(var Message: TMessage);
  procedure Execute; override;
public
  property Wnd: HWND read FWnd;
end;

procedure TWindowThread.Execute;
var M: MSG;
begin
  FWnd := SafeAllocateHWnd(WndProc);
  try
    while not Terminated do
      begin
        // do some work
        ...
        // process messages
        while PeekMessage(M, 0, 0, 0, PM_REMOVE) do
          begin
            TranslateMessage(M);
            DispatchMessage(M);
          end;
      end;
  finally
    SafeDeallocateHWnd(FWnd);
  end;
end;
```

```
procedure TWindowThread.WndProc(var Message: TMessage);
begin
  OutputDebugString('Message received');
  DefaultHandler(Message);
end;
```

And then you can send or post messages to such a thread using its `Wnd` property:

```
var
  Thread: TWindowThread;
begin
  Thread := TWindowThread.Create;
  ...
  PostMessage(Thread.Wnd, WM_USER, 0, 0);
```

25.3 Timers

The connection between non-visual components and the UI is rather obvious for some components; for others, not so much. Timers are one such component type, that has deep ties not only with the thread it is constructed on, but also the main thread in particular. All timer implementations in Delphi, regardless of any visual frameworks or platforms, can only be used on the main thread.

Thread-unsafe - main thread only

- `Vcl.ExtCtrls.TTimer`
- `FMX.Types.TTimer`
- `IFMXTimerService`

While it is possible to implement timer functionality in a background thread, it is not as simple as just making sure that all parts of the implementing code are thread-safe. Timers are entangled with OS message loops, and you cannot simply put them in any kind of background thread expecting they will magically just work. Also, on some platforms, implementing background timers is more complicated than main-thread timers. Because the message loop is an integral part of the timer infrastructure, it is no accident that timers are primarily used in the main thread that runs the GUI message loop.

All of the above is part of why Delphi timer implementations don't even try to support background threads.

Incidentally, Windows is the platform where using timers in background threads requires the least adaptations. Timers on Windows are associated with either a window handle or a thread

with a message loop. In order to trigger specific event handlers, the timer needs to be associated with a window, and the main problem with the original VCL timer's thread safety comes from the thread-unsafe window allocation/de-allocation.

Combining the example with thread-safe window allocation/de-allocation and the example with the background message loop, you can easily adjust the VCL timer code to create a timer that can be used in a background thread. All you need to do is replace calls to `AllocateHWnd` and `DeallocateHWnd` with their thread-safe counterparts in the VCL's `TTimer` implementation. Keep in mind that a background timer needs to be constructed in a background thread, and that you need to avoid the unsafe parts of `TComponent` if you are using it as a base class for a timer.

You can also create a timer directly in a background thread that requires timer functionality. Regardless of how you create your background timer, don't forget to pump Windows messages in its background thread.

```
procedure TWindowThread.WndProc(var Message: TMessage);
begin
  if Message.Msg = WM_TIMER then
    OutputDebugString('Timer message received')
  else
    begin
      OutputDebugString('Message received');
      DefaultHandler(Message);
    end;
end;

procedure TWindowThread.Execute;
var
  M: MSG;
begin
  FWnd := SafeAllocateHWnd(WndProc);
  try
    // create timer
    if SetTimer(FWnd, 1, 500, nil) = 0 then
      raise Exception.Create('Cannot create timer');
    while not Terminated do
      begin
        // do some work
        ...
        // process messages
        while PeekMessage(M, 0, 0, 0, PM_REMOVE) do
          begin
            TranslateMessage(M);
            DispatchMessage(M);
          end;
      end;
  end;
end;
```

```
    end;
  finally
    KillTimer(FWnd, 1);
    SafeDeallocateHWnd(FWnd);
  end;
end;
```

Part 5. Graphics and Image Processing

Chapter 26

Graphics and image processing

Working with graphics and images in Delphi is tightly coupled with visual frameworks, both VCL and FMX. As we already know, those are not thread-safe. Graphics processing can be a longish operation. Doing that in the context of the main thread can have a negative impact on the application's responsiveness. Even though VCL and FMX are not thread-safe, some graphics-handling tasks can still be safely done in background threads.

Handling graphics is a complex process, and both VCL and FMX heavily rely on OS-provided functionality. This means that regardless of the common code, different platforms will have platform-specific code involved, and on some platforms, developers can choose between different rendering technologies, where the availability of those also depends on the users' OSes and devices.

Most of the graphics handling happens in the GUI context, which runs in the main thread by design and that fact somewhat makes thread safety a second-class citizen there. However, even in direct interaction with GUI controls, there are many places where delegating some task to background threads can significantly increase application responsiveness. Combined with a shift in hardware evolution, where more processing power is gained by adding more CPU cores rather than improving the performance of each individual core, moving as much work as possible to background threads becomes a necessity.

Operating systems generally support graphics manipulation in background threads. What kind of support they offer greatly varies: From offering graphics objects that have thread affinity once created and cannot be freely shared, through single-thread-safe graphics objects, where you have to fully manage and protect simultaneous access yourself, and all the way to fully thread-safe instances, where data consistency is protected from within.

Some of the unsafety in the VCL and FMX frameworks comes from being more GUI-oriented and not following all the prerequisites for using particular OS APIs in a thread-safe manner. In other words, the speed of operations in the main thread takes precedence over the ability to run some code in background threads. To some extent, that is a relic of the past, because on

much slower hardware, adding thread safety features would have had a visible negative impact on the GUI.

That thread safety depends on all the code involved in the process—both the Delphi frameworks and the OS APIs. With all possible combinations in play, determining thread safety around graphics, and what kind of code is thread-safe and what kind is not can be a daunting task. And even after you have covered all your bases, reading the Delphi framework code, reading the Delphi documentation (which is usually rather light with regards to thread safety, so you will be mostly left to figure it out on your own), and reading the OS API documentation, there is no guarantee that you will be able to work with graphics in a thread-safe manner. It is not just a question of bugs in your own code, but also bugs—and most of all, lack of multithreading support—in Delphi frameworks, and even the OS itself.

As the older and more mature framework, VCL is more *background-thread-friendly* than FMX. However, it still fails flat on some common operations that you would want to perform in background threads. On the other hand, one of VCL’s advantages, as well as most of its thread safety, stems from being constrained to a single platform—Windows. Because the core VCL graphics classes have a deep connection to the Windows API, making workarounds for the thread-unsafe parts is relatively easy, as you can directly use that API when you need without needing to reinvent graphics support from scratch. Also, any code that directly interacts with OS APIs can be smoothly integrated back into the regular VCL code workflow.

You can use a similar approach with FMX and use the graphics APIs directly, which will give you more control and help you avoid some of the thread-unsafe parts there, but will also require more coding to achieve your goals, and you might end up making your own abstraction layer to fulfill your specific needs. Of course, more platforms also means more code and work to make such workarounds.

Because VCL and FMX are primarily GUI frameworks, each of them has its own class hierarchy for working with graphics. This design also comes from the age difference and origin. VCL is tightly coupled with the Windows platform, and modifying its core graphics support in a cross-platform manner to provide a common base for both frameworks would break VCL’s backwards compatibility, and it would also require significant changes to FMX, which had also already been brought in as a working framework.

With time, some of the common graphics-related types and code have been extracted and moved to the shared **System** namespace, but they are either very simple data containers, or just basic abstractions where the actual specific implementations are in the descendant classes belonging to a particular framework. All those common base types are single-thread-safe, and they don’t have any particular thread-safety features built in.

Besides the Delphi-provided graphical frameworks, there are a number of 3rd party libraries you can use. I will not cover the thread safety of any of those, but at some point, they all need to interact with the OS APIs, either directly or through VCL or FMX, and therefore follow their thread safety rules.

If you start with a shortcut, you usually end up taking the scenic route.

Generally, multithreading does not like shortcuts and quick fixes. Occasionally, you will be able to make some quick fix that will hold for a long time and will not make any troubles down the road. But more often, making quick and dirty fixes will cost you more than doing things properly.

If there is one area where the above is bound to happen, then it is dealing with graphics. If your need for multithreading goes beyond the very limited offerings in the VCL and especially FMX, forget about taking shortcuts. You will still end up taking the scenic route, but at least it will be a thoroughly planned one, where you will not easily find yourself lying in a ditch or in front of a insurmountable obstacle you are not equipped to deal with.

26.1 Graphics platforms and frameworks

Because the thread safety of graphics classes is intertwined with the underlying graphics platforms, knowing how to use a particular API in a thread-safe manner is of the utmost importance. This knowledge also can help you fix the thread-unsafe parts of Delphi implementations or make your own more versatile, thread-safe counterparts.

The following listing provides the starting points for the documentation of all the major graphics platforms that are supported by the Delphi core frameworks and for which there is at least a partial API directly translated and accessible through Delphi units out of the box.

- Windows GDI
 - <https://docs.microsoft.com/en-us/windows/win32/gdi/windows-gdi/>
- Windows GDI+
 - <https://docs.microsoft.com/en-us/windows/win32/gdiplus/>
- Windows Imaging Component
 - <https://docs.microsoft.com/en-us/windows/win32/wic/>
- Windows DirectX
 - <https://docs.microsoft.com/en-us/windows/win32/directx/>
- OpenGL - Windows, Linux, macOS,
 - <https://docs.microsoft.com/en-us/windows/win32/opengl/>
 - <https://www.khronos.org/api/opengl>
- OpenGL ES - Android, iOS (deprecated since iOS 12)
 - <https://www.khronos.org/api/opengles>
- Vulkan - Android, macOS, iOS
 - <https://www.vulkan.org/>

- Metal - macOS, iOS
 - <https://developer.apple.com/metal/>
- macOS AppKit
 - <https://developer.apple.com/documentation/appkit>
- iOS UIKit
 - <https://developer.apple.com/documentation/uikit>
- Android Graphics API
 - <https://developer.android.com/guide/topics/graphics>

Those APIs can be found in various API namespaces: **Winapi**, **Androidapi**, **iOSapi**, and **Macapi**. If you want to directly use those APIs, without involving VCL or FMX classes, you can still use the code from both frameworks as learning examples of how to call and use the APIs in Delphi code. Of course, you will have to avoid the thread-unsafe parts with the help of the original documentation.

For other graphics platforms, you will need to either write your own API translation units, or use 3rd-party ones.

Chapter 27

Resource consumption

Before we start investigating the thread safety of graphics operations, there is another related issue worth exploring: consumption of resources, particularly memory.

Graphics and image processing are resource-intensive operations, and performing such processing in background threads can easily gobble up memory and other resources. While the limits can very easily be hit by 32-bit and mobile applications of any kind, even 64-bit desktop programs need to pay attention to this problem.

Of course, processing images is not the only resource-intensive task that can cause problems. But it is much easier to see how keeping only a few photos in memory is a real problem you need to think about, versus loading and processing 500MB's worth of XML files.

There is another, often forgotten, aspect: Image formats use some kind of compression to minimize storage requirements. However, in order to work with image data, even when we just need to present it in the GUI without any modifications, we must decode and unpack that optimized data into a bitmap representation, where it is typical for every pixel to be represented by 3 or 4 bytes.

For instance, some JPEG-encoded photo with a size of 1920×1080 pixels may have a size of only 200KB, but when unpacked in memory, it will consume about 8MB. And nowadays that is a very small photo. Mobile phones commonly have 8MP cameras that take pictures in a resolution of 3840×2160 pixels which consume about 32MB of memory.

Now, you may think that even 32MB is not that huge, but 32-bit applications on Windows have only 2GB of available memory (which can be up to 4GB if it is large-address-aware and running on a 64-bit OS). Android applications are even more constrained, depending on the device, where 512MB is considered large, and applications will more usually have only 256MB (or even less) memory available.

Another often-forgotten fact is memory fragmentation. Just because the system reports that there is some amount of free memory available for your application, that doesn't mean there is a single consecutive memory block of the desired size available for allocation.

And finally, even when you have enough available memory to load the image, additional, temporary memory may be required during processing to perform the desired operation, and consuming two or three times more is not unusual.

One rather common image operation is creating a small image preview—a thumbnail—while you are presenting images to the user in a list. That requires loading the image in memory and downscaling it to the desired preview size. There are also algorithms that downsample the image while loading so that the image is never unpacked in its full size in memory, but I will ignore that approach for this demonstration. Depending on the image size you are loading, following code will trigger an “Out of memory” exception at some point. Iteration goes to 100 to more consistently show the problem on the 32-bit Windows platform, but you can easily experience memory exhaustion with a smaller number of iterations.

Please note that the following example is not thread-safe, as its purpose is focusing on resource exhaustion. Thread unsafety will not cause any issues in that regard—the only thing that can go wrong is that some of the generated thumbnails may be blank.

```
var
  i: Integer;
begin
  Memo.Lines.Add('Running');
  for i := 0 to 100 do
    TThread.CreateAnonymousThread(
      procedure
        var
          Pic: TPicture;
          Bmp: TBitmap;
        begin
          try
            Bmp := nil;
            Pic := TPicture.Create;
            try
              Bmp := TBitmap.Create;
              Bmp.Width := 150;
              Bmp.Height := 100;
              // path to some image
              // for demonstration we can load the same image
              Pic.LoadFromFile('C:\...');
              Bmp.Canvas.StretchDraw(Rect(0, 0, Bmp.Width, Bmp.Height), Pic.Graphic);
            finally
              Pic.Free;
              // intentional leak of thumbnail bitmap
              // to simulate adding bitmap to some image list
              // Bmp.Free;
            end;
          end;
```

```
except
  on E: Exception do
    TThread.Queue(nil,
      procedure
      begin
        Memo.Lines.Add(E.Message);
      end);
    end;
  end).Start;
end;
```

The core of the resource exhaustion in the above example comes from simultaneously starting a large number of threads that perform some resource- (specifically, memory-) intensive operation. Running that many threads even for much lighter operations is not the best practice, and we should avoid uncontrolled spawning of new threads.

Limiting the number of threads will help avoid the problem, although you need to be aware that running even a few such threads can cause problems due to hardware limitations—particularly on mobile devices.

In the *Parallel collection processing* chapter we limited the number of threads based on the number of CPU cores, but that alone might not be enough for resource-intensive operations. We can also include available memory into the calculation, and implement a failsafe mechanism that will retry the operation a few times if it fails. In such a scenario, it is prudent to implement a delay before retrying the operation, because the system could still be under stress if you retry immediately.

The easiest way to limit the number of running threads is by using tasks from the PPL or some other multithreading library that utilizes thread pools.

Replacing anonymous threads with tasks will solve the problem in most scenarios:

```
var
  i: Integer;
begin
  for i := 0 to 100 do
    TTask.Run(
      procedure
        // process images
      end);
end;
```

Another way of solving issues caused by excessive resource usage is running a single background thread, and moving the loop to the thread. However this approach can significantly increase the time needed to process all items.

```
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      i: Integer;
      Pic: TPicture;
      Bmp: TBitmap;
    begin
      for i := 0 to 100 do
        ...
    end).Start;
end;
```

If using tasks running on threads from the default thread pool still causes issues because there are too many threads running at the same time, using a dedicated, more restricted thread pool is also an option.

You can customize the number of minimum and maximum worker threads in a pool, and the number of threads running in that pool will not exceed the specified maximum. That way, you can easily utilize more than one thread, while still limiting the number of threads running in parallel, and thus preventing resource exhaustion.

Because every thread pool can have its own settings, limiting the number of threads in one pool still allows other pools or individual threads to run and consume additional resources. This is something you also need to take into consideration if you have other tasks running at the same time as the resource-intensive tasks you want to limit.

When setting the minimum and maximum number of threads in a pool, you need to keep in mind that your maximum number must be greater than zero, and either greater than or equal to your minimum number.

When you run a task, the thread pool will try to use any idle or retired threads it has available. If there are no such threads, it will check whether it is allowed to construct new ones. If the number of active threads is already at the maximum, then no new threads will be constructed, and the task will have to wait until some other thread finishes processing its current task. If the number of active threads is less than the minimum, the pool will immediately construct more threads up to the minimum, to have them ready to go for the next tasks.

If you are using custom pools, you should construct the pool in a broader scope and reuse it. The pool itself is a rather lightweight instance, and idle threads get retired after a while. If you constructed the pool locally, you would also have to wait for all tasks to finish, which would make the application unresponsive. In the following example, the thread pool is part of the main form instance, and will live as long as the application is alive.

You can follow a similar pattern with secondary forms, but frequent form construction/destruction is not an optimal pattern. A better approach would be to extract the pool into a separate unit and share it for tasks that have similar requirements. If needed, you can

maintain more than one thread pool, but make sure that you don't make too many of them, as each pool will handle its own set of threads, and you can easily defeat the purpose of having thread pool(s) in the first place. In other words, if you ask yourself whether you need to maintain a separate pool or not, and you don't have a clear answer to that question, then using a separate pool is probably not justified.

```
TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
private
    FPool: TThreadPool;
public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
end;

constructor TMainForm.Create(AOwner: TComponent);
begin
    inherited;
    FPool := TThreadPool.Create;
    FPool.SetMinWorkerThreads(2);
    FPool.SetMaxWorkerThreads(4);
end;

destructor TMainForm.Destroy;
begin
    FPool.Free;
    inherited;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
var
    i: Integer;
begin
    for i := 0 to 100 do
        TTask.Run(
            procedure
                // process images
            end, FPool);
end;
```


Chapter 28

Common graphics types and API

The common graphics types shared by VCL and FMX are scarce and simple. Some were abstracted out of visual frameworks in more recent Delphi versions.

Single-thread-safe

- graphics-related types from `System.Types` for handling sizes, points and rectangles
 - `TSize`, `TSizeF`
 - `TSmallPoint`, `TPoint`, `TPointF`
 - `TRect`, `TRectF`
- all types from `System.UITypes` and `System.ImageList`

Thread-safe

- graphics-related functions from `System.Types` for handling sizes, points and rectangles
- standalone functions from `System.UIConsts`

Note: Functions and methods that involve some floating-point operations may suffer from the FPCR thread safety issue. See: The *Floating-point control register* chapter.

Besides those simple types, one would expect some other framework-independent data containers—for instance: image collections, fonts, brushes—but each framework has its own. This is due to the original VCL implementation being tightly coupled with the underlying OS API. In order to preserve compatibility with existing code, those classes had to remain as they are, so FMX has its own classes.

Additionally, even FMX is not fully API-agnostic, and the data containers you would expect to be more universal are not. One of the reasons for this behavior is that those data containers, like image collections and subsequently image items, don't serve just as pure data holders, but also implement other functionality, like painting, that is inherently tied to the graphics platform. This dependency goes even deeper, as even reading and writing image data involves API calls, as graphics formats trigger a complex decoding/encoding process that runs through the API layer.

Generally, simple data types like points, rectangles, colors, as well as the accompanying handling routines, will be single-thread-safe. Adding thread safety at the type level (except for immutability) for such basic building blocks is useless, as it wouldn't accomplish anything for the thread safety of code in the broader context where those types are used. Having a thread-safe pixel does not make a thread-safe image, and if you have a thread-safe image, you don't need to have thread-safe pixels.

There is also the question of what kind of protection mechanism is suitable. Immutability works well in some scenarios, but has limits, as it prevents fast in-place modifications and slows down the processing. Again, even if you can protect the integrity of some small piece of data, you will accomplish nothing or very little in terms of protecting the whole dataset.

Such simple data types are common in other kinds of frameworks, not just ones that handle graphics. They will commonly be single-thread-safe, and simple handling routines for such types will commonly be thread-safe. It is always possible that some handling routines access unprotected global state, and such code can be easily recognized. It can be more complicated to identify routines affected by some more hidden issues like FPCR—but if you are concerned about those, you should deal with the issue at the source, and fix the cause instead of trying to hunt down all the places where the issue can surface, as this will be almost impossible.

When it comes to the thread safety of more complex data types, especially those which have some thread safety features, as well as those which interact with the OS and other API layers, assessing thread safety and correct usage is much more complicated, and requires a deeper analysis of the code and appropriate knowledge of the APIs used.

Without knowing the graphics platform and how it can be used in a thread-safe manner, you will not be able to successfully write thread-safe code using the platform, even if you are working with an additional framework layer that wraps that platform's API. First, not all of the API will be thread-safe, and you need to know which parts are and which are not. Next, thread safety cannot fully be achieved from within, and some of the burden will always be on the developer using those frameworks.

Chapter 29

VCL graphics types and API

The VCL is built on top of the Windows API, with GDI as an integral part of the APIs used to provide GUI functionality. While you can use other graphics APIs, like GDI+ or DirectX, in a VCL-based application, GDI is and will always be present across all the foundation classes.

The first step in understanding the thread safety of VCL graphics classes requires understanding the thread safety of GDI. In the context of the VCL, we also need to take into account the thread safety of those VCL graphics classes which wrap GDI objects exposed through GDI handles, as well as other VCL graphics types used when handling graphics.

The `Vcl.Graphics` unit holds all of the core and most important classes for handling graphics in the VCL. The rest of its graphics support, including handling the graphical representation of GUI controls, is built upon those base classes, and, of course, the underlying Windows API.

While some of the base graphics classes are simple enough, most of them are rather complex, combining multiple classes where each of those not only implements Delphi-specific behavior, but is also tied with OS API—and their thread safety can depend on all of those elements. However, some of the supporting classes are well-encapsulated inside graphics classes used for consumer code, and they don't have a direct impact on the thread-safe usage of those classes. I will not explicitly cover all those classes, because they are all single-thread-safe and parts that require additional consideration are covered through the main graphics classes, including some specifics of GDI and other Windows APIs.

Some Windows GDI objects are immutable, and some are single-thread-safe. When you work with a single-thread-safe API, you have two choices when building an abstraction layer on top of such an API. You can stay in the domain of single-thread-safe types and let the consumers of those classes add protection around shared data if needed, or you can bake in some thread safety features.

Leaving types as-is, without adding multithreading support, requires writing more repetitive code in consumer code, which also increases the likelihood of errors. Encapsulating some threading support within classes makes using those classes in background threads easier. There are

also different levels of threading support: You can protect the instance as a whole, or protect only some aspects and use cases.

Wrapping immutable types may look simpler than wrapping single-thread-safe types, as immutability implies thread safety. But wrapping an immutable type does not mean that the wrapper will also be immutable, and VCL wrappers are indeed not. In the context of graphics objects, that mutability makes them easier to use, but on the other hand, it complicates threading support.

But this is not all. Expected functionality and some additional considerations can also drive some thread safety features. As we could see in the previous chapters, not all data can be meaningfully shared, and just slapping locks around some code does not mean we have functional, thread-safe code.

When working with graphics in background threads, the most common scenarios don't require simultaneous access to graphics data from multiple threads, and they merely require performing some graphics operations in the background thread. In that context, we will be interested in knowing what the requirements are to perform specific graphic operations in a background thread, and whether handing over data from one thread to another is possible.

The starting point for examining VCL graphics is a few core classes in **Vcl.Graphics**, and their descendants that drive the graphical support in the VCL. Those are:

- **TGraphicsObject** and descendants: **TFont**, **TPen**, **TBrush**
- **TCustomCanvas** and descendants: **TCanvas**, **TBitmapCanvas**, **TMetafileCanvas**
- **TGraphic** and descendants: **TBitmap**, **TIcon**, **TMetafile**, **TWICImage**
- **TPicture** - as a universal graphic container

Note: Even though the VCL is a mature framework, and it doesn't change much, you should keep in mind that any new Delphi version or update can result in code changes affecting thread safety. The primary goal of all VCL graphics support is catering to the GUI, and multithreading is just an afterthought. There is no guarantee that code working in background threads today will still work tomorrow. There is also no guarantee that you will not stumble upon some obscure and well-hidden thread safety issue when using any of the VCL classes.

29.1 Fonts, pens, and brushes

Single-thread-safe - can be safely shared as read-only instances

The descendants of **TGraphicsObject**: **TFont**, **TPen**, and **TBrush** are wrapper classes around particular GDI objects exposed through the Windows API by the handles: **HFONT**, **HPEN**, and **HBRUSH**. Those GDI objects are immutable, and they can be safely shared among multiple threads. However, the wrapper classes are more complicated than that.

If you take a look at the declaration of **TGraphicsObject**, two methods immediately attract attention: **Lock** and **Unlock**, as well as a pointer to a critical section. While our immediate assumption might be that those wrapper classes are also thread-safe, we still need to look at the implementation code to see what is actually being protected—and therefore thread-safe—and what is not.

As we know, behind each **TGraphicsObject** instance is a GDI object, and GDI objects are a limited OS resource—in other words, you can run out of them. To prevent that, VCL implements reusing for those GDI handles.

Separate font, pen and brush object instances that hold the same values will share the same GDI object handle instead of having a separate GDI object constructed. Those handles are managed through the following global resource managers in a thread-safe manner:

```
FontManager: TResourceManager;
PenManager: TResourceManager;
BrushManager: TBrushResourceManager;
```

When you change a property of a graphic object, the appropriate manager will be queried, and a new GDI object will be created only if there is no existing one with the same properties. If the old GDI handle is no longer referenced by any other VCL graphic object instance, the associated GDI object will be deleted.

So caching and reusing existing GDI objects is thread-safe, but that still does not tell us anything about sharing the VCL graphics object wrappers. The locking mechanism at the core of **TGraphicsObject** works only for the objects belonging to a canvas, and its purpose is to prevent modifications to a canvas state (which includes its font, pen and brush) from a different thread while the canvas is locked. In other words, that inner locking mechanism is not meant to be used as a general locking mechanism that allows sharing those objects between threads.

If you want to share and modify any of those graphic objects across multiple threads, you would have to use additional locks. If you want to share them only for reading, then it is safe to do so without any additional locks, as reading any of the properties will not trigger any unsafe modification of the object instance.

29.2 Canvas

Single-thread-safe - has thread affinity

Canvas classes, as the name implies, are used as drawing surfaces that support various drawing, painting and rendering operations.

TCustomCanvas is the base canvas class used throughout the VCL, which only implements common canvas locking and unlocking logic, while the rest of the methods used for drawing and other rendering operations are abstract and need to be implemented in descendant classes.

Each of those descendants can have different thread safety requirements, depending on the actual rendering technology used by a particular canvas class.

The most commonly used canvas class across VCL is **TCanvas**, which wraps a GDI device context through its **HDC** handle. The **TCanvas** instance also holds its own font, pen, and brush instances, which are used for performing graphics operations on that canvas. When you need to use different settings for any of those—for instance, the pen—you will not pass a different pen to a drawing function, but change the properties of the canvas-owned pen.

If you use the device context API directly, you will have to associate the needed (select) brush, pen, or other graphics objects with the device context, where only one of a particular object type can be selected at any time. This is what **TCanvas** does behind the scenes: It selects the appropriate pen or other GDI handle into its device context.

Because device contexts are significantly more complex objects, they cannot be reused between threads the way fonts, pens and brushes can. To optimize usage of device contexts, the VCL implements a garbage collection mechanism for device contexts that are not currently used. That automatic cleanup runs in the context of the main thread as part of windowed controls message processing, and it uses a locking mechanism to prevent cleaning up device contexts that are actively used by other threads.

This means that any time you directly or indirectly access the canvas in a background thread, you need to lock it by calling its **Lock** method, and unlock it with **Unlock** when you are no longer using the canvas. Regardless of whether you are just reading from or writing to the canvas, if you don't lock the canvas, its device context can be deleted by the canvas garbage collection mechanism, clearing its contents.

If you are performing multiple operations on the canvas, you need to keep it locked for the whole sequence. The moment you call **Unlock**, the canvas may be cleared and you may lose all your previous work.

Thread-unsafe canvas locking

```
procedure BackgroundThreadPaint(Bmp: TBitmap);
begin
  Bmp.Canvas.Lock;
  try
    Bmp.Canvas.Brush.Color := clRed;
    Bmp.Canvas.FillRect(Rect(0, 0, 20, 20));
  finally
    Bmp.Canvas.Unlock;
  end;
  Bmp.Canvas.Lock;
  try
    Bmp.Canvas.Brush.Color := clBlue;
    Bmp.Canvas.FillRect(Rect(20, 20, 40, 40));
  finally
    Bmp.Canvas.Unlock;
  end;
end;
```

```
    finally
      Bmp.Canvas.Unlock;
    end;
end;
```

Thread-safe canvas locking

```
procedure BackgroundThreadPaint(Bmp: TBitmap);
begin
  Bmp.Canvas.Lock;
  try
    Bmp.Canvas.Brush.Color := clRed;
    Bmp.Canvas.FillRect(Rect(0, 0, 20, 20));
    Bmp.Canvas.Brush.Color := clBlue;
    Bmp.Canvas.FillRect(Rect(20, 20, 40, 40));
  finally
    Bmp.Canvas.Unlock;
  end;
end;
```

If you construct a device context by directly calling the Windows API, then such a context is not subject to the VCL's automatic garbage collection, as long as you use it only directly through the Windows API. If you assign its handle to any VCL canvas instance, it requires locking.

There is an additional constraint around device contexts, regardless of whether they are tied to a VCL canvas or not. They have thread affinity—meaning that the thread where the device context is created owns that context, and if the thread is destroyed, the device context will be destroyed, too. You can use the context in a different thread from the owning one, but only as long as the owning thread is alive.

29.3 Graphics

TGraphic, as a base ancestor class for all graphics containers—bitmaps, icons, metafiles, and similar—is single-thread-safe, but the thread safety of its descendants depends on their specific implementations.

When working with graphics objects in background threads you need to keep in mind that some operations will use the canvas behind the scenes. In order to work with the canvas, you need to lock it to prevent the canvas garbage collection mechanism from invalidating its contents while you are using it. Not all graphics classes will give you easy access to that canvas, and some may require patching in order to work in background threads. Whether or not some graphics class will require patching to access the canvas also depends on the used Delphi version.

Some graphics classes like `TBitmap`, `TIcon`, and `TMetafile` use an additional container based on the `TSharedImage` class—`TBitmapImage`, `TIconImage`, `TMetafileImage`—to speed up copying data by implementing a copy-on-write mechanism. In other words, when you think you have created a copy of a graphic by using the `Assign` method, you will not have a full deep copy of all the data, and triggering modification of a graphic where its shared data belongs to different threads is not thread-safe.

`TBitmap` implements locking around working with the owned `TBitmapImage`, but not all of the required code is thread-safe, and handling one `TBitmap` image can trigger operations on the shared `TBitmapImage` instance in a thread-unsafe way, causing problems. While you can always try to avoid the thread-unsafe parts of the code, the best way to deal with such scenarios is to call `FreeImage` on a bitmap, which will ensure that you are working with an unshared image instance.

Depending on the operations used when working with both of the bitmaps below, as well as their lifetime, the following code pattern can be thread-unsafe, even if both `TBitmap` instances are accessed from a single thread at a time:

Thread-unsafe

```
var Bmp1, Bmp2: TBitmap;
...
Bmp2.Assign(Bmp1);
TThread.CreateAnonymousThread(
  procedure
  begin
    // work only with Bmp2 here
  end).Start;
// work only with Bmp1 here
```

Thread-safe

```
var Bmp1, Bmp2: TBitmap;
...
Bmp2.Assign(Bmp1);
// force creation of a separate 'TBitmapImage' instance inside Bmp2
Bmp2.FreeImage;
TThread.CreateAnonymousThread(
  procedure
  begin
    // work only with Bmp2 here
  end).Start;
// work only with Bmp1 here
```

Icons and metafiles don't have any accessible mechanism to make a copy of the shared image. In any case, using the **Assign** method when working with any kind of graphics type is not the best approach. If you need to share the same image in read-only mode, then you can just pass the GDI handle around, or you will have to use additional locks to protect their state. If you need to create a modifiable copy, then the best option is to reload the graphics from the original resource, or if that is not an option, use the Windows API methods **CopyIcon** or **CopyEnhMetafile**.

Single-thread-safe

- **Vcl.Graphics.TBitmap**
- **Vcl.Imaging.jpeg.TJPEGImage**
- **Vcl.Imaging.pngimage.TPngImage**

Single-thread-safe - additional restrictions apply

- **Vcl.Graphics.TIcon**
- **Vcl.Graphics.TMetafile**

Thread-unsafe - main thread only

- **Vcl.Graphics.TWICImage**
- **Vcl.Imaging.GIFImg.TGIFImage**

29.3.1 TBitmap

The most commonly used graphics class for manipulating graphics is **TBitmap**. While bitmaps are not commonly used as a storage format, other raster-based formats that are not natively supported as GDI handles will have an associated bitmap image, used for working with their *unpacked* data.

TBitmap encapsulates two GDI objects: a bitmap (**HBITMAP**), and its associated palette (**HPALETTE**). It also owns a canvas that can be used for drawing on the bitmap image. Locking the bitmap canvas is essential when working with the bitmap in a background thread, to avoid garbage collection of the associated device context.

Because a bitmap is often hidden behind handling various other graphics formats, garbage collection of bitmap device contexts is the most common source of bugs when handling VCL graphics in general. In other words, don't forget to lock all the canvas instances involved when working in background threads.

Any operation that involves using a temporary canvas that is not locked during that operation cannot be performed in the context of a background thread.

29.3.2 TIcon

TIcon is primarily designed to work with the main thread. However, under some circumstances, you can use icon instances in the background thread. You will have to avoid calling the **Assign** method on icon instances that will be used in background threads, regardless of whether the icon is the source or the destination of that operation, as it will create a thread-unsafe shallow copy of its data.

Also, using an icon instance that contains PNG-based data is not thread-safe, as the code path around retrieving the icon handle is not thread-safe. If you retrieve the icon handle in the main thread context, and the used background thread can avoid recreating the handle, you can use such an icon instance in the background thread.

29.3.3 TMetafile

Similarly to the **TIcon** class, **TMetafile** is also primarily designed to work with the main thread. Again, one source of unsafety is using the **Assign** method.

Another source is **TMetafileCanvas**, which is also subject to garbage collection. When you create a **TMetafileCanvas** in a background thread, make sure that you lock the canvas while you are using it. You also need to make sure that the referenced device context passed as a parameter does not belong to another thread—or if it does, that it will still be valid while you are using it.

29.3.4 TWICImage

TWICImage is designed to be used in the context of the main thread. There are multiple issues that prevent the safe usage of this class in background threads, from the thread affinity of the singleton WIC image factory, to various thread-unsafe chunks of drawing code.

It is possible to use the Windows Imaging Component API in background threads, but that requires either extensive patching of the VCL's **TWICImage** support, or direct usage of the API. Keep in mind that the WIC is COM-based, and that using the WIC API in a background thread also requires COM initialization in that thread.

29.3.5 TJPEGImage

TJPEGImage is single-thread-safe. It also contains data wrapped in a thread-unsafe **TSharedImage** descendant, so using **Assign** is not an option if you want to use a JPEG instance in a background thread.

Another important thing to keep in mind is that drawing a JPEG will use its internal bitmap and its canvas, exposed through the **Canvas** property, and that you need to lock that canvas while using the JPEG graphic.

Thread-safe drawing of a JPEG image

```
var
  Jpeg: TJPEGImage;
  Dest: TCanvas;
  ...
  Jpeg.Canvas.Lock;
  try
    Dest.Lock;
    try
      Dest.Draw(0, 0, Jpeg);
    finally
      Dest.Unlock;
    end;
  finally
    Jpeg.Canvas.Unlock;
  end;
```

29.3.6 TPngImage

The **TPngImage** class is single-thread-safe. It can safely be used from background threads, although there are a few exceptions—**SaveToClipboardFormat** and **LoadFromClipboardFormat** can only be used from the context of the main thread.

The **Assign** method can be used in a background thread if the source is another **TPngImage**, as it will create a deep copy of the data without using any code that needs to run in the context of the main thread. Assigning to and from any other graphic type can only be done in the main thread.

TPngImage also has a **Canvas** property that needs to be locked in a background thread if any performed operation uses it—for instance, the **Resize** method. The draw method of **TPngImage** handles bitmap data directly through the Windows API without using a VCL canvas, so you don't have to lock the PNG image canvas for drawing operations.

Again, VCL graphics types are generally designed for being used in the main thread, and the thread safety of particular methods can change from version to version.

29.3.7 TGIFImage

TGIFImage is designed to be used in the context of the main thread due to extensive usage of internal bitmaps and, if nothing else, their canvases are subject to VCL device context garbage collection.

29.4 Picture

Single-thread-safe - additional constraints apply

The `TPicture` class serves as a universal holder for any graphics type and while on its own it is a single-thread-safe class, the owned `TGraphic` field imposes its own thread safety constraints, that depend on the actual graphics class stored in that field.

Registering and un-registering graphics formats is not thread-safe, and can only be done from the context of the main thread, and only while no other active thread is working with pictures, as they might access the thread-unsafe graphics and clipboard format registry.

Thread-unsafe - main thread only

- `RegisterPixelFormat`
- `RegisterClipboardFormat`
- `UnregisterGraphicClass`
- `RegisterPixelFormatRes`

29.5 Vcl.GraphUtil

When using graphics utility functions, you need to make sure that such functions do not use any of the thread-unsafe code within. Most of those functions are thread-safe, provided that the passed parameters are thread-safe. That includes mandatory locking of used canvases. While you can lock a canvas that is passed as a parameter, or lock the canvas of a passed bitmap, beware of temporary bitmaps and their canvas usage. You will not be able to lock such canvases, and those functions cannot be used in a background thread—`FillRectAlpha` and `SplitTransparentBitmap` are examples of such thread-unsafe functions.

Such code can be easily patched by locking the offending canvas. You can do that either by directly changing the code in `Vcl.GraphUtil`, or by making an additional thread-safe function.

29.6 VCL image collections

Thread-unsafe - main thread only

- `Vcl.BaseImageCollection`, `Vcl.ImageCollection`
- `Vcl.VirtualImageList`
- `Vcl.ImgList`

VCL image collections can only be used in the context of the main thread. That means that loading data modules containing any of the VCL image collection classes cannot be done in background threads.

Chapter 30

VCL graphics example

One of the most common tasks when working with images is creating thumbnails for an image preview in the GUI. Even loading a single image can take time, and loading many of them takes even more. Using background threads will keep the application responsive while it is generating thumbnails.

The basic code workflow and logic from the following examples can be easily extended and applied to other kinds of graphics and image processing.

There are many ways to generate thumbnails. The most effective ones are loading an embedded thumbnail image from the image formats and files that support them. The Delphi VCL graphics classes don't support reading embedded thumbnails out of the box, so if you want to use that kind of approach, you will have to look for some 3rd-party library.

Another approach is loading subsampled data, and then resizing such an already-smaller image to the desired thumbnail size. This is supported by the `TJPEGImage` class, where you can define in which size you want to decode data: full size, half size, quarter size, or one-eighth size. This can significantly reduce the time needed to render thumbnail images. If you are dealing with images that don't support subsampling, you will have to load the whole image and scale it to the desired size.

The following code example shows how to safely create thumbnails from JPEG images in the specified folder, and add those images along with their filenames to a list view. Specifying `Jpeg.Scale := jsEighth;` will use subsampling to speed up JPEG data decoding and create thumbnails faster.

The drawing algorithm is simplified and adjusted to images in 3:2 format, and there is no dynamic calculation and preservation of the image's aspect ratio. Also, the drawing algorithm is a crude `StretchDraw` without smooth resizing. There is also no exception handling (besides taking care of cleanup in the event of exceptions), and if you try to load an invalid image or encounter other issues, further processing will fail. In real life, you would want to put appropriate exception handling around loading each image, and notify the user if something goes wrong.

In this example, `TThread.Synchronize` is used to synchronize GUI interaction. We cannot use `TThread.Queue` for several reasons: The variable capture mechanism would require us to encapsulate that code in a separate method because we would be capturing incorrect index and bitmap values, and it can access bitmap and string array instances after they have been destroyed.

Thread-safe JPEG thumbnail generation

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  Folder: string;
begin
  Folder := 'C:\...';
  TThread.CreateAnonymousThread(
    procedure
    var
      Files: TArray<string>;
      Jpeg: TJpegImage;
      Bmp: TBitmap;
      i: Integer;
    begin
      Files := TDirectory.GetFiles(Folder, '*.jpg');
      Bmp := nil;
      Jpeg := TJPEGImage.Create;
      try
        Jpeg.Scale := jsEighth;
        Bmp := TBitmap.Create;
        Bmp.Width := 150;
        Bmp.Height := 100;
        for i := 0 to High(Files) do
          begin
            Jpeg.LoadFromFile(Files[i]);
            Jpeg.Canvas.Lock;
            try
              Bmp.Canvas.Lock;
              try
                Bmp.Canvas.StretchDraw(Rect(0, 0, 150, 100), Jpeg);
              finally
                Bmp.Canvas.Unlock;
              end;
            finally
              Jpeg.Canvas.Unlock;
            end;
      end;
    end);
```

```
    TThread.Synchronize(nil,
      procedure
        var
          Item: TListItem;
          Index: Integer;
        begin
          Index := ThnList.Add(Bmp, nil);
          Item := ListView1.Items.Add;
          Item.Caption := Files[i];
          Item.ImageIndex := Index;
        end);
      end;
    finally
      Jpeg.Free;
      Bmp.Free;
    end;
  end).Start;
end;
```

There are a few issues with the previous example. It only handles JPEG images, and the generated thumbnails are not as smooth as they could be.

We could use **TPicture** instead of **TJPEGImage** to load all registered graphics types. However, doing that prevents us from using custom scale to speed up the process for JPEG images and it also does not solve rough drawing. There is another, more important issue: By allowing loading all registered graphics types, we lose thread safety, as some graphics types cannot be safely used in background threads, and we cannot properly lock all the canvases used for other types while generating the thumbnail. We could partially solve this issue by querying the file type and handling specific types separately instead of using **TPicture**, but that still leaves us with an inability to load some file formats in a thread-safe manner.

There is a rather easy solution for those issues. It requires using the GDI+ API for loading images, instead of using VCL graphics classes.

There are a few differences in the workflow compared to the previous example. JPEG images are opaque, so drawing over an existing bitmap would fully erase its previous contents. This example handles all kinds of image types, and some of those support transparent images. To prevent the previous image from showing through, we need to clear the contents of the canvas. This can also be solved by recreating the bitmap on each iteration.

Thread-safe image thumbnail generation

```
uses
  ...
  Winapi.GDIPOBJ;

procedure TMainForm.ButtonClick(Sender: TObject);
var
  Folder: string;
begin
  Folder := 'C:\...';
  TThread.CreateAnonymousThread(
    procedure
    var
      Files: TArray<string>;
      Bmp: TBitmap;
      i: Integer;
      Image: TGpBitmap;
      Dest: TGPGraphics;
      ImgStream: IStream;
    begin
      Files := TDirectory.GetFiles(Folder, '*.*');
      Bmp := TBitmap.Create;
      try
        Bmp.Width := 150;
        Bmp.Height := 100;
        for i := 0 to High(Files) do
          begin
            ImgStream := TStreamAdapter.Create(TFileStream.Create(
              Files[i], fmOpenRead or fmShareDenyWrite), soOwned);
            Image := nil;
            Dest := nil;
            Bmp.Canvas.Lock;
            try
              Bmp.Canvas.FillRect(Rect(0, 0, 150, 100));
              Image := TGpBitmap.Create(ImgStream);
              Dest := TGPGraphics.Create(Bmp.Canvas.Handle);
              Dest.DrawImage(Image, 0, 0, 150, 100);
            finally
              Dest.Free;
              Image.Free;
            Bmp.Canvas.Unlock;
          end;
    end;
```

```
    TThread.Synchronize(nil,
      procedure
        var
          Item: TListItem;
          Index: Integer;
        begin
          Index := ThnList.Add(Bmp, nil);
          Item := ListView1.Items.Add;
          Item.Caption := Files[i];
          Item.ImageIndex := Index;
        end);
      end;
    finally
      Bmp.Free;
    end;
  end).Start;
end;
```

One of the disadvantages of using `TThread.Synchronize` is that the background thread will be idle while the synchronized code runs. We can slightly modify the code to use the `TThread.Queue` method, which will enable the background thread to continue working without waiting for the main thread. This kind of code can also be easily modified to use the thread-safe messaging system, where we would encapsulate the data: filename and bitmap.

Because the background thread will continue to work on the next image until the main thread processes the queued code, we need to construct a separate thumbnail bitmap for each iteration, and release the bitmap when the thumbnail image is being added to the image list. In some other scenario, such a bitmap could also be handed over to some other data structure that would hold all image information separately from the GUI.

Thread-safe image thumbnail generation

```
procedure TMainForm.AddThumbnail(const FileName: string; Bmp: TBitmap);
begin
  TThread.Queue(nil,
    procedure
      var
        Item: TListItem;
        Index: Integer;
      begin
        try
          Index := ThnList.Add(Bmp, nil);
          Item := ListView1.Items.Add;
          Item.Caption := FileName;
        end);
      end);
end;
```

```
    Item.ImageIndex := Index;
  finally
    Bmp.Free;
  end;
end);
end;

procedure TMainForm.ButtonClick(Sender: TObject);
var
  Folder: string;
begin
  Folder := 'C:\...';
  TThread.CreateAnonymousThread(
    procedure
    var
      Files: TArray<string>;
      Bmp: TBitmap;
      i: Integer;
      Image: TGPBitmap;
      Dest: TGPGraphics;
      ImgStream: IStream;
    begin
      Files := TDirectory.GetFiles(Folder, '*.*');
      for i := 0 to high(Files) do
        begin
          ImgStream := TStreamAdapter.Create(TFileStream.Create(
            Files[i], fmOpenRead or fmShareDenyWrite), soOwned);
          Image := nil;
          Dest := nil;
          Bmp := TBitmap.Create;
          try
            Bmp.Width := 150;
            Bmp.Height := 100;
            Bmp.Canvas.Lock;
            try
              Bmp.Canvas.FillRect(Rect(0, 0, 150, 100));
              Image := TGPBitmap.Create(ImgStream);
              Dest := TGPGraphics.Create(Bmp.Canvas.Handle);
              Dest.DrawImage(Image, 0, 0, 150, 100);
            finally
              Dest.Free;
              Image.Free;
              Bmp.Canvas.Unlock;
            end;
          end;
        end;
    end;
  );
end;
```

```
    AddThumbnail(Files[i], Bmp);
  except
    Bmp.Free;
    raise;
  end;
end;
end).Start;
end;
```


Chapter 31

FMX graphics types and API

Since Delphi 10.2 Tokyo, according to the documentation, FireMonkey has had multithreading support for some classes.

FireMonkey supports multi-threaded use of `TBitmap`, `TCanvas`, and `TContext3D`. Bitmaps can be created, used, and deleted across any thread.

While the first sentence is true, and there are no specifics defined on how you can use those classes, any further discovery around how to use those in thread-safe manner is left to you and reading further documentation and examples. Meanwhile, the second one should have a *Here be Dragons* warning sign attached. Namely, `TBitmap` contains a `TCanvas` field, and the thread safety of `TCanvas` varies greatly: Some can only be constructed in the context of the main thread, and all canvas classes can only be destroyed in the context of the main thread. While constructing a `TBitmap` image alone does not trigger constructing a canvas instance, destroying a `TBitmap` is definitely not thread-safe, because it also destroys the canvas, if it had been created during the bitmap instance's lifecycle. Besides `TCanvas`, there may be other thread-unsafe classes involved, depending on the platform and rendering technology used.

In other words, while you may be able to construct a `TBitmap` instance in thread-safe manner, you will not be able to do a thing with that instance in a background thread. It is a literal minefield, and every second operation you may use can blow up in your face.

Loading an image from a file in a background thread... Kaboom!!!!

When it comes to bitmap handling, there are two base classes, `FMX.Surfaces.TBitmapSurface` and `FMX.Surfaces.TMipmapSurface`, that can be safely used in background threads. Their functionality is limited, but they allow you to manipulate bitmap pixel data in background threads. Just keep in mind that loading from and applying their data to bitmaps is not thread-safe, and that you will need to synchronize those operations with the main thread.

`TContext3D` is also not thread-safe, as some of its fields are thread-unsafe classes, like `TTexture`, that can only be used in the context of the main thread.

The main cause of thread unsafety in FireMonkey is using the thread-unsafe messaging system, **TMessageManager**, and the **TMessageManager.DefaultManager** instance. There are possibly more unsafe parts, but trying to figure out what else is safe or unsafe in such complex code is mostly a waste of time until the main source of unsafety is removed.

Another source of unsafety for some FMX graphics classes comes from implementing the **IFreeNotification** interface, and another from descending from the **TFmxObject** class. As always, if the source of thread unsafety is not triggered while the class is being used in a background thread, then this will not cause issues, but figuring out whether or not some code path will trigger any of those in such a complex framework is another story.

The next source of unsafety comes from various global instances and managers, that hold thread-unsafe collections used for registering and unregistering various data. If that code is triggered only during some initialization and finalization processes at the very beginning and at application shutdown, and they are used in read-only mode for the rest of the application lifetime, then code using those instances will not cause troubles even if it runs in background threads. If those instances modify the collection data anywhere during the application's lifetime, then any code using those collections will not be thread-safe.

When the registering and unregistering is triggered during construction and destruction of particular object instances, and you can restrict their construction and destruction to the main thread, it is still be possible to hand some of those instances over to background threads and perform some other operations that don't trigger any other thread-unsafe features.

For instance, **TShaderManager** is an example of a class that handles collection data in a thread-unsafe manner, and **TFilter** and **TMaterial** instances will modify the **TShaderManager** collection while being constructed.

FireMonkey also extensively uses floating-point operations, and some operations may suffer from the FPCR thread safety issue when running on the Windows platform. See: The *Floating-point control register* chapter.

There may be ways to avoid thread-unsafe code, and some platforms may be more thread-safe than others, but as we know, thread safety is an absolute category, and one that is hard to prove, as you need to inspect every piece of data involved and all possible execution paths.

This thread unsafety also has a negative impact on other 3rd-party libraries that interact with or extend those base thread-unsafe classes.

Because FireMonkey is deeply unsafe, extracting the all bits and pieces that can be used in background threads will not get us very far. All those bits and pieces also need to interact in various ways, and sometimes you will be able to avoid one booby trap only to have the next one blow up in your face. However, if you really need to do that, then you will have to start from the particular functionality you are interested in, and start examining it from the base types up. Probably, the fastest course of action is to use the platform APIs directly, and perform most of the necessary interaction with FireMonkey in the context of the main thread.

Part 6. Custom Frameworks

Chapter 32

Writing custom frameworks

This part is dedicated to custom frameworks which are indispensable in multithreaded programming, and are not provided out of the box. Because they are indispensable, there is a plethora of open source or commercial solutions available, but building some from scratch provides an opportunity to better understand their underlying concepts, as well as some of the problems commonly encountered during their implementation. This also gives us an opportunity to explore some thread safety patterns which were not covered earlier in the book.

Understanding the concepts and implementing the simplest solution can also be immensely helpful if 3rd-party solutions do not fit your use case. That knowledge can help you in tweaking existing solutions to cover your needs, without breaking their thread safety.

The presented frameworks can also be used as basic building blocks from which you can build your own frameworks with extended functionality. Sometimes it can be easier to add the desired functionality on top of simpler code than to modify and adapt a more complex solutions.

Not all of the problems or potential solutions presented will be strictly related to thread safety. But, generally, many threading issues can be solved by approaching the problem from a different angle, and sometimes by avoiding concurrency altogether. From that point of view, any exercise that teaches how to explore a coding problem from a different perspective will be a good one.

Seeing code evolution is not only helpful in seeing the transformation from an unsuitable solution to a better one. What is an unsuitable solution in one scenario may be a perfect fit in another. It is important to keep in mind that the reasoning behind some code is usually complex: You need to cover all bases and all requirements to find the optimal solution. Sometimes there will be no *best option*, and choosing one over the other will be more a matter of personal preference.

Personal preferences can also influence your choice even when a better option exists. It is important to use coding patterns you feel comfortable with. That does not mean you should never strive to learn, improve and evolve. But using code you don't understand well opens up more possibilities for errors and mistakes. Using code on your level of confidence, or just slightly above it, can mean fewer bugs and result in more maintainable code, where you will not be afraid to make changes and add required features.

You can travel further, safer, and eventually faster using a car, than you can travel with an airplane you will crash and burn every few steps. But you will still need to use a rocket ship, regardless of your skill, if you want to reach the Moon.



Chapter 33

Logging

In the context of thread safety, logging can help us with monitoring code flow for debugging purposes. To be clear, debugging and logging will not always help you in finding a problem—many times, they will not even reveal that you do have a problem—but sometimes, some errors in code will be more obvious when you start logging and you see unexpected results. Logging is just another tool in your toolbox to help you examine code behavior, but it should never be used as a way to prove the thread safety of some code.

Anyone familiar with physics knows about the *observer effect*—a disturbance in the observed system caused by the act of observing. In other words, observing the system changes its behavior. If the intrusion is large enough, the change in behavior can be significant enough to ruin the experiment and give us completely unusable data.

A similar effect can be seen in multithreaded code. The moment you try to observe what is going on in some part of such code, you may significantly alter the behavior of that code. To minimize the possible impact on observed code, logging frameworks used for debugging purposes must be as fast as possible, even if it means sacrificing some features.

The simplest logging would just emit the passed string to the debug console (and to the regular console for console applications). However, for debugging multithreaded applications, that is not always enough. Running with IDE debugging enabled and watching output in the Events (debug log) console takes significantly more time than running the application without debugging. Another problem is that for observing code behavior in certain scenarios, an IDE might not be available.

I said that fast logging requires sacrificing some features, and the first picture that might come to your mind is plain and simple, hardcoded logging with one or maybe a few procedures you will call directly. But that is not a very flexible solution. Without sacrificing speed, we can easily implement a simple, plug-and-play solution where you can easily switch implementations.

The basic interface needed is just a single procedure responsible for writing whatever log message we need to the appropriate output channel. We might be tempted to extend that interface with

additional functionality, but that would be wrong, as we can compose the message outside that interface, and the output mechanism is the only thing that can differ between implementations.

Keep in mind that this is a very simple logger, and its primary function is providing a simple, easy-to-use, and fast debug-oriented logger, not a full-fledged logging solution.

```
type
  INxLogger = interface
    procedure Output(const aMsg: string);
  end;
```

The logger interface will be wrapped inside a simple static class, that will provide an initial logger instance which we can easily replace:

```
type
  NxLog = class
  protected
    class var Logger: INxLogger;
    class constructor ClassCreate;
  public
    class procedure SetLogger(const aValue: INxLogger); static;
    class procedure D(const aMsg: string); static;
  end;

  class constructor NxLog.ClassCreate;
begin
{$IFDEF DEBUG}
  Logger := TNxSystemLogger.New;
{$ENDIF}
end;

class procedure NxLog.SetLogger(const aValue: INxLogger);
begin
  Logger := aValue;
end;

class procedure NxLog.D(const aMsg: string);
begin
  if Assigned(Logger) then
    Logger.Output(aMsg);
end;
```

While the default logger is automatically created if the application is compiled in debug mode, if you want to use some other logger class, or you want to enable logging in release mode, you can also set it manually with `SetLogger`. Passing `nil` disables logging.

Since initializing the logger is something that you will do only once in the application's lifetime, the `Logger` instance is not additionally protected by locks, and the `SetLogger` procedure is not thread-safe for performance reasons. If you want to change `Logger`, you will need to make sure that there are no running threads (other than the main thread) using the `NxLog` class and emitting logs at that time. Once the `Logger` instance is initialized, it can be safely used across threads.

The rest of the framework's thread safety depends solely on the specific logger implementation, and whether the specific implementation of the `Output` procedure contains any thread-unsafe code.

For easier usage and avoiding `nil` checks everywhere, you can use the `D`—debug—class procedure instead of accessing the `Logger` instance directly. You may wonder whether this procedure is thread-safe or not. Generally, `if Assigned(...)` `then` is not a thread-safe construct, as the instance we are checking can be released after the `nil` check succeeded, but before it is used. However, in the `NxLog` class, the only thing that can release the instance in the context where `NxLog.D` can be called is `SetLogger`, and if we follow the documented rule about initializing the logger before we spawn any threads, and not modifying it once initialized, then the `NxLog.D` procedure will also be thread-safe.

There is a point during application shutdown where the `Logger` instance, being an automatically managed field, will be cleared. The thread safety of that scenario is covered further down in the *Cleanup on shutdown* chapter.

This `NxLog` class is the simplest possible logging implementation which allows us to add extremely detailed logging in the debug phase when the `DEBUG` compiler directive is turned on, where we don't need to make any code changes for release mode—calling `NxLog.D` still takes a few CPU instructions even when logging is disabled, but that cost is negligible in most code. Depending on the `INxLogger` implementation, it is also rather flexible, and enables easy switching between outputting the log to the console, a file, or any other output provider.

Its greatest disadvantage is that it only has “on” and “off” modes. If you have a lot of logging code meant for debug mode, and would still like to have some logging capabilities in release mode, the initial implementation will not suffice. Of course, you can always switch to a full-fledged logging solution, but with just a few modifications, `NxLog` can become more versatile, and quite usable for lightweight logging.

33.1 Logging levels

Further extension of logging levels, beyond “on” and “off”, can be done by adding a new enum, an associated `Level` field, and expanding the number of logging methods:

```
type
  TNxLogLevel = (LogDebug, LogInfo, LogWarning, LogError, LogFatal, LogOff);

  NxLog = class
  protected
    class var Logger: INxLogger;
    class var Level: TNxLogLevel;
    class constructor ClassCreate;
  public
    class procedure SetLogger(const aValue: INxLogger); static;
    class procedure SetLevel(aValue: TNxLogLevel); static;
    class procedure D(const aMsg: string); static;
    class procedure I(const aMsg: string); static;
    class procedure W(const aMsg: string); static;
    class procedure E(const aMsg: string); static;
    class procedure F(const aMsg: string); static;
  end;

  class constructor NxLog.ClassCreate;
begin
  Level := LogOff;
{$IFDEF DEBUG}
  Logger := TNxSystemLogger.New;
  Level := LogDebug;
{$ENDIF}
end;

class procedure NxLog.SetLogger(const aValue: INxLogger);
begin
  if not Assigned(aValue) then
    Level := LogOff;
  Logger := aValue;
end;

class procedure NxLog.SetLevel(aValue: TNxLogLevel);
begin
  Level := aValue;
end;

class procedure NxLog.D(const aMsg: string);
begin
  if Ord(Level) <= Ord(LogDebug) then
    Logger.Output(aMsg);
end;
```

```

class procedure NxLog.I(const aMsg: string);
begin
  if Ord(Level) <= Ord(LogInfo) then
    Logger.Output(aMsg);
end;
...

```

Now we have more fine-grained logging capabilities, which we can easily modify without sacrificing too much performance.

Because we now have a `Level` field, we can now set the logging level to `LogOff` if we pass a `nil` reference to `SetLogger`, and we only need to check `Level` in our logging methods.

33.2 Cleanup on shutdown

It has been said many times before: In multithreading, cleanup is the hardest thing to do properly. Even without multiple threads involved, it can often be hard to do it right.

We need to make sure that any logging during shutdown does not cause exceptions because the logging framework is being dismantled. In the initial implementation, we checked whether `Logger` is assigned. In the extended implementation, we have removed that safeguard. We could add it back, but let's see whether this is really necessary. Calling `Assigned` is fast, but not calling it is even faster.

We can solve this problem by using a class destructor to turn off logging before the `Logger` instance is released.

```

NxLog = class
protected
  class destructor ClassDestroy;
...
end;

class destructor NxLog.ClassDestroy;
begin
  Level := LogOff;
end;

```

But is it thread safe?

```

if Ord(Level) <= Ord(LogDebug) then
  Logger.Output(aMsg);

```

The above code from the `NxLog.D` method is not thread-safe in the sense that, by the time we pass the `if..then` check, another thread could change the logging level. If `Level` is all that is changed, then the worst thing that could happen is we would have an entry in the log that was written after logging was turned off.

This is not a critical issue—extra log entries can be tolerated, and there is no reason to introduce a performance penalty for that.

But, there is a more critical issue here. After we run the code in the class destructor, the next part of the automatic class cleanup will clean—release—all managed class fields in the `NxLog` class—our `Logger` instance. That means, if we are inside `NxLog.D` or some other logging method, we could access the `Logger` variable after it has been cleared, causing a crash. Merely turning logging off will not help much in such a scenario. Even the initial code that used `Assigned` check could fail under the same circumstances.

To determine the thread safety of `NxLog`'s class cleanup process, we need to determine whether it is possible that some code in another thread is using it during the cleanup. This brings us to the application and unit initialization/finalization sequence, where our carefully crafted logging solution starts falling apart.

As a quick reminder, the initialization/finalization sequence runs in the following order:

Unit initialization order is defined by their order in a unit's uses clause, starting from the main uses clause, and each unit is visited only once—there is no recursion. Finalization is executed in reverse order.

When unit initialization starts, all class constructors are executed first, then the unit initialization block. Descendant class constructors will run after ancestor classes. Finalization runs in reverse order: First the unit finalization block executes, then the class destructors.

When using runtime packages, things get a bit more complicated, because the order of loading is affected by the order in which the modules are loaded, but the same principles apply.

With code containing complex unit dependencies and unit cycles, as well as class references from one class constructor to another, determining the order of initialization is not straight forward—in other words, it may not always be what you would expect it to be.

Back to our `NxLog` class cleanup and thread safety. The core thread class, `TThread`, is declared in `System.Classes`, but that does not mean there will be no running threads after that unit's finalization is completed. If nothing else, there may still be some self-destroying threads running, as the RTL does not perform any cleanup or wait for such threads during application shutdown.

From that perspective, whether the `NxLog` class will be cleaned up after or before `System.Classes` does not make much difference. What does make a difference, is that it is cleaned up after the `System.Threading` unit, which utilizes threads for tasks, and manages the proper shutdown of those threads. If you use logging in your task's code, you don't want to turn logging off before those tasks have finished running. The same applies if you are using any other 3-rd party threading library, or any other library that uses threads from which you might trigger logging calls.

To ensure that `NxLog` is finalized after the previously mentioned or if it is not the first, any unit before it must also not have any of the threading units in its dependency graph.

There is still the open issue of self-destroying threads that might be alive after logging is finalized, where trying to adjust the order of finalization will be impossible and will achieve nothing. But in such cases, you already have bigger problems than the logging itself.

If you cannot meet those requirements, it is possible that some logging code will run after the logger is dismantled, and however slim, there is a chance that such code can pass the `if Ord(Level) ...` check and land on the now non-existent `Logger` instance. Adding an additional assignment check will not help here, because it will be just as unsafe as the previous one.

If you are running the logger only for debugging purposes, and you can tolerate a very slim chance that application will blow up on shutdown, you don't need to pay much attention to the cleanup order.

Technically, if you maintain the proper order of initialization/finalization, you don't even need a class destructor or turning off logging, because there will be no code calling the logger after it is dismantled. However, adding the class destructor does not cost anything, and turning the logger off will help in preventing most issues—there is only an extremely small timeframe where threads can interleave—even if you fail to maintain a proper cleanup order. This is one of the rare places where adding code *just in case* is justified, especially if you are using logging in release mode and you definitely don't want to cause crashes just because your logger is not setup properly.

There is an additional issue we can have during initialization. If the logger is not yet initialized, and any other code that uses logging is triggered before initialization, logging would crash because the `Logger` instance is `nil`. An additional `nil` check would help prevent such a situation. With logging, we probably have more room to adjust unit initialization and finalization order to avoid problems. We might not have such a luxury in some other situations and frameworks.

While using an additional `nil` check would fix the problem, there is another solution in this particular case. Our initialization problem begins because `0`, the initial value of `Level`, is equal to `LogDebug`. That can be easily solved by reversing the order of the enum's values, and changing `<=` to `>=` in our expressions.

So the updated version of the `NxLog` class would be:

```
type
  TNxLogLevel = (LogOff, LogFatal,.LogError, LogWarning, LogInfo, LogDebug);

class procedure NxLog.D(const aMsg: string);
begin
  if Ord(Level) >= Ord(LogDebug) then
    Logger.Output(aMsg);
end;
```

```
class procedure NxLog.I(const aMsg: string);
begin
  if Ord(Level) >= Ord(LogInfo) then
    Logger.Output(aMsg);
end;
...
```

When dealing with initialization and finalization sequences, even without threads involved, you have to keep in mind that the order of execution might not be the one you need, and you need to make sure that non-critical code that does not need to run does not crash. If you are dealing with code that must execute (logging can be safely omitted without causing trouble in any application) and that must execute in a specific order, you will have to change and maintain the order of initialization/finalization.

Having said that, relying on a specific order is rather fragile—with or without threads. You should avoid such code when possible. If necessary, initiate initialization explicitly in the main code block, after all the low-level automatic initialization is completed. The same rules are valid for the shutdown process: The less you depend on automatic processes for code that needs to run in a specific sequence, the better.

In the context of our logging class, and generally concerning background threads, it is better and safer to shut threads down manually—through your code—rather than leaving them for automatic cleanup or, in some cases, no cleanup at all. While it is certainly possible to have a well-behaved application with automatic process, taking care of thread cleanup early in the application shutdown process can avoid a whole range of potential issues.

33.3 Loggers

This chapter covers two basic logger classes: **TNxSystemLogger** and **TNxFileLogger**. For most applications, those two will be more than enough. Since the loggers have extremely simple interfaces and only need to implement a single method, it is easy to create various other logger classes, or modify existing ones, if needed.

The only requirement that needs to be taken care of is that the **Output** method must be thread-safe. Constructing a logger instance does not need to be a thread-safe operation because the whole **NxLog** class requires that initialization and setting the logger instance is completed before any threads start to use it.

33.3.1 System Logger

This most basic logger uses various debugging APIs like **OutputDebugString** and plain console output. Since it relies heavily on OS-specific APIs, different platforms will require different

implementations. Each particular output implementation is very simple on its own and does not need additional explanation about what it does.

```
uses
{$IFDEF MSWINDOWS}
  Winapi.Windows,
{$ENDIF}
{$IFDEF ANDROID}
  Androidapi.JNI.JavaTypes,
  Androidapi.Helpers,
  Androidapi.Log,
{$ENDIF}
{$IFDEF IOS}
  Macapi.Helpers,
  Macapi.ObjectiveC,
  iOSapi.Foundation,
{$ENDIF}
{$IFDEF OSX}
  Macapi.Helpers,
  Macapi.ObjectiveC,
  Macapi.Foundation,
{$ENDIF}
  System.SysUtils,
  System.SyncObjs;

type
  TNxSystemLogger = class(TInterfacedObject, INxLogger)
public
  procedure Output(const aMsg: string);
  class function New: INxLogger;
end;

{$IF defined(MSWINDOWS)}
procedure TNxSystemLogger.Output(const aMsg: string);
begin
  OutputDebugString(PChar(aMsg));
  if IsConsole then
    begin
      TMonitor.Enter(Self);
      try
        Writeln(aMsg);
      finally
        TMonitor.Exit(Self);
      end;
    end;
end;
```

```
    end;
end;
{$ELSEIF defined(ANDROID)}
procedure TNxSystemLogger.Output(const aMsg: string);
begin
  LOGI(PUtf8Char(Utf8String(aMsg)));
end;
{$ELSEIF defined(MACOS)}
procedure TNxSystemLogger.Output(const aMsg: string);
begin
  NSLog(StringToID(aMsg));
end;
{$ELSEIF defined(LINUX)}
procedure TNxSystemLogger.Output(const aMsg: string);
begin
  if IsConsole then
    begin
      TMonitor.Enter(Self);
      try
        Writeln(aMsg);
      finally
        TMonitor.Exit(Self);
      end;
    end;
  end;
{$ELSE}
procedure TNxSystemLogger.Output(const aMsg: string);
begin
end;
{$ENDIF}

class function TNxSystemLogger.New: INxLogger;
begin
  Result := TNxSystemLogger.Create;
end;
```

Let's see whether those `Output` methods are thread-safe.

`OutputDebugString`, `LogI` and `NSLog` are thread-safe. What is not thread-safe is writing to the console with `Writeln`, which needs to run in the context of the main thread using `TThread.Synchronize` or `TThread.Queue`.

In this particular case, `Synchronize` and `Queue` pose a problem, as they don't work in console applications out of the box. They require cooperation from the main thread, from which `CheckSynchronize` needs to be called in order to run any pending code.

To avoid calls to `Synchronize` or `Queue`, we can use a locking mechanism to protect the console output. This is also much faster than synchronization with the main thread, and we want logging to run as fast as possible. However, the locking approach is only thread-safe as long as no other code uses `Write` or `Writeln` directly.

If your console applications are just quick demos to observe some behavior, you can use this partially thread-safe system logger in combination with `Write` or `Writeln` - it is possible you will get some garbled output, but in most cases it will work properly.

33.3.2 File logger

`TStream` and its descendants are not thread-safe classes, and in order to have a thread-safe file logger, we need to protect writing to a file stream. But, depending on the use case, we can exploit the fact that most instances of writing a buffer to a file are atomic and protected on the OS level. As long as we always start with a new file, keep it open during the lifetime of the application, don't use seeking, and always write every log entry in a single write operation, we don't need to put additional protection in place—at least, not if we can live with the occasional *mis-write*. For debugging purposes, this is usually the case.

The atomicity of writing a buffer to a file is also platform-dependent. On Posix platforms, it is guaranteed, but macOS and iOS don't always follow all Posix rules, including this one, so writes there will not be atomic. On Windows, single sector writes are atomic, but if a write crosses a sector boundary, then it is not.

The file logger is much faster than the system logger. If you are interested in observing the code with as little interference as possible, the file logger is the best choice, especially in code with frequent logging calls. On the other hand, the system logger will do just fine for observing long-running code, where logging calls will not have much impact and will not cause a significant slowdown, as well as mobile applications, where setting up a file logger and accessing the written logs is more cumbersome.

```
TNxFileLogger = class(TInterfacedObject, INxLogger)
protected
  f: TFileStream;
public
  constructor Create(const aFileName: string);
  destructor Destroy; override;
  procedure Output(const aMsg: string);
  class function New(const aFileName: string): INxLogger;
end;

constructor TNxFileLogger.Create(const aFileName: string);
begin
  inherited Create;
  f := TFileStream.Create(aFileName, fmCreate or fmShareExclusive);
```

```
end;

destructor TNxFilLogger.Destroy;
begin
  f.Free;
  inherited;
end;

procedure TNxFilLogger.Output(const aMsg: string);
var
  Buf: TBytes;
begin
  Buf := TEncoding.UTF8.GetBytes(aMsg + #13#10);
  f.WriteData(Buf, Length(Buf));
end;

class function TNxFilLogger.New(const aFileName: string): INxLogger;
begin
  Result := TNxFilLogger.Create(aFileName);
end;
```

For a fully thread-safe file logger, you need to protect writing to a file:

```
procedure TNxFilLogger.Output(const aMsg: string);
var
  Buf: TBytes;
begin
  Buf := TEncoding.UTF8.GetBytes(aMsg + #13#10);
  TMonitor.Enter(f);
  try
    f.WriteData(Buf, Length(Buf));
  finally
    TMonitor.Exit(f);
  end;
end;
```

Note: The `Output` method of the file logger can raise an exception. Raising exceptions during logging can have a negative impact on the application. Logging is usually not a critical operation, and you don't want your application to break because your logging function failed. To prevent that, you might want to put a `try...except` block in the `Output` method and silently eat the exception, or you may want to show an error message to notify the user the logging failed, and turn logging off at that point (to prevent a flood of logging failure messages).

However, handling exceptions slows things down, and this may interfere with debugging. That is why the presented file logger does not bother with exception handling. If you want to keep

logging turned on in the application you release to your clients, you will have to use a different logging implementation that fully handles exceptions and prevents spurious issues from arising because of failed logging.

33.4 Logging extensions

In addition to having different unit initialization orders, a logging class may need to log different data in different applications and for different purposes. This is where class helpers can come in handy and allow us to tweak logging behavior and automatically add other information we deem helpful, like timestamps, thread information, or anything else. We can certainly expand functionality and add those into the base `NxLog` class directly in `D` and other methods, but every added line of code has an impact on performance. Since the initial goal was to make a fast logging framework for debugging and observing multithreaded code, additional functionality can be *too heavy* in some situations.

With class helpers, you can, for instance, redeclare the `D` method, and then everywhere the class helper is in scope, it will override the `NxLog.D` method:

```
type
  NxLogHelper = class helper for NxLog
  public
    class procedure D(const aMsg: string);
  end;

  class procedure NxLogHelper.D(const aMsg: string);
begin
  if Ord(Level) >= Ord(LogDebug) then
    Logger.Output('Some other information ' + aMsg);
end;
```

The downside of class helpers is that while they can add functionality and inject different behavior, doing that requires an additional unit, and changing the uses clause of a unit where we want to change a behavior. There is also another problem: We can only have single class helper in scope—though even if we had multiple, they would not be able to neatly solve all extension problems.

Helpers are mostly suitable when you want a different behavior on the application level, but if you have some common library code across applications where logging is used, and you want that output to be different in different applications, then helpers will not be of much help.

One of the possible output additions, quite helpful for debugging threads, would be the ability to log current thread information so you can easily see whether particular code is running on the expected thread—if code runs on a background thread and you expect it to run on the main thread, or vice versa, it will be more obvious where the problem is, and what needs to be fixed and how.

If you want to add the thread name only in some situations, you could declare `NxLog.DT`, and then call that method instead of `NxLog.D`. For such extensions, where you just add a differently named method or overload, you don't even need helper classes, unless implementing the method directly would drag in some other units you might not want included in a general-purpose logging unit.

If adding functionality would add dependencies that would change initialization/finalization order, the easiest solution for such problem, would be to split the functionality of the `NxLog` class, leaving only the interface declaration, logging levels and class variables in the base unit, and moving everything else into an additional unit that will hold the rest of the implementation. With that kind of separation, into the `NX.LogBase` and `NX.Log` units, we can easily guarantee order of finalization for the `NX.LogBase` unit, and that the `Logger` instance will not be destroyed while we are still using it.

Splitting the `NxLog` class complicates the logger a bit, because it now requires two units instead of one. If you don't add any dependencies to the logger that might change the finalization order, you don't have to use two units, and you can have all the required logging supported in one unit.

```
unit NX.LogBase;

interface

type
  INxLogger = interface
    procedure Output(const aMsg: string);
  end;

TNxLogLevel = (LogOff, LogFatal, LogError, LogWarning, LogInfo, LogDebug);

NxLogBase = class
protected
  class var Logger: INxLogger;
  class var Level: TNxLogLevel;
public
  class procedure SetLogger(const aValue: INxLogger); static;
  class procedure SetLevel(aValue: TNxLogLevel); static;
end;
...
```

The `NX.Log` unit is organized in such a manner that you don't need to change the uses clause in any units that use logging, and you only need to put `NX.Log` there, the same way you would if you only had a single logging unit. The `NxLog` class inherits from the `NxLogBase` class, so you can have direct access to the `SetLogger` and `SetLevel` methods, as well as the protected `Logger` and `Level` fields. To avoid adding `NX.LogBase` into the uses list in places where you want to change logging levels, `TNxLogLevel` has been re-declared as an alias:

```
unit NX.Log;

interface

uses
  NX.LogBase,
  ...; // other dependencies

type
  TNxLogLevel = NX.LogBase.TNxLogLevel;

  NxLog = class(NxLogBase)
  ...
  
```

We can put the class destructor in either of those two units. If we put it in the `NX.Log` unit and turn logging off there, we will get additional time between thread shutdown and releasing the logger. Even if our cleanup goes south, it will be almost impossible for some thread to pass the `if Ord(Level)` check and land on an already-destroyed `Logger` instance.

However, we might be interested in debugging the thread shutdown process even if we are doing that manually, just to be sure that it goes smoothly and as expected. If that is so, then turning logging off in `NX.Log` may be too soon, and turning it off in `NX.LogBase` is better.

In any case, you can always turn off logging at any point in your application lifecycle. Just remember that merely turning it off does not mean that you will not get some extra log entries if there are threads actively logging at that moment.

Now, we can start adding other functionality to the `NxLog` class or its helpers.

33.4.1 Thread identification

Knowing the thread context of some running code can be extremely important for debugging. The most important information is whether some code is running on the main or a background thread. The next most helpful thing is identifying threads, so that we can distinguish between different background threads.

It would be great if we could give threads more descriptive names. We could use the `TThread.NameThreadForDebugging` method, but it is somewhat usable only if there is a debugger attached—somewhat is an overstatement, because retrieving information from such a named thread requires throwing an exception, and this is clearly a bad choice for logging purposes.

Some platforms, like Android, support named threads, and we can use the available OS APIs to provide named threads for our logging purposes. Another option would be to keep a thread-safe dictionary of thread IDs and their associated names in the `NxLog` class and use that for thread identification—making sure that each named thread is also removed from the dictionary upon

destruction. However, for most cases, identifying the main thread and different background threads by thread ID will suffice. This is also the fastest option.

We can add support for **ThreadName** in the following manner:

```
NxLog = class
  ...
  protected
    class function GetThreadName: string; static;
    class procedure SetThreadName(const aValue: string); static;
  public
    class property ThreadName: string read GetThreadName write SetThreadName;
  end;
```

And then extend logging functions to log that information:

```
class procedure NxLog.D(const aMsg: string);
begin
  if Ord(Level) >= Ord(LogDebug) then
    Logger.Output(ThreadName + ' - ' + aMsg);
end;
```

But, if we want to use different implementations for retrieving **ThreadName**, it is obvious that adding a class property with an associated class getter and setter is not very extendable, and there is also the problem of turning thread logging on and off. As we have seen, class helpers are also quite limited, and cannot properly cover all use cases in this particular situation.

We can make this more flexible by moving that implementation into the **INxLogger** interface, where we can easily plug different implementations into different applications.

But we are breaking the single responsibility principle here. The **INxLogger** interface has one simple responsibility: writing log messages to the output. Handling thread names certainly does not fit in here.

Being dogmatic about some principles when you have a particular problem you need to solve can often prevent you from seeing and applying simpler and better solutions. In this case, the clash is more obvious.

Let's say that you want to have two implementations of named threads - one that will merely return a "main or background thread" flag and thread ID, and another that will use a custom thread name.

If we put that functionality inside the logger class, we would have to make two separate implementations for each logger class—**TNxSystemLogger** and **TNxFileLogger**—creating a total of four classes with duplicated code. If you have other logger classes, or you have other kinds of thread name-handling implementations, those numbers will rise.

We already have class inflation, and we've barely started adding extensions.

Besides duplicating code, there is another issue with that approach. Naming threads needs an additional interface method through which you can give a custom name to the thread. Adding that method to the **INxLogger** interface would mean that every logger class must implement that method regardless of whether it supports named threads or not.

The only proper way to solve this is to split the output and thread naming functionalities. However, speed is crucial here, and splitting functionalities will have an impact on the speed. Or maybe not...

The thing is, adding thread information to the log data will be significantly slower than just outputting the passed log message, and the speed improvement you can possibly achieve by stuffing all functionality in the logger will be negligible.

The best way to separate this additional functionality is by introducing another interface responsible for naming threads.

Having said that, naming threads is a rather specific functionality that does not have too many options. We can either turn it off, output the **ThreadID** number or use a custom name. Adding an interface just for that may be an overkill.

Eventually, it makes sense to go back to the initial idea, and implement those directly in the **NxLog** class, then tweak the output based on some flag:

```
NxLog = class
  ...
  protected
    class function GetThreadID: string; static;
    class function GetThreadName: string; static;
    class procedure SetThreadName(const aValue: string); static;
  public
    class property ThreadName: string read GetThreadName write SetThreadName;
    class property ThreadID: string read GetThreadID;
  end;

  class function NxLog.GetThreadID: string;
begin
  if TThread.currentThread.ThreadID = MainThreadID then
    Result := 'MT ' + UIntToStr(TThread.currentThread.ThreadID)
  else
    Result := 'BT ' + UIntToStr(TThread.currentThread.ThreadID);
end;

  class function NxLog.GetThreadName: string;
{$IFDEF POSIX}
var
  Buf: array[0..16] of AnsiChar;
{$ENDIF}
```

```

begin
{$IF defined(ANDROID)}
Result := JStringToString(TJThread.JavaClass.currentThread.getName);
{$ELSEIF defined(POSIX)}
pthread_getname_np(pthread_self, @Buf, SizeOf(Buf));
Result := string(Utf8String(buf));
{$ELSE}
Result := UIntToStr(TThread.CurrentThread.ThreadID);
{$ENDIF}
if TThread.CurrentThread.ThreadID = MainThreadID then
  Result := 'MT ' + Result
else
  Result := 'BT ' + Result;
end;

class procedure NxLog.SetThreadName(const aValue: string);
begin
{$IF defined(ANDROID)}
TJThread.JavaClass.currentThread.setName(StringToJString(aValue));
{$ELSEIF defined(LINUX)}
pthread_setname_np(pthread_self, PUtf8Char(Utf8String(aValue)));
{$ELSEIF defined(MACOS)}
pthread_setname_np(PUtf8Char(Utf8String(aValue)));
{$ELSE}
// not implemented
{$ENDIF}
end;

```

The **ThreadName** and **ThreadID** properties work in the context of the current thread. You can only set or retrieve information about the current thread.

```

procedure TestThreadName;
begin
  TThread.CreateAnonymousThread(
    procedure
    begin
      NxLog.ThreadName := 'MyThread';
      NxLog.D('Thread name test');
    end).Start;
end;

```

33.4.2 Timestamp

While thread identification can be crucial for understanding code flow, a timestamp does not seem like something as important. And in most cases it isn't. But occasionally, a timestamp can come in handy to determine how fast some code runs—it can show you where potential bottlenecks are without involving other kinds of code profiling. It can make it easier to spot whether some part of the code runs much longer than expected, or whether you are overwhelming the system by running some other code more often than necessary.

And sometimes, knowing the time when some event occurred can be crucial for understanding some race conditions or date-time related issues. For instance, seeing that two threads access the same data around the same time in parts of the misbehaving code might imply that data is not properly protected, or you can observe one thread starving others, or if your program crashes or misbehaves because of daylight saving time, you can more easily pinpoint the problem.

When using logging to inspect times, one should keep in mind that the *observer effect* can have a huge impact on code behavior, especially for fast-running code, where the time taken to write a log message can have a much higher impact—to the point that it can completely change the flow.

Because writing timestamps also takes time, it needs to be a configurable option. When it comes to output format, there is no need to fiddle with various formats—this is a debugging tool, after all. Writing timestamps in the ISO-8601 format will do.

There is the question of whether timestamps should be written in local time or UTC. While UTC is a more universal solution, for simple logging it makes no difference which one we will choose. Since retrieving UTC time is a bit more complicated than local time, I will use local time for simplicity. For the purpose of faster formatting, it will be treated as UTC when passed to the `DateToISO8601` function, so there will be no time zone information added.

If, for some reason, logging must store real UTC time, then all that is needed is to replace the call to `Now` with an appropriate UTC time function. Since Delphi 11, you can use `TDateTime.NowUTC` for that.

If you really need faster timestamps, you can always use milliseconds elapsed since system startup, instead of using actual time.

One last thing to keep in mind when logging time: Because the thread-safe `Output` method will serialize passed messages, it will be possible for the order of the logged messages to not match their time. In other words, events that happened sooner might be logged after events that happened later.

33.4.3 Logging level

For filtering and inspecting logs, knowing which logging function was called—debug, info, warning...—can also help, both with finding the appropriate message in the log, as well as the associated code.

Filtering is especially useful if you are watching live messages (on platforms that support that kind of message filtering) and you need to filter out the ones you are interested in.

In order to do that, we can use the logging level we defined already, but besides using it merely as a factor that will decide which message should be written to log and which should not, we also need to write that information to the log.

33.4.4 Combining extensions

We have already determined that the simplest way to combine all extensions is directly in the `NxLog` class. To avoid code repetition in the `D`, `I`, `W...` methods, there is an additional, inline `PrepareOutput` method:

```
type
  TNxLogThread = (LogThreadOff, LogThreadID, LogThreadName);

  NxLog = class
  protected
    ...
    class var ThreadInfo: TNxLogThread;
    class varTimeStamp: Boolean;

    class function PrepareOutput(aLevel: TNxLogLevel; const aMsg: string): string;
      static; inline;
  public
    ...
    class procedure SetThreadInfo(aValue: TNxLogThread); static;
    class procedure SetTimeStamp(aValue: Boolean); static;
    class procedure D(const aMsg: string); static;
    class procedure I(const aMsg: string); static;
    class procedure W(const aMsg: string); static;
    class procedure E(const aMsg: string); static;
    class procedure F(const aMsg: string); static;
  end;

  const
    LogDelimiter = ' - ';
    TNxLogLevelText: array[TNxLogLevel] of string =
      ('OFF' + LogDelimiter,
       'NXFATAL' + LogDelimiter,
       'NXERROR' + LogDelimiter,
       'NXWARNING' + LogDelimiter,
       'NXINFO' + LogDelimiter,
       'NXDEBUG' + LogDelimiter);
```

```
class function NxLog.PrepareOutput(aLevel: TNxLogLevel; const aMsg: string): string;
begin
  Result := TNxLogLevelText[aLevel];
  if TimeStamp then
    Result := Result + DateToISO8601(Now) + LogDelimiter;
  case ThreadInfo of
    LogThreadID : Result := Result + ThreadID + LogDelimiter;
    LogThreadName : Result := Result + ThreadName + LogDelimiter;
  end;
  Result := Result + aMsg;
end;

class procedure NxLog.SetThreadInfo(aValue: TNxLogThread);
begin
  ThreadInfo := aValue;
end;

class procedure NxLog.SetTimeStamp(aValue: Boolean);
begin
  TimeStamp := aValue;
end;

class procedure NxLog.D(const aMsg: string);
var
  Result: string;
begin
  if Ord(Level) >= Ord(LogDebug) then
    begin
      Result := PrepareOutput(LogDebug, aMsg);
      Logger.Output(Result);
    end;
end;

class procedure NxLog.I(const aMsg: string);
var
  Result: string;
begin
  if Ord(Level) >= Ord(LogInfo) then
    begin
      Result := PrepareOutput(LogInfo, aMsg);
      Logger.Output(Result);
    end;
end;
...
```

33.5 Final thread safety check

After all the code is put in place, we need to consider thread safety once again. In real life, you would naturally be thinking about thread safety all the time as you write code, to prevent situations where you will complete some code and then discover that you have thread-unsafe parts that need to be rewritten.

However, doing an additional final check never hurts. In simple code like this logger, it is probably not necessary, but in more complex code, focusing on thread safety after you have completed the code helps discover hidden issues that you might have missed while you were occupied with writing it.

The thread safety of the `Logger` reference is unchanged. Everything that was previously said still stands. The thread safety of the `Level` field is unchanged, too. `ThreadInfo` and `TimeStamp` fall into the same category as `Level`, and everything that is valid for `Level` is valid for them, too.

Just like `Level` can be safely modified at any time during the execution of the program, so can `ThreadInfo` and `TimeStamp`. But, our logging methods—`PrepareOutput`, `D`, `I`, `W`—are as thread-unsafe as they were before. In other words, if one thread changes the configuration, another thread running the logging methods might pick up the new configuration... or maybe not. It might also pick up only half of the configuration changes. But, just like before, that will only result in extra logging information written, or in the case of changing `ThreadInfo` or `TimeStamp`, partially written information.

In any case, that is not a serious thread safety issue. We just have to keep the consequences in mind, if for some reason we want to change the configuration after the application has started using logging functions.

Again, it is important to note that changing the `Logger` interface is not thread-safe, because interface assignment is not a thread-safe operation. It is critical that `Logger` is always set before the application changes the initial logging level from `LogOff` to some other value.

Even if some other threads are already running, we can still change `Logger`, because its access is protected by the initial value of the `Level` field. Once we change the `Level` field, changing `Logger` is not safe, unless we can guarantee that, at the time of the change, there are no other running threads using logging functionality.

And one last thing to keep in mind. Logging frameworks in general, and even more this particular logging solution, will not always follow all good coding practices. This is partially due to the nature of logging: it must be available through the application, being simple to use without the need for additional checks, and must be both thread-safe and fast at the same time.

As written, being accessed through static functions, this logging functionality is not the most testable code ever written. But because it is simple code that will not change much—if at

all—once it is finished and working, there is no need to retest its behavior all the time. It is still flexible enough to allow testing with some constraints.

One thing that is not testable in this particular implementation is **TimeStamp**, because the output depends on the current time. Verifying the correctness of the written timestamp value, beyond format, is not an absolute must-have, and this *lack of testability* is not a serious violation.

The testability of some code that uses logging functions will not be impacted by them, especially if logging is turned off. If there is a need to test the specific log output of some particular code, it is always possible to use a custom logger that will allow verifying its output. The only thing to keep in mind during testing, is that **Logger** is a global, thread-unsafe, state. It can only be changed if there are no other threads running, and if changed, it will have an impact on all code that runs afterwards.

The individual logger implementation classes can be tested, provided that you can capture and verify the output of the **Output** method. For some classes, like the system logger that would be almost impossible, but again, the ability to automatically test them all is not of vital importance.

Chapter 34

Cancellation tokens

One of the important features missing from the *Parallel Programming Library* is cancellation tokens. Also, as a feature, their applicability reaches beyond tasks, and they could be used in other classes and code that provides the ability to cancel some operation.

Cancellation tokens, as such, are not hard to implement. While a custom implementation will lack integration with other classes or frameworks, it is still an indispensable feature.

The most basic form of a cancellation token can be represented by an `IsCanceled` Boolean flag, and a `Cancel` method.

```
type
    INxCancellationToken = interface
        function GetIsCanceled: Boolean;
        procedure Cancel;
        property IsCanceled: Boolean read GetIsCanceled;
    end;
```

And then such a token can be used in the following manner: The token instance is passed as a parameter to a method that supports cancellation, and then the method logic periodically checks whether the token has been canceled. If it has, the method halts any further processing. If the operation is implemented within an anonymous method, the token can also be passed through the variable capture mechanism.

Basically, cancellation tokens provide a universal solution that can be used in various scenarios that would otherwise require slightly different code.

The implementation of such a basic thread-safe cancellation token is extremely simple. The token instance is not reusable, and once canceled, it cannot be reset back to a non-canceled state. This kind of functionality, built around a Boolean flag that can be atomically assigned, does not require any additional protection to be thread-safe. You only need to pay attention to the thread safety of the reference to the token instance.

Thread-safe

```
type
  TNxCancellationToken = class(TInterfacedObject, INxCancellationToken)
  protected
    fIsCanceled: Boolean;
    function GetIsCanceled: Boolean;
  public
    procedure Cancel;
    property IsCanceled: Boolean read GetIsCanceled;
  end;

  function TNxCancellationToken.GetIsCanceled: Boolean;
begin
  Result := fIsCanceled;
end;

procedure TNxCancellationToken.Cancel;
begin
  fIsCanceled := True;
end;
```

34.1 Cancellations without tokens

Before we start exploring how such a token can be used and extended to provide more functionality, let's see some examples of code that uses some form of cancellation. That way we can observe the differences between codebases using different mechanisms, and how using a cancellation token compares to those examples.

However, the full advantage of cancellation tokens over other mechanisms might not be immediately obvious when a token is applied in its simplest form, and when it is used without full integration with other frameworks. To demonstrate the full capabilities of cancellation tokens, subsequent chapters will also explore how the token could have been used if it were more deeply integrated with other parts.

Please note that the examples don't protect against re-entrancy, to keep the code simpler and more focused on the cancellation functionality alone. Examples with cancelling a task and thread also require cleanup after the operation is completed, and that part is also omitted from the code, as there are various strategies to do so. The cleanup alone, like the re-entrancy problem, is not directly connected to the cancellation functionality.

Cancelling an operation with a boolean flag:

```
TMainForm = class(TForm)
...
  fCanceled: Boolean;
end;

procedure TMainForm.CancelButtonClick(Sender: TObject);
begin
  fCanceled := True;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
begin
  fCanceled := False;
  TThread.CreateAnonymousThread(
    procedure
    begin
      DoFoo;
      if fCanceled then Exit;
      DoBar;
      if fCanceled then Exit;
      DoFork;
    end).Start;
end;
```

Cancelling an APL operation:

```
TMainForm = class(TForm)
...
  fAsyncResult: IAsyncResult;
end;

procedure TMainForm.CancelButtonClick(Sender: TObject);
begin
  if Assigned(fAsyncResult) then
    fAsyncResult.Cancel;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
begin
  fAsyncResult := Client.BeginOperation(...);
end;
```

Cancelling a task:

```
TMainForm = class(TForm)
  ...
  fTask: ITask;
end;

procedure TMainForm.CancelButtonClick(Sender: TObject);
begin
  if Assigned(fTask) then
    fTask.Cancel;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
begin
  fTask := TTask.Create(
    procedure
    begin
      DoFoo;
      fTask.CheckCanceled;
      DoBar;
      fTask.CheckCanceled;
      DoFork;
    end);
  fTask.Start;
end;
```

Cancelling a thread:

```
TMainForm = class(TForm)
  ...
  fThread: TFooBarForkThread;
end;

procedure TMainForm.CancelButtonClick(Sender: TObject);
begin
  if Assigned(fThread) then
    fThread.Terminate;
end;

procedure TFooBarForkThread.Execute;
begin
  DoFoo;
  if Terminated then Exit;
```

```
DoBar;
if Terminated then Exit;
DoFork;
end;
```

34.2 Cancellations with a token

Now we can apply a cancellation token in the context of the previous examples and observe the differences.

Using a token instead of a boolean flag:

```
type
  TMainForm = class(TForm)
  ...
  fToken: INxCancellationToken;
  end;

procedure TMainForm.CancelButtonClick(Sender: TObject);
begin
  if Assigned(fToken) then
    fToken.Cancel;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
begin
  fToken := TNxCancellationToken.Create;

  TThread.CreateAnonymousThread(
    procedure
    begin
      DoFoo;
      if fToken.IsCanceled then Exit;
      DoBar;
      if fToken.IsCanceled then Exit;
      DoFork;
    end).Start;
end;
```

The above example with a cancellation token highly resembles using a boolean flag. At first glance, compared to a plain boolean flag, it does have some overhead, because it allocates memory for the token instance.

Using a token to cancel a task:

```
type
  TMainForm = class(TForm)
  ...
  fToken: TNxCancellationToken;
end;

procedure TMainForm.CancelButtonClick(Sender: TObject);
begin
  if Assigned(fToken) then
    fToken.Cancel;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
begin
  fToken := TNxCancellationToken.Create;

  TTask.Run(
    procedure
    begin
      DoFoo;
      if fToken.IsCanceled then Exit;
      DoBar;
      if fToken.IsCanceled then Exit;
      DoFork;
    end);
end;
```

If we compare cancelling a task with a token and cancelling an anonymous thread with a token, we can see that, in terms of cancelling functionality, those two examples have exactly the same code, whereas their tokenless counterparts don't. This is the consistency I was talking about.

However, if we focus only on the task examples, there is a significant difference in behavior, even though it might not be immediately obvious from the code. Namely, `fTask.CheckCanceled` will raise an exception, while using a token will just exit. If you query the state of the task in the first case, you will be able to retrieve the exception information, as well as the task's status, which will show that the task has been cancelled. If you use a token as shown above, the task will look like it has been successfully completed. Of course, you can query the token's status to determine the actual status of a task.

This is the point at which we can extend the most basic token functionality to have more options available for handling cancellation, depending on our use case. With the following extension, we can still query the `IsCanceled` flag and exit without an exception, or we can call `RaiseIfCanceled`, which will raise the same kind of exception as `CheckCanceled`, giving us the same behavior.

```
type
  INxCancellationToken = interface
    function GetIsCanceled: Boolean;
    procedure Cancel;
    procedure RaiseIfCanceled;
    property IsCanceled: Boolean read GetIsCanceled;
  end;

  function TNxCancellationToken.GetIsCanceled: Boolean;
begin
  Result := fIsCanceled;
end;

procedure TNxCancellationToken.Cancel;
begin
  fIsCanceled := True;
end;

procedure TNxCancellationToken.RaiseIfCanceled;
begin
  if fIsCanceled then
    raise EOperationCancelled.Create('Operation canceled');
end;

procedure TMainForm.CancelButtonClick(Sender: TObject);
begin
  if Assigned(fToken) then
    fToken.Cancel;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
begin
  fToken := TNxCancellationToken.Create;

  TTask.Run(
    procedure
    begin
      DoFoo;
      fToken.RaiseIfCanceled;
      DoBar;
      fToken.RaiseIfCanceled;
      DoFork;
    end);
end;
```

34.3 Tokens, tasks, and async results

The previous examples were meant to show the basic workflow for using cancellation tokens, and to compare that workflow and code with other methods of cancelling an operation. They have also shown some differences in behavior and some small improvements in code consistency, but not large enough to justify token usage over existing solutions. If you need to handle multiple operations that would otherwise use a different cancelling system, using tokens will make such code simpler, as you will have to keep a collection of tokens instead of different incompatible instances. You can also use the same token for cancelling multiple operations.

But all of the above is just one part of the picture, and there are even more significant gains to be had by using tokens. The easiest way to explain those is by looking at the .NET framework, and how cancellation tokens as a coding pattern are integrated and used there.

One of the more obvious differences, as we have seen in previous chapters, between using cancellation tokens and other ways to cancel an operation is in inconsistent APIs. For instance, you don't **Cancel** a thread, you **Terminate** it. On the other hand, you can **Cancel** both a task and an async result. But there are inconsistencies between querying the task and async result. The task has a **CheckCanceled** method that raises an exception, which is meant to be used from within the task code, but querying the task's status from external code differs from querying the status of the async result: the task has a **Status** property, and the async result has the **IsCanceled** and **IsCompleted** properties.

Now, those discrepancies in those APIs are incidental, and in theory, they could be more unified to provide more consistency. But even if those discrepancies were removed, there is still an advantage to using cancellation tokens, and even more for having an RTL-provided cancellation token.

This becomes more clear if we take a look at the usage of the **IAsyncResult** interface in .NET, where it is more deeply integrated with the rest of the API. Namely, in .NET, **IAsyncResult** is not only used for its *Asynchronous Programming Model*, the equivalent of Delphi's *Asynchronous Programming Library*, but also in tasks. In .NET, creating a task will not return some task-specific interface, but an **IAsyncResult**, while in Delphi, tasks use a separate **ITask** interface that is in no way related to the APL.

.NET's **IAsyncResult** interface declaration:

```
public interface IAsyncResult
{
    object AsyncState{ get; }
    WaitHandle AsyncWaitHandle{ get; }
    bool CompletedSynchronously{ get; }
    bool IsCompleted{ get; }
}
```

Delphi's **IAsyncResult** interface declaration:

```
type
  IAsyncResult = interface
    function GetAsyncContext: TObject;
    function GetAsyncResultEvent: TMultiWaitEvent;
    function GetCompletedSynchronously: Boolean;
    function GetIsCompleted: Boolean;
    function GetIsCancelled: Boolean;
    function Cancel: Boolean;
    property AsyncContext: TObject read GetAsyncContext;
    property AsyncWaitEvent: TMultiWaitEvent read GetAsyncResultEvent;
    property CompletedSynchronously: Boolean read GetCompletedSynchronously;
    property IsCompleted: Boolean read GetIsCompleted;
    property IsCancelled: Boolean read GetIsCancelled;
  end;
```

The common parts of both interfaces are the state, status and wait handle. The one thing that differs is that the Delphi interface has cancellation support, while the .NET interface doesn't. But that difference alone does not explain the need for a token as a separate entity, as it might be just an arbitrary design choice without any deeper reasoning behind it.

In other words, if we imagine that the .NET **IAsyncResult** supported cancelling the way Delphi does, and if we imagine that Delphi tasks and threads also used **IAsyncResult** to unify the API, the real question left for us to answer is: What can a cancellation token do for us, that an async result cannot?

And the greatest advantage is that cancellation tokens are passed as parameters to the methods that support cancelling, while **IAsyncResult** is returned as a function result.

Being passed as parameters allows us to use tokens in both synchronous or asynchronous calls, promoting code consistency, and it simplifies code in classes that support cancellation. Of course, if you want to cancel a synchronous operation through a token, you still need to invoke that cancellation from a different thread.

By contrast, an async result returned from a function cannot be used for cancelling synchronous methods, only for ones that force asynchronous execution, as the function result will be assigned only after the function call returns. That means a class supporting synchronous and asynchronous operations requires different cancellation code to support both operation types. This also complicates outside code using such a class, as it needs to additionally implement and handle cancellation code, instead of just passing tokens downstream.

Using a cancellation token also simplifies the implementation of classes with cancellable operations as it delegates cancellation to another class that can be reused across all classes that support cancellable operations.

Let's compare different implementations of a simple synchronous class with cancellable operation named **Foo**, one using a cancellation token, and another without it. The difference in the code

required is obvious: A single method, versus three methods and a Boolean flag.

A cancellable operation with a token:

```
type
  TFooOperation = class
  public
    procedure Foo(const Token: INxCancellationToken);
  end;

procedure TFooOperation.Foo(const Token: INxCancellationToken);
begin
  // do some work
  ...
  Token.RaiseIfCanceled;
  // do more work
  ...
  Token.RaiseIfCanceled;
  // do more work
  ...
end;
```

A cancellable operation without a token:

```
type
  TFooOperation = class
  private
    fIsCanceled: Boolean;
    procedure RaiseIfCanceled;
  public
    procedure Cancel;
    procedure Foo;
  end;

procedure TFooOperation.Cancel;
begin
  fIsCanceled := True;
end;

procedure TFooOperation.RaiseIfCanceled;
begin
  if fIsCanceled then
    raise EOperationCancelled.Create('Operation canceled');
end;
```

```
procedure TFooOperation.Foo;
begin
    // do some work
    ...
    RaiseIfCanceled;
    // do more work
    ...
    RaiseIfCanceled;
    // do more work
    ...
end;
```

Adding support for asynchronous calls in the above classes via `IAsyncResult` would yield the same results—the code using a token would be simpler than the one without it. Supporting cancellation in `IAsyncResult`, regardless of the underlying class used, adds another unnecessary layer of complexity.

When it comes to cancelling tasks or threads, passing a cancellation token as a parameter would also simplify their handling.

In the case of tasks, we have seen how the task example with a token has cleaner code: Its task code does not have to capture a reference to the task itself, which can be tricky, as it may cause circular reference and memory leaks, so it requires additional attention.

Using a cancellation token in threads instead of the thread's `Terminate` method simplifies the handling of threads, as it gives you the ability to cancel a thread without holding a thread reference, which opens up the possibility of using self-destroying threads in broader scenarios.

Custom token implementations give most of the mentioned advantages, but an RTL-provided cancellation token would give more consistency across different frameworks, including RTL, as different token implementations cannot be easily mixed. For instance, you cannot easily hold a collection of different tokens.

Even though I am not expecting that the `IAsyncResult` or `ITask` interfaces will change in the future, and that the ability to cancel an operation will be removed from them, it is possible to extend those APIs to use cancellation tokens passed as parameters, and utilize their advantages without breaking existing code.

34.4 Non-cancellable tokens

Sometimes you may want to perform some operation without the ability to cancel it. This can be easily achieved in your own code by simply not checking for cancellation. However, once you start building your code around cancellation token functionality, there will be code where you will need to have both options.

While you can always solve such situations differently, one solution that probably fits this

scenario best is to implement a non-cancellable or empty token. Simply put, this is a token that cannot be cancelled, and as such, one instance (a singleton) can be used everywhere.

```
type
  TNxEmptyCancellationToken = class(TInterfacedObject, INxCancellationToken)
  protected
    function GetIsCanceled: Boolean;
  public
    procedure Cancel;
    procedure RaiseIfCanceled;
    property IsCanceled: Boolean read GetIsCanceled;
  end;

  function TNxEmptyCancellationToken.GetIsCanceled: Boolean;
begin
  Result := False;
end;

procedure TNxEmptyCancellationToken.Cancel;
begin
  // do nothing
end;

procedure TNxEmptyCancellationToken.RaiseIfCanceled;
begin
  // do nothing
end;
```

The cancellation token presented in the previous chapters is extremely simple in both implementation and usage. While that is primarily to simplify the explanation of a workflow, it is also because adding more features in the context of multithreading often comes at a price, as those features may require more protection against concurrency issues.

When you are implementing something for your own usage you can make it as simple as you need it to be. On the other hand, frameworks need to cover broad usage and different scenarios, and as such, require more complex features, or may want to have more clear separation of concerns.

For instance, one of the features the presented cancellation token doesn't implement is giving feedback on whether the operation was really cancelled—interrupted—or the cancellation request was simply ignored. This information can be put into the token itself, but it can also be part of some different data structure.

If you take a look at the .NET framework, you can see an example of such a more elaborate and feature-rich cancellation framework, built around the `CancellationTokenSource` and `CancellationToken` types. That does not mean that all of the features there are a good fit for Delphi. If you are looking for a 3rd-party cancellation token, and even more so if you are writing your own implementation, pay attention to additional features. Make sure they don't cause more problems than they solve, and that you have a good use case for any of those.

Chapter 35

Event bus

Messaging systems are not extremely complicated, and making your own messaging system is a nice opportunity to learn and experiment with code. If you are not too adventurous, you can try modifying `System.Messaging` to make it thread-safe.

Keeping a thread-safe collection of topics and subscribers is not complicated. That is the easiest part of making a multithreaded messaging system. The hardest part is sending messages. Actually, sending messages with blocking calls is also trivial, you just need to iterate through a collection of subscribers and call their event handlers.

Designing a messaging system capable of *posting* messages (using non-blocking calls)... well, that is something completely different.

The main problem with making non-blocking calls to subscribed handler methods is that by the time you make the asynchronous call, the subscriber may already be gone. With a little help from interfaces to handle the lifetime, and anonymous methods' variable capture, that problem can be easily solved.

And this is the place where quickly fixing `System.Messaging` to make it thread-safe will fail. I mean, you can certainly start modifying `System.Messaging` and add all the bells and whistles there, but by the time you are done, it will be more of a rewrite than a small patch.

In that light, creating a thread-safe event bus from scratch is a far better option. When the original implementation cannot be easily tweaked, and a rewrite is in order, it is a good time to *forget all* you know about that particular implementation and really start designing that piece of code from the beginning. In other words, start thinking about the requirements and the concept itself.

That does not mean that some parts of your design and code will not resemble some other implementation. After all, you are just making a slightly different kind of duck, so it is inevitable that it will quack like a duck, walk like a duck...

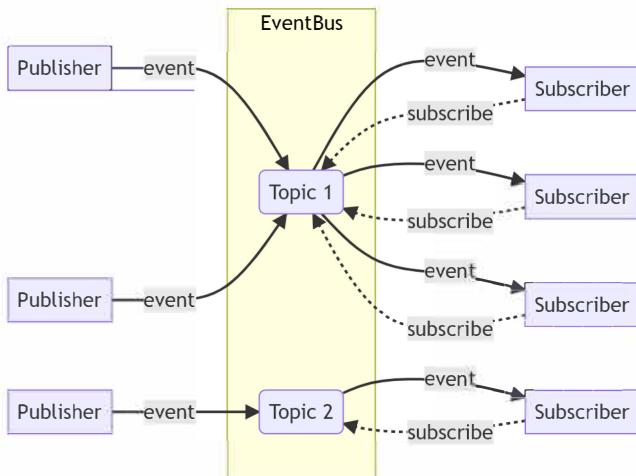
The point is, forgetting what you know about existing implementations and reinventing the wheel will help you avoid design choices that may have been added to meet some other require-

ments and constraints, and that would not be a good fit for yours. Existing implementations also might contain some design flaws or bugs. If you have to carefully craft every part of your solution, the odds of you repeating the same mistakes will be much lower.

In the context of event bus implementation, that means figuring out what the purpose—the functional requirement—of the event bus is. Just as with many other concepts, there is no single answer to that question. There is a basic definition of what an event bus is and what an event bus does, but often enough, such *mere basic* concepts are enhanced and extended with features based on common or very specific use cases.

This book covers thread-safe coding patterns, so one of the requirements—beyond the basics—for our event bus design is thread safety. Let's put the requirements down on paper.

An event bus is a mediator—a communication channel—between publishers and subscribers. It provides a means for publishers to send events, and for receivers (subscribers) to receive and handle those events without them knowing about each other.



Thread-safe event bus requirements

- publishing events
- subscribing and unsubscribing to particular events (topics)
- thread safety
- the ability to handle events in an asynchronous manner
- performance

The first two points are an absolute requirement for any event bus. Without publishers, events and subscribers, there is no event bus. The third point, thread safety, is an additional requirement driven by the desire to use the same event bus instance across multiple threads in a thread-safe manner. The fourth point is not mandatory for achieving thread safety, but without it, a thread-safe event bus would be far less useful, as asynchronous handling enables faster

throughput and prevents long-running event handlers from blocking the whole bus while they run.

Performance also plays a significant role in messaging systems, as it can have quite an impact on the application's behavior. If the messaging system is not fast enough, it can easily turn into a bottleneck. While having a fast messaging system does not, on its own, mean that other parts around it won't suffer from performance issues, those will appear less often, and can be more easily solved than bus congestion.

35.1 Event bus building blocks

The basic building blocks of an event bus:

- events
- the ability to categorize events
- topics or channels - optional, additional categorization
- event delivery options - synchronous, asynchronous (main or background thread)
- publishers - publish method for sending event to subscribers
- subscribers - subscribe and unsubscribe event handlers
- collection of subscribers

The basic methods such an event bus needs would be:

- A **Subscribe** function with delivery option and event type parameters, that returns a subscription instance stored in the collection of subscribers, and which is also used for canceling and unsubscribing
- An **Unsubscribe** method for unsubscribing a particular subscription instance
- Methods for sending events to subscribers—technically, we could only have a single method, but having more configurable options is more flexible and usable in different scenarios
 - A **Send** method for sending an event to subscribers, with a delivery parameter overriding subscription delivery options
 - A **Post** method for sending an event to subscribers, based on the delivery options specified by each particular subscription

35.1.1 Events and categories

The main actors in the event bus are, well, events. The base requirement for all events is that we need to be able to distinguish between different event types. The next requirement is that we need to be able to pass some data along with an event. There are certain types of events that don't require any data, we just need to know that a particular event happened, but most of them do need additional data to be useful. And the last crucial requirement is having automatic

memory management handling: Manual memory management is at odds with asynchronous dispatching to multiple handlers, as we don't know when the last event handler will process some event and when we can safely free the event instance.

In **System.Messaging**, events are objects—they can easily be categorized by class, and different classes (object types) can carry any data we can think of. For instance, the Android OS messaging system (equivalent of Windows messaging) also uses objects, and facilitates garbage collection for memory management.

There are several approaches that can help solve the manual memory management problem (like cloning and tracking handlers progress), but fortunately, Delphi also has automatic memory management for objects that implement interfaces. Using a reference-counted class as our base event type solves the memory management problem in a thread-safe event bus.

Allocating objects (reference-counted or not) always adds some overhead. For frequent messages that just need to notify subscribers that some event happened, but don't need to pass any additional data, the same object instance could be reused instead of constructing new ones every time an event happens.

Besides interfaces, we can also use value types or strings, which are also automatically managed. Thread safety for reference-counted types in asynchronous handling is ensured by the ARC mechanism, where clearing (going out of scope) of any reference-counted type instance is a thread-safe operation.

Strings are also one of the most versatile and powerful and yet simple to use Delphi type. We can use a string to define the event type, and we can certainly pack any data inside the string. We can also declare different string types for different events—this is the same approach we would use for interface-based event types or any other type.

While strings are simple to use, they are bad from a performance standpoint. Not so much because reference counting induces some overhead, but because parsing data from the string is far from being fast.

We could also use different record types for passing different event categories, but as soon as you use a record type that holds anything beyond a few bytes' worth of simple types, value type semantics and content copying for asynchronous calls will have a negative impact on performance.

When all options are considered, interfaces are the most flexible approach suitable for, if not all, then certainly *most* use cases. Ideally, we should be able to use any of those types with value semantics or automatic memory management as events. This is where generics come into play.

But using generics imposes some constraints, and we need to see what kind of implications generics have for code using the event bus, as we would want to avoid unnecessary complications or extreme verbosity.

Interface based events

For interface-based events, we would have the following (simplified) type declarations:

```

type
  // Event
  INxEvent<T> = interface
    ['{24708666-A690-4056-8C62-E072DED7AE04}']
    function GetValue: T;
    property Value: T read GetValue;
  end;

  // Subscription
  INxEventSubscription = interface
    ['{15BE488F-CFE3-4EFB-A3DA-910D0C443D50}']
    procedure Cancel;
    function GetIsCanceled: Boolean;
    property IsCanceled: Boolean read GetIsCanceled;
  end;

  // Subscribed event handler
  TNxEventMethod<T: IInterface> = procedure(const aEvent: T) of object;

  // Event bus
  TNxHorizon = class
  public
    function Subscribe<T: IInterface>(aObserver: TNxEventMethod<T>):
      INxEventSubscription;
    procedure Unsubscribe(const aSubscription: INxEventSubscription);
    procedure Post(const aEvent: IInterface);
  end;

```

And then code snippets using the interface-based event bus would look like:

```

var
  Horizon: TNxHorizon;
  Subscription: INxEventSubscription;

```

Subscribing/unsubscribing event handlers:

```

Subscription := Horizon.Subscribe<INxEvent<TFoo>>(OnFoo);

Horizon.Unsubscribe(Subscription);

```

Sending (posting) events, where **New** is a class function that constructs and returns a new instance of a given type:

```
Horizon.Post(TNxEvent<TFoo>.New(...));
```

Generic events

For a generic event bus, we don't need an event declaration, because events can be of literally any type, even regular objects—though, as we already know, they are not suitable for use because of the manual memory management. The subscription interface declaration is the same as in the non-generic version.

```
type
  // Subscription
  INxEventSubscription = interface
    ['{15BE488F-CFE3-4EFB-A3DA-910D0C443D50}']
    procedure Cancel;
    function GetIsCanceled: Boolean;
    property IsCanceled: Boolean read GetIsCanceled;
  end;

  // Subscribed event handler
  TNxEventMethod<T> = procedure(const aEvent: T) of object;

  // Event bus
  TNxHorizon = class
  public
    function Subscribe<T>(aObserver: TNxEventMethod<T>): INxEventSubscription;
    procedure Unsubscribe(const aSubscription: INxEventSubscription);
    procedure Post<T>(const aEvent: T);
  end;
```

Subscribing/unsubscribing event handlers:

```
Subscription := Horizon.Subscribe<TFoo>(OnFoo);

Horizon.Unsubscribe(Subscription);
```

Sending (posting) events:

```
Horizon.Post(TFoo.New(...));
```

As you can see, the difference in code is minimal, and the generic version is even simpler than the interface-based one.

However, our generic **Post** method relies on type inference introduced in 10.3 Rio. In older versions of Delphi which don't have type inference, we need to explicitly write the type signature.

```
Horizon.Post<TFoo>(TFoo.New(...));
```

While the generic approach has some advantages, it also has some disadvantages, and we need to see whether the restrictions imposed will make this approach unsustainable, or we can live with them without compromising the usability of the framework.

The generic `TNxHorizon` class needs parameterized methods for simplified use, as it is simpler to extract type information from the generic placeholder type `T` inside the bus than to manually pass the type to the `Subscription`, `Post` and `Send` methods. Because of that we cannot use interfaces to declare a generic `INxHorizon` bus API, and consequently, we cannot use automatic memory management for such bus instances.

Generally, event buses are used as singleton instances, or as long lived parts of some other classes or frameworks, and manual memory management shouldn't be an issue. As far as the API is concerned, the bus itself does not require various implementations on top of different class hierarchies, so we don't need interfaces for API abstraction purposes.

Even though the final decision was to use a generic version of the events, we can still make use of the `INxEvent<T>` interface, and implement a base reference-counted event class that will simplify handling various object-based events. In that light, it also makes sense to have a record-based generic event type. Having those generic event types is purely matter of reusing code for simpler use cases, you can still use whichever other type you want for events.

The following interface-based event type implementation takes ownership over passed object instances, and will automatically manage their memory. For non-owned object instances, the record-based event type is more appropriate, but you need to ensure the wrapped object is not released as long as some event handler might be using it. In the generic event bus, reference-counted classes can be directly used for events, and don't require any wrapper.

```
type
  TNxEventObject<T> = class(TInterfacedObject, INxEvent<T>)
  protected
    fValue: T;
    function GetValue: T;
  public
    constructor Create(const aValue: T);
    destructor Destroy; override;
    property Value: T read GetValue;
    class function New(const aValue: T): INxEvent<T>;
  end;

constructor TNxEventObject<T>.Create(const aValue: T);
begin
  fValue := aValue;
end;
```

```
destructor TNxEventObject<T>.Destroy;
var
  Obj: TObject;
begin
  if PTypeInfo(TypeInfo(T)).Kind = tkClass then
    begin
      PObject(@Obj)^ := PPointer(@fValue)^;
      Obj.Free;
    end;
  inherited;
end;

function TNxEventObject<T>.GetValue: T;
begin
  Result := fValue;
end;

class function TNxEventObject<T>.New(const aValue: T): INxEvent<T>;
begin
  Result := TNxEventObject<T>.Create(aValue);
end;
```

The record-based event type does not involve heap allocations, and is suitable for passing small-sized value types (as well as reference types) that don't require manual memory management. For larger value types, using the interface-based event type is more appropriate, as it avoids excessive value copying while dispatching events.

```
type
  TNxEvent<T> = record
private
  fValue: T;
  function GetValue: T;
public
  constructor New(const aValue: T);
  property Value: T read GetValue;
end;

constructor TNxEvent<T>.New(const aValue: T);
begin
  fValue := aValue;
end;
```

```

function TNxEvent<T>.GetValue: T;
begin
  Result := fValue;
end;

```

35.1.2 Event delivery options

Event delivery options are not strictly necessary, and specific delivery requirements could be handled inside each event handler. But handling delivery is boring, repeatable code that would obstruct code clarity in each event handler. Handling the delivery in the bus itself is by far the best option.

The possible delivery options would be:

- synchronous in the context of the calling thread
- asynchronous in a random background thread
- synchronous in the context of the main thread
- asynchronous in the context of the main thread

Which can be translated to the following enumerated type:

```

type
  TNxHorizonDelivery = (
    Sync,      // Run synchronously on current thread - BLOCKING
    Async,     // Run asynchronously in a random background thread
    MainSync, // Run synchronously on main thread - BLOCKING
    MainAsync // Run asynchronously on main thread
  );

```

Synchronous delivery in the context of the main thread is the least useful option, and also rather a poor delivery method for many reasons. It can be easily simulated with synchronous event handling and calling `TThread.Synchronize` inside the event handler.

From that perspective, omitting synchronous main thread event dispatching seems like a good choice—it will be used rarely, if at all, and not having it as an option prevents people from casually shooting themselves in the foot by killing the performance of the whole bus.

For that matter, any kind of synchronous processing can kill the bus' performance, so we may as well remove all synchronous processing. But it is not that simple.

Yes, synchronous processing blocks the bus while the event handler runs, but asynchronous dispatching is a bit slower than merely calling the event handler in place. If you really need fast throughput, then synchronous handling in the context of the calling thread, provided that the event handlers also run fast, is the fastest option. But that is not all. Any kind of asynchronous

handling will also involve a queue, and if you fill up the queue too fast, you have another problem. See: *Back-pressure*

Which kind of dispatching you will use really depends on the situation and particular code. So removing synchronized handling in the context of the calling thread is a bad idea after all, as it removes flexibility from the bus and forces a *one size fits all* approach, which seldom works nicely. In that light, removing the option for synchronized dispatch in the context of the main thread, also removes some flexibility, because synchronizing with the main thread in the event handler not only complicates code in the handler, but also imposes a performance penalty on all events dispatched to that handler, even ones that were being sent from the main thread. Of course, you can always have separate handlers for separate dispatching needs, but that is not the best option, as it leads to unnecessary code duplication.

So, while asynchronous dispatching will be commonly used, if the need arises, you can always use synchronous dispatching. And if one bus gets overwhelmed with events, you can always use additional dedicated buses for very frequent events that require synchronous dispatching and would cause trouble in the shared event bus instance.

Synchronous and main-thread dispatching is straightforward. There is only one way you can implement any of those: synchronous just calls the event handler, synchronous in main thread is achieved with `TThread.Synchronize`, and asynchronous dispatching in the main thread needs `TThread.ForceQueue`. `ForceQueue` is used instead of `Queue`, to force asynchronous dispatching of event handlers in cases where the publisher posts events from the context of the main thread.

When it comes to asynchronous dispatching—running the event handler in a background thread—there are different ways to achieve such dispatching. The simplest and most effective solution would be to use tasks from the PPL (or any similar library) to invoke the event handler in the context of a background thread.

However, running a task opens up some questions. For instance, should we use the default thread pool or have a dedicated thread pool for the event bus? If we decide to use a dedicated thread pool, do we create a pool for each event bus, or use a shared thread pool instance for all bus instances?

This is where the YAGNI—*You Aren't Gonna Need It*—principle kicks in.

Using the default thread pool is perfectly fine for an event bus. This default thread pool is also used across the core Delphi frameworks, and if you are running plenty of heavy and long-running tasks that might exhaust the available threads and cause problems with timely event dispatching, then you should probably use a dedicated thread pool for such tasks, instead of causing problems for the rest of the code. Besides, if your application imposes such stress on the system that all threads in the default pool are way too busy for too long, then you have bigger problems.

What if other possibly running tasks are not a problem, but some of the event handlers can launch long-running tasks that can overwhelm the default thread pool?

In such cases, you will have more control over your code if you handle such scenarios in the event handlers rather than in the event bus. In other words, if you know that your event handler code will run for a significant amount of time and that you will have to deal with a large number of

such events in a small time frame, threatening to overwhelm the default thread pool, then it is better that you spawn another task on a specific thread pool, or even use an additional dedicated thread from within such an event handler. In some scenarios, increasing the maximum number of threads in the default pool is also one of the options, but this solution is more suitable if you have more constant work loads than for solving occasional peaks.

You should also keep in mind that having more threads running on a stressed system will not make the application run faster, but it will slow everything down. In such cases, it is hard to give generalized advice on how to solve a particular problem, and you may need to experiment in order to find the most appropriate solution. It is also important to conduct stress tests on different hardware—with emphasis on the number of CPUs, as they are one of the core variables that impact the default thread pool size and other behaviors that depend on number of running threads. This is especially important for CPU-bound tasks, while additional threads working on IO-bound tasks will not cause too much trouble.

The presented event bus class uses tasks running on the default thread pool as the simplest solution that covers most, if not all, use cases. If you feel like none of the above solutions work well for your use case, you can easily extend the presented event bus class to allow passing the thread pool as a parameter to a bus constructor, and thus adding more configurable behavior where each event bus instance will hold its own thread pool. You should allow using shared thread pools, so you can simply pass the default or any other shared thread pool to the event bus.

There are also other possible approaches and more complicated solutions that could include specifying a thread pool for each subscription, or passing a thread pool as an additional parameter when sending messages, but those options would make the event bus and the code that uses it more complicated, and solving rare problems directly in the event handler is a simpler and more flexible approach than imposing some weird solution on the event bus itself.

If you are using some older Delphi version that does not have PPL support, you don't want to use some 3rd party library, and you don't need a highly efficient event bus, only the ability to run handlers in the context of a background thread, you can also create anonymous threads for dispatching asynchronous event handlers.

35.1.3 Publishing

An event bus doesn't need to maintain a collection of publishers, the only thing we need is a mechanism for publishing (sending) events to the bus. This can be a single method with a mandatory event parameter and various other parameters to define specific delivery behavior, or multiple ones that will separately cater for different behavior needs.

In this particular bus implementation, there will be two methods: `Post` with only an event parameter, and `Send` with an event parameter and a delivery option that overrides the default delivery option defined for each subscription. And now we can fully declare the event bus' public API:

```
type
TNxHorizon = class
public
  function Subscribe<T>(aDelivery: TNxHorizonDelivery;
    aObserver: TNxEventMethod<T>): INxEventSubscription;
  procedure Unsubscribe(const aSubscription: INxEventSubscription);
  procedure Post<T>(const aEvent: T);
  procedure Send<T>(const aEvent: T; aDelivery: TNxHorizonDelivery);
end;
```

35.1.4 Subscribers

Subscribers (event handlers) will be procedural types. In Delphi, there are three kinds of procedural types we can use: standalone procedures, object methods and anonymous methods. To simplify the code presented here, only object methods will be used, but a real-life event bus implementation would allow using anonymous methods, too.

One of the challenges in asynchronous event handling is solving race conditions when the subscriber is destroyed while there are some already dispatched, but unhandled events. Luckily, we can use interfaces and their automatic memory management to solve that issue.

By storing a subscription as an interface reference, it will keep the subscription instance alive, even if the subscriber is destroyed in the meantime. Thus, we can safely check whether the subscription is still active after the event handler has been asynchronously dispatched, just immediately before it is invoked.

Being able to safely invoke an event handler is not enough to achieve full thread safety, as the event handler may run on a different thread from the thread that owns the subscriber. We still need to ensure the subscriber is not destroyed while the event handler itself is running—which is not as simple a problem as it might seem at first glance. In languages with fully automatic memory management, that is a non-issue, as either taking a reference to a method keeps the associated object alive, or you cannot even use references to methods, and subscribers will be objects and not methods.

One of the solutions for that problem would be to subscribe objects instead of methods, and enforcing automatic memory management for subscribers. But in the context of Delphi code, such a constraint means losing flexibility, and it just introduces another level of complexity if you want to use the event bus in existing code, where most classes are not reference-counted. And even if you use reference-counted subscribers, eventually you will still have problems if such a subscriber is owned by a non-reference-counted class, or needs to interact with such class instances. In other words, enforcing automatic memory management on subscribers doesn't fully solve the original problem and it merely moves it to another place in your code.

A better, more universal solution comes in the form of waitable subscriptions. Since subscribers also maintain their subscription(s), instead of just cancelling the subscription and unsubscribing

in the event of their destruction, they can also wait for any running event handlers to finish. This solves the clean shutdown issue.

Waitable subscriptions can be implemented with the help of the `TCountdownEvent` synchronization object from the `System.SyncObjs` unit. The countdown event instance is created with some non-zero count value, and will be signalled when that count reaches zero. During the time it is non-zero, the count can be increased, and this mechanism is perfect for protecting running event handlers.

Initially, the subscription countdown event is created with count one, and before each event handler is executed, the subscription will try to increase the count. If the count is successfully increased, that will prevent the subscription count from reaching zero while the event handler is running. If it is not increased, that means the subscription is no longer active. After the event handler finishes its run, the count is decreased.

Waiting for subscriptions is a three-step process: First, it cancels the subscription to prevent high-frequency publishers from triggering event handlers from that point on, then the count is decreased to counteract the initial value, and then the subscription waits for the countdown event to be signalled. If there are no event handlers running, that will already have happened, and the subscription owner can be immediately released at that point. If some event handlers are still running, the subscription owner will still be intact until they finish.

To implement all necessary functionality, the subscription interface is extended with the `BeginWork`, `EndWork`, and `WaitFor` methods, where the `xxxWork` methods are used for increasing/decreasing the work count in the countdown event, and `WaitFor` is used to wait for the countdown signal. If `BeginWork` returns `False`, that means the countdown event has already been signalled, and we must not invoke the event handler.

To allow cancelling the subscription early in the process, which can also mean issuing multiple cancellation requests, the `Cancel` method just sets the `IsCanceled` flag, and does not decrease the count in the countdown event. That way, it can be freely called multiple times, without decreasing the count too many times.

```
type
  INxEventSubscription = interface
    ['{15BE488F-CFE3-4EFB-A3DA-910D0C443D50}']
    function BeginWork: Boolean;
    procedure EndWork;
    procedure WaitFor;
    procedure Cancel;
    function GetIsCanceled: Boolean;
    property IsCanceled: Boolean read GetIsCanceled;
  end;
```

Adding a countdown event to each subscription adds some overhead in both the memory and the time needed to subscribe each event handler. A more versatile event bus implementation would allow binding multiple, related subscriptions to a single countdown event and its canceling

behavior. In that case, we would have two kinds of subscription implementations: a heavy implementation holding all the necessary data and functionality on its own, and a lightweight implementation that would delegate all other functionality to a heavy, parent subscription instance, and it would only hold the associated event handler.

You have probably noticed that the subscription interface doesn't contain any information about event categorization, the event handler, and delivery options. This is because the purpose of that public interface, besides enabling ARC, is to provide access to the subscription instance outside the event bus, and an API for actions that are meaningful in that external context. All other data is relevant only for handling the subscription within the bus, and not having those exposed in the public API makes that API simpler, and its intended usage clearer.

The subscription API doesn't require different implementations, so we will have only a single implementation of the `INxSubscription` interface—the `TNxSubscription` class—and we can safely use typecasting of that class in the event bus code to access other subscription data. The `TNxSubscription` class is implemented in the interface section of the event bus unit, but it can also be hidden in the implementation section. Having it in the interface section gives us the ability to unit test that class separately from the bus functionality.

We also need to maintain a thread-safe collection of subscribers. A dictionary with the event type as a key, and a value holding a list of matching subscriptions, is the most optimal solution that will allow fast event dispatching.

To minimize locking operations, the used dictionary class is not thread-safe on its own, but it will be protected by a lock within the event bus. Choosing the most appropriate lock is a bit more complicated matter. Because the collection will be read more than it will be written to, MREW locks are more suitable than exclusive locks.

But, the fastest Delphi-provided MREW lock, `TLightweightMREW` is not reentrant for write access, and read access cannot be upgraded to write access. The code in the event handlers could trigger invalid access to an MREW lock, and we need a more flexible lock in this case. `TMultiReadExclusiveWriteSynchronizer` fits our requirements. On non-Windows platforms, it is equivalent to `TSimpleRWSync`, which uses `TMonitor`, so if you target those platforms, you can directly use a `TMonitor` on the subscriber collection instead of allocating another object.

You can also use any other lock, even an exclusive lock, as long as it satisfies the previously mentioned conditions: it must be reentrant, and if it is an MREW lock, it must allow upgrading of read access to write access.

There is another important scenario we need to think about when talking about the thread safety of the subscriber collection. Namely, if we call the event bus' `Unsubscribe` method from a synchronously dispatched event handler, we will effectively break the collection. Specifically, we will delete the subscription from the collection while the collection is being iterated through.

This particular problem is not actually caused by threads, and it is something that would happen in a single thread if we were to iterate through the collection and randomly delete its elements during iteration. Because recursive access to the collection from a single thread is the issue here, locking the collection will not help, as we will be able to reacquire the lock when unsubscribing from the same thread.

However, locking will be a part of the final solution for this scenario. If we cancel the subscription immediately to prevent further event dispatching, and then delegate the process of deleting the subscription from the collection to another thread, locking the collection in another thread will succeed only between iterations. In other words, if we call **Unsubscribe** from another background thread, the write lock in that thread will ensure that no other thread has access to the collection of subscribers while we are modifying it and removing a subscription.

To make the process of unsubscribing easier in such a scenario, there is an additional **UnsubscribeAsync** method that handles the complete process: cancelling the subscription and running a task in which the subscription will be deleted from the collection.

If the event handler is being asynchronously dispatched, it will be running in a different thread from the iterating one, and we can safely call the regular **Unsubscribe** method from such a handler.

Calling **Subscribe** from any event handler will not cause problems because subscribing appends the new subscription at the end and doesn't modify part of the collection that is being iterated through.

35.1.5 Topics or channels

In addition to basic event categorization, in real-life scenarios we may need the ability to fine-tune categorization for the same type of events. While we can always define separate event types to match our needs, a much simpler and more flexible option is to define subscription topics or channels.

When defining channels, we can use the same approach we used for categorizing events, and define new types for each channel, or we can use strings to simply name channels. When it comes to using strings, using case-sensitive strings is the fastest option. Any kind of case insensitivity, even when constricted to the base ASCII character set, will be a real performance killer.

Dealing with types is much faster than dealing with strings, but using string names for channels is the more configurable and therefore more flexible option. If speed is paramount in some scenarios, one can always use event subtyping.

One of the other possible solutions is using separate event bus instances for each channel, instead of extending the core bus functionality, and organizing those instances in a thread-safe collection. Keeping channels outside of the core event bus functionality also has fewer (or zero) negative impacts on event bus performance, compared to solutions where you need to pass an additional parameter and act on its contents.

Having separate bus instances allows you to keep a direct reference to a particular channel bus instance, and dispatch events at the same rate as with the basic bus implementation. You can still keep a dictionary for organizing all bus instances around channels for accessing bus instances that are not mission-critical.

Because additional event categorization (or other possible features) steps outside the basic requirements for an event bus, the presented event bus does not support that feature out of

the box. One reason for keeping this implementation as simple as possible is focusing on the required code to implement a minimal, yet usable event bus. A minimal implementation allows better understanding of the code and its workflow, where additional features aren't standing in our way.

My personal preference when extending an event bus would be to use separate event bus instances for channels, as this approach preserves the bus speed. This solution is also the easiest to build upon existing code without changing previously implemented parts.

35.2 Event bus implementation

The following is the full code for a generic event bus. `TNxEvent<T>` and `TNxEventObject<T>` are not included, as their full implementation can be found in previous chapters, and they are not an essential part of the event bus implementation.

Some additional explanations of the provided code that were not covered previously:

The `INxEventSubscription` interface is extended with an `IsActive` property which is just opposite of `IsCanceled`. It is not strictly necessary, but it makes the code easier to follow if you want to check whether the subscription has not been canceled yet.

The `BeginWork/EndWork` methods are used internally in the event bus code, but those are left as part of the public API because they can be useful in event handlers that might invoke additional asynchronous code, and those methods can be used to prevent premature destruction in the same way they are used to ensure the subscriber is still alive while its event handler runs.

In the event bus, synchronous dispatching is done directly from the `Send` and `Post` methods, and a separate `DispatchEvent` method is used for asynchronous dispatching. This is a speed optimization for synchronous dispatching, because asynchronous dispatching requires the usage of an anonymous method instance that needs to be initialized and finalized, which takes some time, no matter how small.

`NxHorizon` provides a default event bus instance. This singleton instance could also be declared inside the `TNxHorizon` class, but I like maintaining a clean separation between a class and its singleton holder. It is also easier to extend the singleton class with additional functionality without accidentally trampling over code that is not subject to change. It is also easier to extract such a singleton holder into a separate unit and provide different singleton implementations if needed.

If you want to extend the event bus with channel support, the `NxHorizon` class would be a good place to do so. This is where you can maintain an additional collection of channels and associated bus instances (you can also implement channels in a completely separate class, if you prefer having additional separation), and still maintain unchanged access to the default singleton instance that does not belong to any particular channel.

```
unit NX.Horizon;

interface

uses
  System.SysUtils,
  System.Classes,
  System.Generics.Collections,
  System.TypInfo,
  System.SyncObjs,
  System.Threading;

type
  TNxEventMethod<T> = procedure(const aEvent: T) of object;
  TNxEventMethod = TNxEventMethod<TObject>;

  TNxHorizonDelivery = (
    Sync,      // Run synchronously on current thread - BLOCKING
    Async,     // Run asynchronously in a random background thread
    MainSync,  // Run synchronously on main thread - BLOCKING
    MainAsync // Run asynchronously on main thread
  );

  INxEventSubscription = interface
    ['{15BE488F-CFE3-4EFB-A3DA-910D0C443D50}']
    function BeginWork: Boolean;
    procedure EndWork;
    procedure WaitFor;
    procedure Cancel;
    function GetIsActive: Boolean;
    function GetIsCanceled: Boolean;
    property IsActive: Boolean read GetIsActive;
    property IsCanceled: Boolean read GetIsCanceled;
  end;

  TNxEventSubscription = class(TInterfacedObject, INxEventSubscription)
  protected
    fCountdown: TCountdownEvent;
    fEventMethod: TNxEventMethod;
    fEventInfo: PTypeInfo;
    fDelivery: TNxHorizonDelivery;
    fIsCanceled: Boolean;
    function GetIsActive: Boolean;
    function GetIsCanceled: Boolean;
  end;
```

```
public
  constructor Create(aEventInfo: PTypeInfo;
    aDelivery: TNxHorizonDelivery; aObserver: TNxEventMethod);
  destructor Destroy; override;
  function BeginWork: Boolean;
  procedure EndWork;
  procedure WaitFor;
  procedure Cancel;
  property IsActive: Boolean read GetIsActive;
  property IsCanceled: Boolean read GetIsCanceled;
end;

TNxHorizon = class
protected
  fLock: IReadWriteSync;
  fSubscriptions: TDDictionary<PTypeInfo, TList<INxEventSubscription>>;
  procedure DispatchEvent<T>(
    const aEvent: T; const aSubscription: INxEventSubscription;
    aDelivery: TNxHorizonDelivery; aObserver: TNxEventMethod);
public
  constructor Create;
  destructor Destroy; override;
  function Subscribe<T>(aDelivery: TNxHorizonDelivery;
    aObserver: TNxEventMethod<T>): INxEventSubscription;
  procedure Unsubscribe(const aSubscription: INxEventSubscription);
  procedure UnsubscribeAsync(const aSubscription: INxEventSubscription);
  procedure Post<T>(const aEvent: T);
  procedure Send<T>(const aEvent: T; aDelivery: TNxHorizonDelivery);
end;

NxHorizon = class
protected
  class var
    fInstance: TNxHorizon;
  class constructor ClassCreate;
  class destructor ClassDestroy;
public
  class property Instance: TNxHorizon read fInstance;
end;

implementation

{ TNxEventSubscription }
```

```
constructor TNxEventSubscription.Create(aEventInfo: PTypeInfo;
  aDelivery: TNxHorizonDelivery; aObserver: TNxEventMethod);
begin
  fEventInfo := aEventInfo;
  fDelivery := aDelivery;
  fEventMethod := aObserver;
  fCountdown := TCountdownEvent.Create(1);
end;

destructor TNxEventSubscription.Destroy;
begin
  fCountdown.Free;
  inherited;
end;

function TNxEventSubscription.BeginWork: Boolean;
begin
  Result := (not fIsCanceled) and fCountdown.TryAddCount;
end;

procedure TNxEventSubscription.EndWork;
begin
  fCountdown.Signal;
end;

procedure TNxEventSubscription.WaitFor;
begin
  fIsCanceled := True;
  fCountdown.Signal;
  fCountdown.WaitFor;
end;

function TNxEventSubscription.GetIsActive: Boolean;
begin
  Result := not fIsCanceled;
end;

function TNxEventSubscription.GetIsCanceled: Boolean;
begin
  Result := fIsCanceled;
end;

procedure TNxEventSubscription.Cancel;
begin
```

```
    fIsCanceled := True;
end;

{ TNxHorizon }

constructor TNxHorizon.Create;
begin
    fLock := TMultiReadExclusiveWriteSynchronizer.Create;
    fSubscriptions := TObjectDictionary<PTypeInfo,
        TList<INxEventSubscription>>.Create([doOwnsValues]);
end;

destructor TNxHorizon.Destroy;
begin
    fSubscriptions.Free;
    inherited;
end;

function TNxHorizon.Subscribe<T>(aDelivery: TNxHorizonDelivery;
    aObserver: TNxEventMethod<T>): INxEventSubscription;
var
    SubList: TList<INxEventSubscription>;
begin
    Result := TNxEventSubscription.Create(
        PTypeInfo(TypeInfo(T)), aDelivery, TNxEventMethod(aObserver));
    fLock.BeginWrite;
    try
        if not fSubscriptions.TryGetValue(PTypeInfo(TypeInfo(T)), SubList) then
            begin
                SubList := TList<INxEventSubscription>.Create;
                fSubscriptions.Add(PTypeInfo(TypeInfo(T)), SubList);
            end;
        SubList.Add(Result);
    finally
        fLock.EndWrite;
    end;
end;

procedure TNxHorizon.Unsubscribe(const aSubscription: INxEventSubscription);
var
    SubList: TList<INxEventSubscription>;
begin
    aSubscription.Cancel;
    fLock.BeginWrite;
```

```
try
  if fSubscriptions.TryGetValue(
    TNxEventSubscription(aSubscription).fEventInfo, SubList) then
    SubList.Remove(aSubscription);
  finally
    fLock.EndWrite;
  end;
end;

procedure TNxHorizon.UnsubscribeAsync(const aSubscription: INxEventSubscription);
var
  [unsafe] lProc: TProc;
begin
  // cancel as soon as possible
  aSubscription.Cancel;
  lProc :=
    procedure
    begin
      Unsubscribe(aSubscription);
    end;
  TTask.Run(lProc);
end;

procedure TNxHorizon.DispatchEvent<T>(
  const aEvent: T; const aSubscription: INxEventSubscription;
  aDelivery: TNxHorizonDelivery; aObserver: TNxEventMethod);
var
  [unsafe] lProc: TProc;
begin
  lProc :=
    procedure
    begin
      if aSubscription.BeginWork then
        try
          TNxEventMethod<T>(aObserver)(aEvent);
        finally;
          aSubscription.EndWork;
        end;
    end;
  end;

  case aDelivery of
  // Synchronous dispatching is done directly in Send and Post methods
  // Sync :
  //   begin
  //     TNxEventMethod<T>(aObserver)(aEvent);
```

```
//      end;
Async :
begin
  TTask.Run(lProc);
end;
MainSync :
begin
  if TThread.CurrentThread.ThreadID = MainThreadID then
    lProc
  else
    TThread.Synchronize(nil, TThreadProcedure(lProc));
end;
MainAsync :
begin
  TThread.ForceQueue(nil, TThreadProcedure(lProc));
end;
end;

procedure TNxHorizon.Post<T>(const aEvent: T);
var
  SubList: TList<INxEventSubscription>;
  Sub: TNxEventSubscription;
  i: Integer;
begin
  fLock.BeginRead;
  try
    if fSubscriptions.TryGetValue(PTypeInfo(TypeInfo(T)), SubList) then
      for i := 0 to SubList.Count - 1 do
        begin
          Sub := TNxEventSubscription(SubList.List[i]);
          if Sub.IsActive and (Sub.fMethodInfo = PTypeInfo(TypeInfo(T))) then
            begin
              // check if delivery is Sync because
              // DispatchEvent has anonymous methods setup
              // that is unnecessary for synchronous execution path
              if Sub.fDelivery = Sync then
                begin
                  if Sub.BeginWork then
                    try
                      TNxEventMethod<T>(Sub.fEventMethod)(aEvent);
                    finally
                      Sub.EndWork;
                    end;
                end;
            end;
  end
```

```
        else
            DispatchEvent(aEvent, Sub, Sub.fDelivery, Sub.fEventMethod);
        end;
    end;
finally
    fLock.EndRead;
end;
end;

procedure TNxHorizon.Send<T>(const aEvent: T; aDelivery: TNxHorizonDelivery);
var
    SubList: TList<INxEventSubscription>;
    Sub: TNxEventSubscription;
    i: Integer;
begin
    fLock.BeginRead;
    try
        if fSubscriptions.TryGetValue(PTypeInfo(TypeInfo(T)), SubList) then
            for i := 0 to SubList.Count - 1 do
                begin
                    Sub := TNxEventSubscription(SubList.List[i]);
                    if Sub.IsActive and (Sub.fEventInfo = PTypeInfo(TypeInfo(T))) then
                        begin
                            // check if delivery is Sync because
                            // DispatchEvent has anonymous methods setup
                            // that is unnecessary for synchronous execution path
                            if aDelivery = Sync then
                                begin
                                    if Sub.BeginWork then
                                        try
                                            TNxEventMethod<T>(Sub.fEventMethod)(aEvent);
                                        finally
                                            Sub.EndWork;
                                        end;
                                end
                            else
                                DispatchEvent<T>(aEvent, Sub, aDelivery, Sub.fEventMethod);
                        end;
                end;
    finally
        fLock.EndRead;
    end;
end;
```

```
{ NxHorizon }

class constructor NxHorizon.ClassCreate;
begin
  fInstance := TNxHorizon.Create;
end;

class destructor NxHorizon.ClassDestroy;
begin
  fInstance.Free;
end;

end.
```

35.3 Back-pressure

Back-pressure occurs when you have fast producers and slow consumers—in other words, when producers generate more events than consumers can handle in a particular time frame. While occasional smaller peaks can be tolerated, extreme peaks or prolonged periods of inadequate processing speed can have a catastrophic impact.

There are several ways of dealing with back-pressure:

- **Controlling the producer** - Control the rate of message generation to match what the consumer can handle. This approach is not always feasible, as you may not have a producer under your control.
- **Queue messages** - Queue incoming messages (this is already indirectly a part of event bus), but this approach works only if there are occasional spikes in incoming data, that can be successfully processed in a reasonable amount of time before the queue grows too large. Most basic implementations of event buses will use queues outside its control - like the main thread event queue for `TThread.Queue` calls, and the task queue for pending `TTask` work items. If those queues are overwhelmed, performance will start to suffer, and eventually they can crash when they fail to allocate the memory required.
- **Drop messages** - Dropping messages if there are too many can also solve the problem, but this approach is only feasible for non-critical messages.

Back-pressure will be the breaking point for an ordinary event bus. A more sophisticated event bus can implement additional event queue, under its control, along with a mechanism for dropping particular messages and signalling contention to the producers.

In case you are wondering how an additional queue can solve the problem of overwhelming the existing queues: The basic queue implementation is backed by a dynamic array that needs to

allocate all required memory in a single consecutive memory block. If there is no available block of the required size, allocation will fail even if there is otherwise enough available memory on the system.

Adapting the queue to use multiple arrays in a linked list, instead of using a single array that may grow too large, reduces the memory allocation strain in peak pressure, and allows the queue to expand using significantly smaller chunks of memory.

Chapter 36

Measuring performance

Multithreading does not make your code run faster, it makes your code run slower.

If you are just moving long-running operations to a background thread to avoid a non-responsive application, then gaining extra performance from better utilization of available hardware resources is probably not your main concern. On the other hand, if you are hoping to reduce the time needed to complete some operations by running them in parallel, then measuring performance and knowing whether you have accomplished your goal or not will be critical. And sometimes, even when performance is not your final goal, it makes sense to measure, especially when you are using code constructs or frameworks you haven't used before.

Measuring is also important for observing the behavior of some code under stress. Thread contention and other forms of resource starvation can bring your application down to a halt, and in such cases, performance issues will become a priority. Knowing which parts of your code experience the most serious slowdown allows you to locate and fix issues faster.

There are two kinds of performance measurement: profiling and benchmarking. Profiling tells you *where* your application or its parts spends most of the time, and benchmarking will tell you *how fast* some code runs.

This chapter is intended as a brief introduction to measuring code performance, and a signpost directing you to required tools. It also covers some less-known aspects and pitfalls of measuring performance in multithreaded scenarios.

Note: When measuring code performance, it is extremely important that you don't use `Sleep` for simulating CPU work, because sleeping threads don't consume resources, and you will not get even vaguely accurate results when measuring such code. You will need to write appropriate dummy code that actually does something in order to measure properly.

36.1 Profiling

Performance profiling tools belong to a much larger group of profiling tools that give us a variety of information on the application's behavior and resource usage, which can also give us additional hints about where and how we can improve performance. For instance, using too many resources can have a serious impact on the performance. At the end (or start) of the day, excessive memory usage and memory leaks also count. But as our primary interest in multithreading, beyond thread safety, is to improve performance, measuring speed is of most interest here.

Performance profiling tools fall into two categories: instrumenting profilers and sampling profilers.

Instrumenting profilers add—inject—instructions into our code in order to collect the data and measure time, and that additional code significantly impacts the performance of the measured code. In the case of multithreaded code, it can also have a significant impact on behavior. Because of that, instrumenting profilers are not the best tool for accurate analysis of multithreaded applications, even though they can also give us valuable hints where to look and which parts of the code would benefit the most from the optimization.

Sampling profilers, on the other hand, observe and probe the application from the outside, without altering original code. That way, the impact on the application's speed and behavior is minimal, and the measured results are more accurate in terms of overall time and will give a better overview of potential bottlenecks. However, because they collect their information by probing, the data they collect in terms of individual functions and pieces of code called will only be an approximation.

There are numerous profiling tools for Delphi, some free, some commercial, but a very good place to start is the free *Delphi Tools Sampling Profiler* <https://www.delphitools.info/samplingprofiler/> for the Windows platform.

It is important to remember that other applications and processes will also have an impact on measuring performance, because they will compete with the measured application for hardware resources. To minimize that influence and get more accurate data, you should stop running all non-essential applications. On the other hand, if you are interested in improving and finding bottlenecks in a real-life scenario, when other applications and processes interfere with your own, you will want to measure and collect data in such scenarios, too.

36.2 Benchmarking

The term “benchmark” is often used in the context of measuring hardware performance, where you run a set of prebuilt tests to measure and compare different hardware systems or their individual components. In terms of code benchmarking, you are not measuring the performance of particular hardware, but rather some code. However, that process also involves measuring that same code across different systems, as different hardware configurations can have an impact on when and where some bottlenecks occur. This is especially true for multithreaded applications,

where the hardware configuration, like the number of CPU cores, has significant impact not only on speed, but also on behavior.

Not only do different systems behave differently, but our code also behaves differently under different circumstances and loads. That is why it is important that you don't limit your testing only to a single system or small data samples, regardless of whether you are profiling or benchmarking your code. While the results of such tests will give you some place to start, it is very likely that they will not expose all the issues, or that optimizing performance for such a narrow dataset will actually make your application perform worse in some other common scenarios.

The simplest approach to benchmarking code is by timestamping: retrieving the current time before and after the measured piece of code, and calculating the difference. In Delphi, that can be done by using the `TStopwatch` record from the `System.Diagnostics` unit. You can get the elapsed time in several forms: ticks (100 nanoseconds), milliseconds, as well as `TTimeSpan`, which allows you to retrieve the value in other time units.

```
uses
  System.Diagnostics;

procedure Measure;
var
  sw: TStopwatch;
begin
  sw := TStopwatch.StartNew;

  // measured code

  sw.Stop;
  Writeln(sw.ElapsedTicks);
  Writeln(sw.ElapsedMilliseconds);
end;
```

There are several issues with this kind of performance measurement. The first, fairly obvious one is that you cannot accurately measure very small parts of code—you can circumvent that problem by timing several iterations instead of one, when possible. Another larger issue is that you will be measuring real time elapsed, and with that, you will not be measuring just the performance of your code, but of the whole system, where other processes running will skew those results.

Repeatability is an extremely important part of benchmarking, and if you are getting vastly different results for repeated runs, then you definitely cannot use such results to perform any meaningful assessment nor optimizations.

Benchmarking libraries can help with solving some of the above issues, as well as help with collecting and comparing the results.

One such library is the *Delphi port of Google Benchmark* <https://github.com/spring4d/>

benchmark, and a useful addition for visual presentation of those results is the *Google Benchmark Viewer* <https://github.com/davidbernedo/GoogleBenchmarkViewer>

The basic approach in benchmark libraries is similar to using **TStopwatch**, but with some significant differences. Instead of measuring only real time, benchmark libraries will also measure process or thread time, giving more accurate results that involve only the measured code, not the whole system. The activity of other applications and processes running can still have an impact, and only a minimal number of those should be kept running while measuring performance.

Besides more specific time measuring, benchmarking libraries offer a variety of other features that allow us to get more precise data. When it comes to measuring small amounts of fast-running code, it is important to run such code more than once to get more statistically stable results. At runtime, Google Benchmark measures the time needed to run the code, and dynamically adjusts the number of iterations. There is also an option to manually specify the number of iterations. Another important feature is repeating the tests and reporting the mean, median, and standard deviation, as well as other, custom statistics.

Sometimes, you may want to measure code that is too complex to be measured with the help of a benchmarking library, as it cannot be conveniently extracted into a code snippet, or is intertwined with the UI. In such situations, you can use the **TStopwatch** approach, and measure the execution of some code no matter how broad its scope is. The one thing you need to pay attention to in such a scenario is that the code needs to run automatically, without human interaction or you will not be able to time it properly, unless it takes a significantly large amount of time to execute, where a few seconds will be an allowable variance.

Since **TStopwatch** measures only real time, you will need a more sophisticated stopwatch to measure thread or process time you. Such a stopwatch is not very complicated to implement, but there are some gotchas in accurate measuring of thread and process time on the Windows platform that also impact benchmarking libraries.

Namely, the API functions **GetProcessTimes** and **GetThreadTimes** used to retrieve those times can give wrong results under some circumstances, depending on the code you are measuring. If the thread you are measuring does not fully consume its time slice—for instance, if it gets interrupted by a higher-priority thread or goes into a wait state—it is possible that it will not be correctly measured. In other words, you can get zero time spent when measuring a thread that has been running for seconds.

Using the **TNxChronometer** class that is implemented further in this chapter, we can create a code example where measuring thread time using **GetThreadTimes** is not accurate. While this thread is merely sleeping in a loop, adding some other small operation can yield similar results. Even if you don't get zero time, you may get significantly incorrect timing for code doing busy waiting—which can lead you to inaccurate conclusions about potential issues and bottlenecks in some very time-sensitive code.

```
procedure Measure;
var
  Thread: TThread;
begin
  Thread := TThread.CreateAnonymousThread(
    procedure
    var
      tsr, tsc, ts: TNxChronometer;
      i: Integer;
    begin
      tsr := TNxChronometer.Start(CalendarTime);
      tsc := TNxChronometer.Start(ThreadCycles);
      ts := TNxChronometer.Start(ThreadTime);
      for i := 0 to 1000 do
        Sleep(1);
      ts.Stop;
      tsc.Stop;
      tsr.Stop;
      Writeln('Real time: ', tsr.ElapsedMs);
      Writeln('Thread time: ', ts.Elapsed);
      Writeln('Cycles: ', tsc.Elapsed);
    end);
  Thread.FreeOnTerminate := False;
  Thread.Start;
  Thread.WaitFor;
  Thread.Free;
end;
```

Output (values may vary depending on your system):

```
Real time: 15563
Thread time: 0
Cycles: 48778095
```

To get more accurate measurement in such situations, as well as measure differences in very fast-running code that requires greater accuracy, `QueryProcessCycleTime` and `QueryThreadCycleTime` can be used, and they will give accurate results. The main problem with those two functions is that they return the number of CPU clock cycles, and it is impossible to accurately convert that data to real time due to differences in the implementation of timing services by different CPUs. So you cannot use cycles to compare performance between different systems.

Because the results of those two functions cannot be converted to real time, benchmarking libraries cannot use them. If there is significant difference in system time and CPU time, it is very likely that measured code is not correctly timed or another currently running process is

skewing the results.

The following is an implementation of a cross-platform stopwatch with similar functionality to `TStopwatch`, but it allows measuring different times: real time, process time, and thread time, including highly accurate process cycles and thread cycles. If you are measuring cycles, you will only be able to get a raw `Elapsed` value that cannot be converted to nanoseconds or milliseconds.

```
unit NX.Cronos;

interface

uses
{$IFDEF MSWINDOWS}
  Winapi.Windows,
{$ELSE}
  Posix.Base,
  Posix.Time,
{$IFDEF ANDROID}
  Androidapi.JNI.JavaTypes,
  Androidapi.Helpers,
  Androidapi.Log,
{$ENDIF}
{$IFDEF IOS}
  Macapi.Mach,
  Macapi.Helpers,
  Macapi.ObjectiveC,
  iOSapi.Foundation,
{$ENDIF}
{$IFDEF OSX}
  Macapi.Mach,
  Macapi.Helpers,
  Macapi.ObjectiveC,
  Macapi.Foundation,
{$ENDIF}
{$ENDIF}
  System.SysUtils,
  System.Classes;

type
  TNxChronoMode =
    (CalendarTime, ProcessTime, ThreadTime, ProcessCycles, ThreadCycles);

  TNxChronometer = record
  private
    fMode: TNxChronoMode;
```

```
// accumulated time
fElapsed: UInt64;
// current time stamp - if 0 nothing is being measured
fStartTimeStamp: UInt64;
function CurrentTimeStamp: UInt64;
function GetElapsedMs: UInt64;
function GetElapsedNs: UInt64;
public
  constructor Create(aMode: TNxChronoMode);
  constructor Start(aMode: TNxChronoMode); overload;
  procedure Start; overload;
  procedure Stop;
  procedure Clear;
  property Elapsed: UInt64 read fElapsed;
  property ElapsedNs: UInt64 read GetElapsedNs;
  property ElapsedMs: UInt64 read GetElapsedMs;
end;

implementation

// ***** Time measuring APIs *****

{$IFDEF MACOS}
type
  clockid_t = clock_res_t;

function clock_gettime_nsec_np(clock_id: clockid_t): uint64_t; cdecl;
  external libc name _PU + 'clock_gettime_nsec_np';
{$EXTERNALSYM clock_gettime_nsec_np}

function clock_gettime(clk_id: clockid_t; ts: Ptimespec): Integer; cdecl;
  external libc name _PU + 'clock_gettime';
{$EXTERNALSYM clock_gettime}

const
  CLOCK_REALTIME = 0;
  CLOCK_MONOTONIC_RAW = 4;
  CLOCK_MONOTONIC_RAW_APPROX = 5;
  CLOCK_MONOTONIC = 6;
  CLOCK_UPTIME_RAW = 8;
  CLOCK_UPTIME_RAW_APPROX = 9;
  CLOCK_PROCESS_CPUTIME_ID = 12;
  CLOCK_THREAD_CPUTIME_ID = 16;
```

```
NSEC_PER_USEC = 1000;           // nanoseconds per microsecond
USEC_PER_SEC  = 1000000;         // microseconds per second
NSEC_PER_SEC  = 1000000000;      // nanoseconds per second
NSEC_PER_MSEC = 1000000;         // nanoseconds per millisecond
{$ENDIF}

{$IFDEF MSWINDOWS}
function GetCalendarTimeStamp: UInt64;
begin
  Result := GetTickCount * UInt64(10000);
end;

function GetProcessTimeStamp: UInt64;
var
  lpCreationTime, lpExitTime, lpKernelTime, lpUserTime: TFileTime;
  ts: ULARGE_INTEGER;
begin
  Result := 0;
  if GetProcessTimes(GetCurrentProcess, lpCreationTime, lpExitTime,
    lpKernelTime, lpUserTime) then
  begin
    ts.HighPart := lpKernelTime.dwHighDateTime;
    ts.LowPart := lpKernelTime.dwLowDateTime;
    Result := ts.QuadPart;
    ts.HighPart := lpUserTime.dwHighDateTime;
    ts.LowPart := lpUserTime.dwLowDateTime;
    Result := Result + ts.QuadPart;
  end;
end;

function GetThreadTimeStamp: UInt64;
var
  lpCreationTime, lpExitTime, lpKernelTime, lpUserTime: TFileTime;
  ts: ULARGE_INTEGER;
begin
  Result := 0;
  if GetThreadTimes(TThread.CurrentThread.Handle, lpCreationTime, lpExitTime,
    lpKernelTime, lpUserTime) then
  begin
    ts.HighPart := lpKernelTime.dwHighDateTime;
    ts.LowPart := lpKernelTime.dwLowDateTime;
    Result := ts.QuadPart;
    ts.HighPart := lpUserTime.dwHighDateTime;
    ts.LowPart := lpUserTime.dwLowDateTime;
```

```
    Result := Result + ts.QuadPart;
  end;
end;

function GetProcessCycles: UInt64;
begin
  if not QueryProcessCycleTime(GetCurrentProcess, Result) then
    Result := 0;
end;

function GetThreadCycles: UInt64;
begin
  if not QueryThreadCycleTime(TThread.CurrentThread.Handle, Result) then
    Result := 0;
end;

{$ELSE}
function GetCalendarTimeStamp: UInt64;
var
  ts: timespec;
begin
  Result := 0;
  if clock_gettime(CLOCK_MONOTONIC, @ts) = 0 then
    Result := (Int64(1000000000) * ts.tv_sec + ts.tv_nsec) div 100;
end;

function GetProcessTimeStamp: UInt64;
var
  ts: timespec;
begin
  Result := 0;
  if clock_gettime(CLOCK_PROCESS_CPUTIME_ID, @ts) = 0 then
    Result := (Int64(1000000000) * ts.tv_sec + ts.tv_nsec) div 100;
end;

function GetThreadTimeStamp: UInt64;
var
  ts: timespec;
begin
  Result := 0;
  if clock_gettime(CLOCK_THREAD_CPUTIME_ID, @ts) = 0 then
    Result := (Int64(1000000000) * ts.tv_sec + ts.tv_nsec) div 100;
end;
```

```
function GetProcessCycles: UInt64;
var
  ts: timespec;
begin
  Result := 0;
  if clock_gettime(CLOCK_PROCESS_CPUTIME_ID, @ts) = 0 then
    Result := (Int64(1000000000) * ts.tv_sec + ts.tv_nsec) div 100;
end;

function GetThreadCycles: UInt64;
var
  ts: timespec;
begin
  Result := 0;
  if clock_gettime(CLOCK_THREAD_CPUTIME_ID, @ts) = 0 then
    Result := (Int64(1000000000) * ts.tv_sec + ts.tv_nsec) div 100;
end;
{$ENDIF}

// ***** TNxChronometer *****

constructor TNxChronometer.Create(aMode: TNxChronoMode);
begin
  fMode := aMode;
  fElapsed := 0;
  fStartTimeStamp := 0;
end;

constructor TNxChronometer.Start(aMode: TNxChronoMode);
begin
  fMode := aMode;
  fElapsed := 0;
  fStartTimeStamp := CurrentTimeStamp;
end;

procedure TNxChronometer.Start;
begin
  fStartTimeStamp := CurrentTimeStamp;
end;

procedure TNxChronometer.Stop;
var
  Current: UInt64;
begin
```

```
if fStartTimeStamp = 0 then
  Exit;
Current := CurrentTimeStamp - fStartTimeStamp;
fStartTimeStamp := 0;
if Current > 0 then
  fElapsed := fElapsed + Current;
end;

procedure TNxChronometer.Clear;
begin
  fStartTimeStamp := 0;
  fElapsed := 0;
end;

function TNxChronometer.CurrentTimeStamp: UInt64;
begin
  case fMode of
    CalendarTime : Result := GetCalendarTimeStamp;
    ProcessTime : Result := GetProcessTimeStamp;
    ThreadTime : Result := GetThreadTimeStamp;
    ProcessCycles : Result := GetProcessCycles;
    ThreadCycles : Result := GetThreadCycles;
    else Result := 0;
  end;
end;

function TNxChronometer.GetElapsedNs: UInt64;
begin
  Result := fElapsed * 100;
end;

function TNxChronometer.GetElapsedMs: UInt64;
begin
  Result := fElapsed div 10000;
end;

end.
```


Appendix

References

- Delphi Reference

http://docwiki.embarcadero.com/RADStudio/en/Delphi_Reference

- Delphi Language Guide

http://docwiki.embarcadero.com/RADStudio/en/Delphi_Language_Guide_Index

- Using Floating-Point Routines

https://docwiki.embarcadero.com/RADStudio/en/Using_Floating-Point_Routines

- David Heffernan - Rethinking Delphi's management of floating point control registers

[https://en.delphipraxis.net/topic/3099-rethinking-delphi%E2%80%99s-management-of-floatin](https://en.delphipraxis.net/topic/3099-rethinking-delphi%E2%80%99s-management-of-floating-point-control-registers/)

<https://stackoverflow.com/a/27125657>

- Raymond Chen - What are these dire multithreading consequences that the GetFullPathName documentation is trying to warn me about?

<https://devblogs.microsoft.com/oldnewthing/20210816-00/?p=105562>

<https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-getfullpathnamew>

- TComponent Notification

<http://docwiki.embarcadero.com/Libraries/en/System.Classes.TComponent.Notification>

- Order of unit initialization/finalization

[https://docwiki.embarcadero.com/RADStudio/en/Programs_and_Units_\(Delphi\)](https://docwiki.embarcadero.com/RADStudio/en/Programs_and_Units_(Delphi))

- Allen Bauer - Class Constructors. Popping the hood.

https://blog.therealoracleatdelphi.com/2009/09/class-constructors-popping-hood_3.html

- Document Object Model (DOM) Level 2 Core Specification

<https://www.w3.org/TR/DOM-Level-2-Core/>

- Indy Project Home Page

<https://www.indyproject.org/>

- PCRE

<https://pcre.org/pcre.txt>

- Multiple Threads and GDI Objects

<https://docs.microsoft.com/en-us/windows/win32/procthread/multiple-threads-and-gdi-objects>

- Security Considerations: GDI+

<https://docs.microsoft.com/en-us/windows/win32/gdiplus/sec-gdiplus>

- Vcl.Graphics.TCustomCanvas.Lock

<https://docwiki.embarcadero.com/Libraries/en/Vcl.Graphics.TCustomCanvas.Lock>

- Multi-Threading for TBitmap, TCanvas, and TContext3D

http://docwiki.embarcadero.com/RADStudio/en/Multi-Threading_for_TBitmap,_TCanvas,_and_TContext3D

- System.Classes.CheckSynchronize

<https://docwiki.embarcadero.com/Libraries/en/System.Classes.CheckSynchronize>

- How to: Set a Thread Name in Native Code

<https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-set-a-thread-name-in-native-code?view=vs-2019>

- Cancellation in Managed Threads

<https://docs.microsoft.com/en-us/dotnet/standard/threading/cancellation-in-managed-threads>

- IAsyncResult Interface

<https://docs.microsoft.com/en-us/dotnet/api/system.iasyncresult>

- Benchmark

[https://en.wikipedia.org/wiki/Benchmark_\(computing\)](https://en.wikipedia.org/wiki/Benchmark_(computing))

- Profiling

[https://en.wikipedia.org/wiki/Profiling_\(computer_programming\)](https://en.wikipedia.org/wiki/Profiling_(computer_programming))

- Raymond Chen - Is there anything better than GetThreadTimes for obtaining per-thread CPU usage information?

<https://devblogs.microsoft.com/oldnewthing/20161021-00/?p=94565>

- QueryThreadCycleTime function

<https://docs.microsoft.com/en-us/windows/win32/api/realtimapiset/nf-realtimapiset-querythreadcycletime>

- QueryProcessCycleTime function

<https://docs.microsoft.com/en-us/windows/win32/api/realtimapiset/nf-realtimapiset-queryprocesscycletime>

- GetThreadTimes function

<https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getthreadtimes>

- Delphi Tools Sampling Profiler

<https://www.delphitools.info/samplingprofiler/>

- Delphi port of Google Benchmark

<https://github.com/spring4d/benchmark>

- Google Benchmark Viewer

<https://github.com/davidbernedo/GoogleBenchmarkViewer>

Quality Portal Reports

- RSP-21105 - Altered SSE - MXCSR Default value

<https://quality.embarcadero.com/browse/RSP-21105>

- RSP-36674 - SysLocale is not thread-safe

<https://quality.embarcadero.com/browse/RSP-36674>

- RSP-38281 EnsurePoolToken in System.Rtti is not thread-safe

<https://quality.embarcadero.com/browse/RSP-38281>

- RSP-9815 - TRealPackage.FindType thread-unsafe

<https://quality.embarcadero.com/browse/RSP-9815>

- System.Rtti: TVirtualInterface (TMethodImplementation) is not thread-safe on ARM

<https://quality.embarcadero.com/browse/RSP-17897>

- XML Parsing is not thread-safe

<https://quality.embarcadero.com/browse/RSP-36814>

- TODOMIImplementationFactory in Xml.omnixmldom and Xml.adomxml dom are not thread-safe

<https://quality.embarcadero.com/browse/RSP-36819>

- PPL Task Continuation and Cancellation Token support

<https://quality.embarcadero.com/browse/RSP-13286>