Delphi EVENT-BASED AND ASYNCHRONOUS Programming

Dalija Prasnikar
Neven Prasnikar Jr.

Dalija Prasnikar

Neven Prasnikar Jr.

Delphi Event-based and Asynchronous Programming

Copyright ©2020 Dalija Prasnikar & Neven Prasnikar Jr.

First Edition

Contents

Introduction

Event-based programming is everywhere. Nowadays, you can hardly write any kind of application without leaning on events and messages.

This simple, yet extremely powerful mechanism is also the cornerstone of asynchronous and multithreaded programming. Without events, we would not know when some task was completed.

But, asynchronous and multithreaded programming consists of more than just handling multiple threads, protecting shared resources, and synchronization. It also includes designing and understanding program flow. That design aspect is often forgotten, taken for granted, and solving all the fine-grained nuances of multithreaded programming hogs the spotlight.

Without understanding asynchronous flow and the bigger picture it can be hard to properly solve all the other issues, including multithreading. What used to be plain spaghetti code, now becomes temporal spaghetti.

You can lose control over your objects, have bits of code creating and releasing them all over the place, even without adding multithreading into the mix. Memory leaks, crashes, and holding references to dead data long after its destruction are perfectly possible even in a singlethreaded environment.

Good program flow design is an important blueprint in application programming, just like architecture starts with designing whole building up front, and having a set of well-designed blueprints is the first step in having a good building.

Just like in architecture, a good blueprint is not all that is needed. How you build and which materials you use are also important. Doing both things well is necessary for having a good end result.

While the starting point of this book is giving a proper introduction to event-based and asynchronous programming flow and design, it also provides the necessary implementation details, explanations and code examples needed to put those designs into practice.

Acknowledgements

Huge thanks to William Meyer for providing the technical review, and Sandra Prasnikar for providing illustrations.

Code

The primary purpose of the code examples provided in this book is to explain certain concepts, show possible mistakes and bugs, and provide basic templates for implementing coding patterns.

There are many ways to write code that needs to perform some functionality. There are always some really bad or completely incorrect approaches that will be marked as such, but for the rest, the difference between good or bad code is in the eye of the beholder.

Using coding patterns you are familiar and confident enough with that you can easily follow the logic of the code and avoid mistakes is probably more important than always using some really stellar and optimized coding patterns.

That does not mean that you never need to improve your skills, but that you need to take your time and improve at your own pace.

To distinguish identifiers that might collide with existing Delphi classes prefix `Nx`—a more readable variant of `NX`—will be used. `NX`, because it will boldly take you where no man has gone before. Actually, it is `NX` because it is the next generation of my `X` library, which got named `X` because naming is hard…

Code examples are available for download from GitHub:

https://github.com/dalijap/code-delphi-async

Part 1. The Loop

Chapter 1

Hidden complexity

Every program has a beginning and an end. Quite literally for Delphi—and generally Pascal—programs. Every program has a main `begin`-`end` block. The execution of Delphi programs starts with the `begin` keyword and finishes with `end`.

The main `begin`-`end` block holds more than meets the eye. At the main `begin`, the initialization sections of all used units in the application will be executed, and before running the initialization section for each unit, all the class constructors of any used classes declared in that unit will run. Similarly, the main `end` is responsible for executing the finalization sections, and after each finalization section is executed, running the class destructors of all used classes in a particular unit.

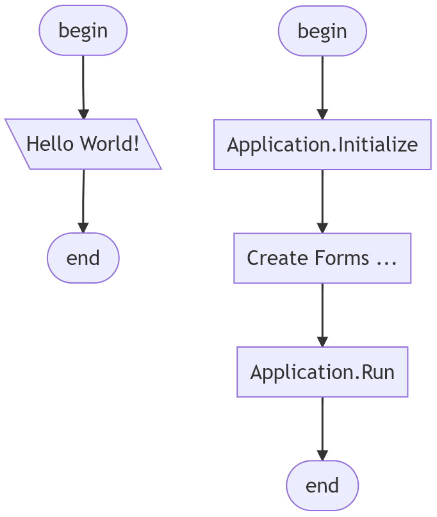
The classic *Hello World* console example code looks pretty straightforward:

```
program ConHelloWorld;

{$APPTYPE CONSOLE}

begin
  Writeln('Hello World!');
end.
```

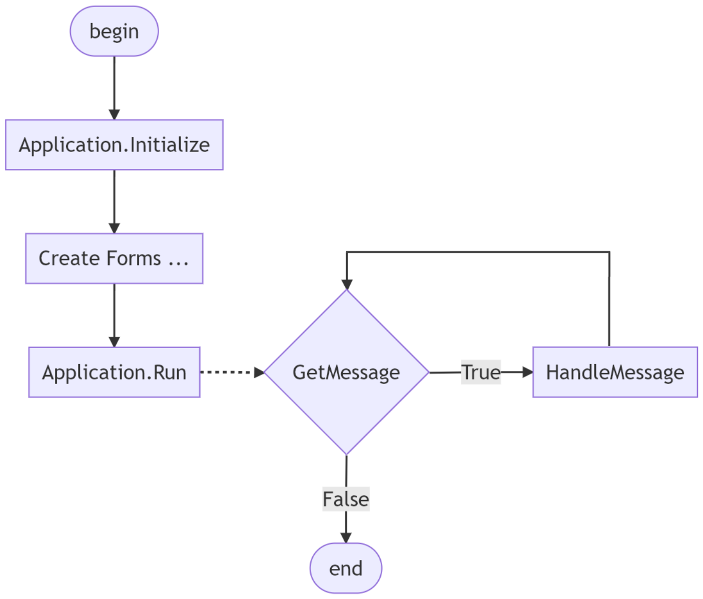
The basic Windows *Hello World* application code does not look too complex either. Just a few lines of code more than the console application has. Yes, we can expect that those few lines hide a more complex code, but the code still looks pretty linear:

```
program HelloWorld;

uses
  Vcl.Forms;

var
  MainForm: TForm;

begin
  Application.Initialize;
  Application.CreateForm(TForm, MainForm);
  MainForm.Caption := 'Hello World!';
  Application.Run;
end.
```

On the surface, their program flow diagrams also look rather similar:



But, in reality, while the code may look simple and the flow may seem linear, the actual program flow is more complex, with plenty of complexity hidden behind that single call to `Application.Run`.



That single call contains the central building block of all non-trivial applications, called the *application event loop*, sometimes also referred to as an *application loop*, *main loop*, or *main event loop*. Basically, everything the program does, every decision, every code path—except some initialization and finalization code—will run within that main control flow loop.

While I said that all non-trivial applications have an *application loop*, that is not entirely true. There are console applications containing code and functionality that is far from trivial, and don't have such a loop.

OK, you might say, so GUI applications have an *application loop*, while console applications don't. While you are on the right track, this is not quite true when it comes to console applications, as they can also have a loop, even though most of them do not.

Chapter 2

Peeking at the console

Console applications that don't have a main loop are usually programmed to perform some specific task, and after the task is finished, such an application will also finish running.

During execution, the console application can give feedback to the user about its progress, or output its final results, but that depends solely on the application and its purpose. Some even have an option called *silent mode* where, depending on specific input, the application can skip printing any feedback back to the console and optionally write it to some log file, or use some other means of leaving the feedback or results.

When it comes to input, some console programs don't require any kind of input, as they are designed to perform some fixed task. Some programs will take input in the form of parameters passed through the command line. Others will ask for and read all necessary input immediately after starting, and when all the necessary data is entered, they will run the required task.

The following two are examples of console programs that calculate the sum of two user-provided integers. The first one takes the numbers as command line parameters, and the second one will wait for the user to type input into the console:

```
program ConSumParams;

{$APPTYPE CONSOLE}

uses
  System.SysUtils;

var
  X, Y, Sum: Integer;
begin
  X := StrToIntDef(ParamStr(1), 0);
  Y := StrToIntDef(ParamStr(2), 0);
  Sum := X + Y;
  Writeln('Sum: ', Sum);
end.
```

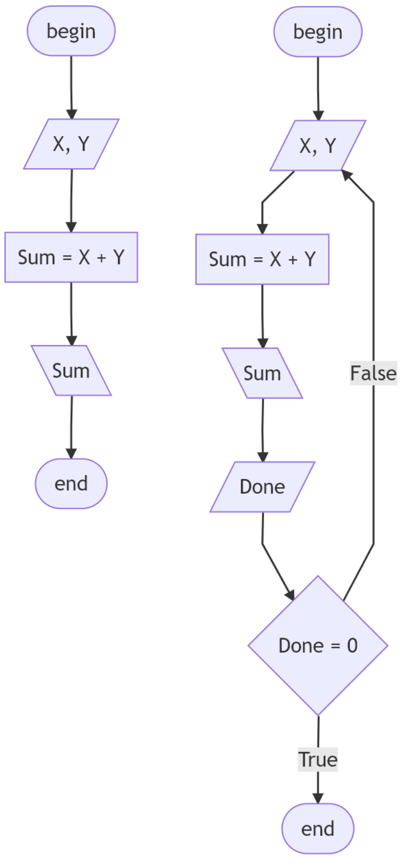
```
program ConSumRead;

{$APPTYPE CONSOLE}

uses
  System.SysUtils;

var
  X, Y, Sum: Integer;
begin
  Write('X: ');
  Readln(X);
  Write('Y: ');
  Readln(Y);
  Sum := X + Y;
  Writeln('Sum: ', Sum);
end.
```

It is pretty obvious that the above programs don't have any kind of loop at all, much less a main program flow loop. More complex console programs, will have a very similar, very linear program flow logic on the highest level: collect input, do the task, output results and exit. (Please note, having some piece of code that loops does not mean that such a program has a main program flow loop. The purpose of the loop is what matters here.)

On the other hand, if you change that program slightly and add a loop that allows the user to repeatedly enter number sequences, you will create an application that does have a *main loop*, however rudimentary it might be:

```
program ConSumLoop;

{$APPTYPE CONSOLE}

uses
  System.SysUtils;


var
  X, Y, Sum: Integer;
  Done: Char;
begin
```

```
  repeat
    Write('X: ');
    Readln(X);
    Write('Y: ');
    Readln(Y);
    Sum := X + Y;
    Writeln('Sum: ', Sum);
    Writeln('Enter 0 to exit, enter to continue');
    Readln(Done);
  until Done = '0';
end.
```



If we compare both program flow diagrams, the difference becomes more obvious. The first program has a plain, linear flow, and the second one has a distinct *main loop* logic implemented.

Chapter 3

Main event loop

Until now, I have been using the terms *main loop* and *main event loop* interchangeably. Even though this is how this terminology is generally used, I have slightly broadened the initial definition of *main event loop*.

An *event loop* is a program flow pattern that waits in a loop for events or messages, and dispatches those messages to appropriate event handlers in the application. This pattern is also called a *message loop*, *run loop*, or *message pump*. This whole process that happens in the *event loop*, retrieving and processing messages from the message queue, is often called *pumping messages*.

The *main event loop* is the central, main, event dispatch system in the application, that represents the highest level of flow control in the application. While applications can have more than one event loop, only one can be the *main event loop*.

In the previous `ConSumLoop` example with the `repeat...until` loop, we don't have a *main event loop* because there is no event dispatching mechanism involved, but we do have a *main loop* in terms of a loop that is on the highest level of flow control in the application. Such basic loop patterns are also known as *menu-driven design*.

While *menu-driven* applications, and console applications without any loop, are still used for various utilities and purposes, they have been dethroned with the appearance of GUI operating systems and *event-driven architecture*.

On the other hand, our Windows *Hello World* example with the `Application.Run` call represents an actual event-based application, that contains a real *main event loop*.

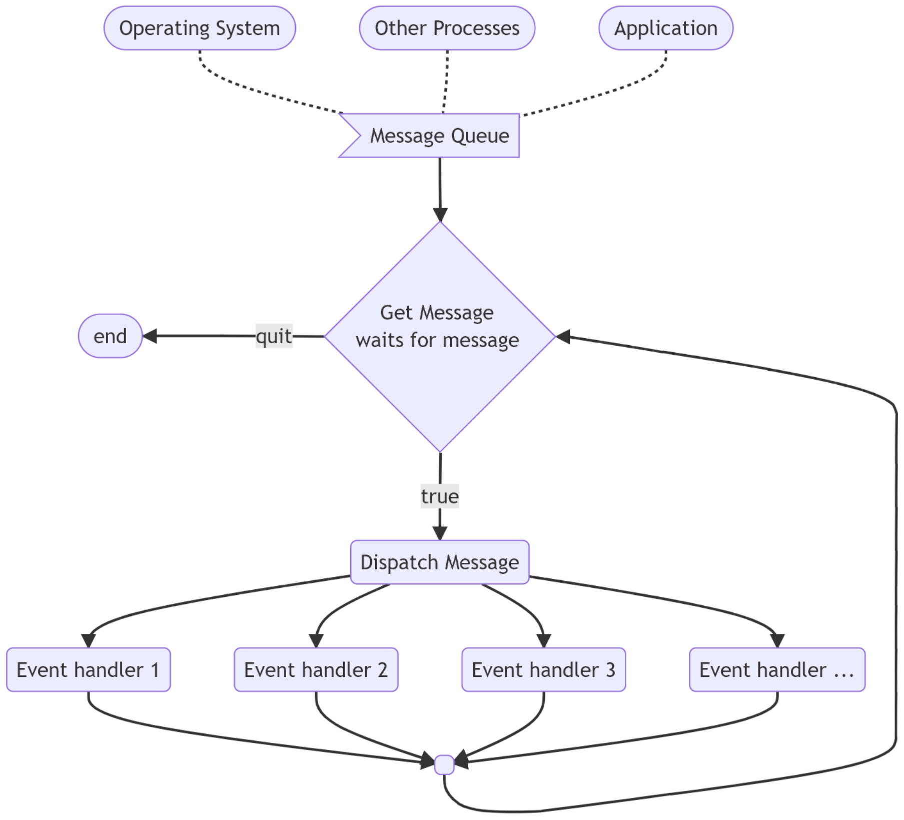
The *main event loop* architecture is tightly coupled with the operating system on which the application is running. This is not a surprising fact, considering that applications have to interact with the OS to receive input and send output to various hardware devices and other processes and applications in the system, and event loops represent a core communication channel.

Every operating system has its own slightly different architecture and slightly different behavior, but despite the differences, they all share some common building blocks. Once you understand the basic *event loop* pattern on one OS, understanding *event loops* on different operating systems will be rather trivial—and mostly focusing on the implementation details of a commonly shared *event-based* design.

The basic building blocks of any *event-based* system are:

- Loop
- Message queue
- Mechanism for retrieving and sending messages to the queue
- Mechanism for dispatching and handling messages

Some operating systems have those responsibilities encapsulated in classes with distinct and separate roles. Others have them exposed through simple API functions, where the loop itself may be left to the application developer to implement.



The basic mechanics of an *event loop*, regardless of the platform, are the following:

Messages arrive in a dedicated loop message queue. Messages can be sent by the OS (hardware), other processes (software), and from within the application itself. Filling the queue happens outside of the loop's logic.

The most basic loop logic is rather simple, even though actual loops will have more refined and more complex implementations. The following example will use Windows API terminology. On other platforms, the equivalent API calls might have different names.

The code will perform a blocking call to `GetMessage`, that will retrieve a message from the message queue if there is one. If there isn't, it will suspend the calling thread until the next message arrives and wakes up the thread. If the received message is not a *quit* message (in Windows terminology, `WM_QUIT`) that will tell us to break the application loop and finish running, we will call a message handler that will dispatch the message to the appropriate message handler, and the handler will then execute its code. All of this executes synchronously in the same thread. Handlers can freely launch additional threads for running particular code, but the loop itself, and the process of triggering event handlers, happens in the context of a single thread.

After the event handler finishes its execution, the loop starts from the beginning, waiting for or retrieving the next message in the queue.

A significant part in the described *event loop* is suspending the thread if there are no messages for processing. Without that, the thread would keep spinning the loop at full speed, burning CPU cycles in vain, which wastes energy and prevents other applications and processes from using those hardware resources to do some useful work.

Another combination that allows a bit more complex logic in the loop, and is more commonly used in real-life scenarios, is using `PeekMessage` and `WaitMessage`. Unlike `GetMessage`, which will wait if there are no messages in the queue, `PeekMessage` will return immediately, giving us a chance to do additional work if message queue is empty. In other words, to detect the moment before the application will go into an idle state and give us the chance to act on that information, and then we can call `WaitMessage` to suspend the current thread until the next message arrives.

Of course, we can also choose not to put the application to sleep, and omit the call to `WaitMessage`, but that is not something a general-purpose application should do.

Chapter 4

Event-based programming

When you create a new Delphi application, either VCL- or FMX-based, drop the button on the form, double-click on the button, and attach an `OnClick` event handler on it, you are basically doing *event-based* or *event-driven* programming.

That *main event loop* process explained in the previous chapter will act upon user interaction with your application, and when the user does click on your button, the OS will generate an appropriate message, send it to your application's main message queue, and through message dispatching and handling, eventually that message will arrive to your `OnClick` event handler, and the code inside will be executed accordingly.

All the complexity and fine detail of event dispatching have been hidden away by the VCL or FMX frameworks, and you can focus on the work you need to do instead of fiddling with low-level application architecture.

Console applications, on the other hand, don't have a message loop on their own, and if you want or need to interact with the OS through *official* messaging channels, you will have to write your own message handling and directly call the appropriate OS APIs.

Event-based programming is the backbone of modern software, from operating systems to desktop applications, mobile applications, games...

It has extremely simple and easily understandable logic behind it, and yet it is powerful and versatile and allows building the most complex systems around it.

Delphi originated on Windows, and its original event handling architecture is built on top of the Windows messaging system. While you can create complex Windows VCL applications without having to go deeper into how Windows event dispatching and handling works, if you ever need to tweak some behavior or add more complex interactions with Windows, you will find that you can do all that in Delphi with ease and you can start without being a Windows programming expert.

Cross-platform application development obviously cannot rely on the Windows messaging system, and some built-in language and VCL features that directly support Windows cannot be used for cross-platform development.

But, just like Windows, other operating systems rely on similar event-based mechanisms. While Delphi applications allow you access to any OS API and give you the ability to directly communicate with the OS, doing that in a cross-platform manner would mean having separate code for every platform.

Delphi is known for its RAD capabilities and ease of development, where you don't need to go to low levels in order to write functional applications, and this is exactly what the FMX framework provides. It wraps up all platform-specific interactions and you can write FMX applications that will run on various platforms without having to write separate code for separate platforms.

That does not mean you cannot go deep and provide platform-specific code when you need to, but that most common needs are solved inside FMX itself, and that both VCL and FMX share a common core for handling events. Dropping a button on a form and attaching an `OnClick` event works the same in both.

However, you may still need to send and receive messages in cross-platform applications, and without having the Windows messaging system you could use in such situations, some other mean of communication is needed for those. FMX provides a cross-platform messaging system in `System.Messaging` that can be used in cross-platform scenarios, without the need to handle events on the OS level.

Of course, the additional abstraction layer will make it harder to use the Windows messaging system in Windows-based FMX applications, but that is the price you need to pay for cross-platform functionality. Again, you can still use the Windows native messaging system in Windows FMX applications, but it will be a bit more complicated than using the same in VCL. The same goes for interacting with other operating systems. It will not be as easy as interacting with Windows through the VCL, and will require more code to achieve some functionality.

If you need to interact with any OS, and that includes Windows through VCL, you will have to learn something about particular OS APIs and read their documentation. And just like many Delphi developers learned how to interact with Windows by reading the Microsoft API documentation and reading VCL code to apply common patterns, learning how to interact with other operating systems, besides reading documentation, will also include finding the appropriate coding patterns in FMX code.

Chapter 5

Pushmi-pullyu

In computing, there's two ways of handling events, analogous to the ways one might operate a post office. One option is for you to periodically go to the post office and check if your mail's arrived. This is called **polling**, where a program keeps checking the state of a variable every once in a while:

```
procedure TPerson.PollMail;
begin
  GotoPostOffice;
  if MailHasArrived then
      begin
        GetMail;
            ReadMail;
        end;
end;
```

The other option is for the post office to hand your mail over to a mailman once it arrives. To stretch the analogy a little, let's say the mailman delivers it to you personally, instead of requiring you to poll your mailbox. This is called **pushing**, where changes to the variable state are immediately pushed to the relevant location(s) for processing. More experienced programmers may recognize this principle as the centerpiece of the *Observer pattern*, only viewed from a different perspective.

```
procedure TPostOffice.PushMail;
var
  Receiver: TPerson;
begin
  if MailHasArrived then
    begin
      Receiver := GetMail;
      PostMan.DeliverMail(Receiver);
            Receiver.ReadMail;
      end;
end;
```

Of course, the post office does not operate only for one person, so it needs to process all mail available at the post office at some point:

```
procedure TPostOffice.PushMail;
var
  Receiver: TPerson;
begin
  while MailHasArrived do
    begin
      Receiver := GetMail;
      PostMan.DeliverMail(Receiver);
            Receiver.ReadMail;
      end;
end;
```

If `PushMail` reminds you of an *event loop*, there is a good reason for it. It is basically the same pattern. While there is *something* in *storage*, you take one piece at a time and *dispatch* it to the *receiver*. If, at some moment, there is nothing in storage, the *dispatcher* can take a break.

Besides the *push* model, using a `GetMessage` *event loop* also allows you to use the *poll* model with `PeekMessage`. Both push or pull patterns used in the *main event loop* can be useful in other situations across the application.

Each approach has its pros and cons. On one hand, polling doesn't require the oh-so-busy mailman to finally get around to delivering *your* mail... but it does require you to keep making constant trips to the post office. Between going to the post office and doing all your other daily stuff, you're using up *that* much more of your day. You might end up trying to poll the post office more often than you can afford, preventing you from doing any of the other things you planned for the day... or getting any idle time to rest. Translating back to a computing context: You use up more processor time, depending on how long the polling operation takes, and how often you try to do it. In extreme cases, you might starve other legitimate operations because you're too busy polling.

Pushing, on the other hand, doesn't take up a lot of your time, but might cause much larger delays between the mail reaching the post office and being delivered to you. Maybe there's too many houses for the mailman to deliver to (lots of objects to push to). Maybe the houses are too far apart (takes too long to push, for example, if `DeliverMail` involves a long, synchronous operation, like synchronous disk I/O). Maybe people even get too much mail for the poor mailman, so he can't deliver it all in one day. Either way, the mail is delayed, and in particularly severe cases, it might become irrelevant by the time it gets there.

As you can see, both approaches are faster in some contexts, and slower in others. Multithreading and other tricks can mitigate or even prevent slowdowns with either option, but in the end, what you choose should depend on how many resources you expect your application to have, and how often you receive new events:

- Pushing works best if you don't expect the object to have to push a lot of messages (or to a lot of observers) at a time, and are trying to conserve the observers' resources.
- Conversely, polling is quite effective if you have a lot of messages or observers, and want each of them to receive the message as soon as possible.

5.1 When does pushing happen?

Pushing is often used on a hardware level—where it is typically referred to as **interrupting**—to conserve resources. For instance, most modern systems do not constantly poll your keyboard, and most applications probably don't redraw their user interface unless something happened to warrant a redraw. For instance, the VCL and FMX frameworks only redraw UI elements when it is deemed necessary.

The *Observer pattern* is based on pushing, where one object—an *observable*—will iterate on any *observers* interested in some event, and notify them when it happens. But even when you do pushing in an *observer pattern*, you may still have a situation where, deep in the observer implementation itself, you might need to do some polling to get the actual data.

5.2 What about polling?

Nowadays, most common libraries and frameworks abstract polling away under a pushing facade, if it occurs at all. However, there are some cases where polling remains fairly common and relatively obvious. For example, TCP/IP uses heartbeat packets to make sure a connection is still alive.

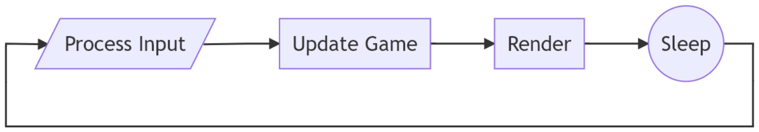
Chapter 6

Game loop

A prominent example of polling can often be seen in video games. These are the most likely to require the developer to explicitly define an event loop, or in this case, a game loop, which polls user input, calculates changes, and redraws the game world.

Video games are like a movie: Changes happen all the time regardless of the player's actions, and games need to redraw these changes at some minimal framerate to animate them smoothly. Going too fast, at faster rates than the hardware can refresh the screen, will just waste power. Depending on the computer's speed and the complexity of the calculations, the game might sleep for a short while at the end of each rendering cycle, to reduce power consumption and maintain a steady framerate.

If the calculations take too long, the game may stutter or run at a lower framerate altogether. In such cases, games can either reduce graphics or simulation quality, dumb down the calculations, but they can also implement more sophisticated calculation logic, reducing the frequency with which the most complicated calculations are performed.



```
procedure GameLoop;
var
  StartTime: TDateTime;
  Elapsed: Int64;
begin
  while True do
    begin
      StartTime := Now;
      ProcessInput;
      UpdateGame;
      RenderGame;
      Elapsed := MilliSecondsBetween(Now, StartTime);
      if Elapsed < FPSTime then
        Sleep(FPSTime - Elapsed);
    end;
end;
```

The above code is just an example of how the core logic of a game loop might look, not a working example you can actually use in a real life scenario. For starters, if you want to run the game at 60 FPS, your frame duration will be 16.7 milliseconds. Sleeping for such small periods of time is not precise enough.

Creating game loops and gaming frameworks is a topic complex enough to warrant its own book. The purpose of this chapter is to give an overview of another type of *event loop*, a more active one that does not sleep and wait to be pushed, but actively polls for information and runs at full speed. It is not meant to be an introduction to "How to write your own game engine".

If you are into writing games, then you would probably want to start with an existing game engine, instead of writing your own. It is not that writing game engines is not fun, but if you actually want to release the game before the end of the century, then using an existing one is a better course of action.

Most—if not all—commercial game engines, such as Unity or Unreal, bury the game loop behind several layers of abstraction stacked on top of each other, but at their core, the loop is still there. However, game development frameworks such as libGDX or SpriteKit still require you to implement your own game loop on a fairly low level.

A simplified game loop for a top-down arcade shooter might look something like this:

```
procedure Render(DeltaTime: Double);
var
  Timer: Double;
begin
  // Asteroids go down over time
  AsteroidTick(DeltaTime);

  // Alien ships go down too...
  // but they also maneuver sideways and shoot at you!
  AlienTick(DeltaTime);

  // Bullets fly up or down, depending on who fired them
  BulletTick(DeltaTime);

  if LeftKeyPressed then // Move the player's ship left
    MovePlayerLeft(DeltaTime)
  else
  if RightKeyPressed then // Move the player's ship right, instead
    MovePlayerRight(DeltaTime);

  if AttackKeyPressed then // The player wants to shoot.
    begin
      // How many times could we have fired since the last rendering pass?
      Timer := TimeSinceLastShot + DeltaTime;
      while (Timer > DELAY_BETWEEN_SHOTS) do
        begin
          // Let's make the new bullet travel based on how long it's been
          // since we're supposed to have fired it. Also, we're
          // the player, so the bullet goes up.
          FireBullet(Timer, True);
          Timer := Timer - DELAY_BETWEEN_SHOTS;
        end;
      TimeSinceLastShot := Timer;
    end;

  DrawBackground;
  DrawAsteroids;
  DrawAliens;
  DrawBullets;
  DrawPlayer;
end;
```

While your code editor does not care how long it takes you to type in some code, the game needs to track the passage of time.

Delta time

So, what is this `DeltaTime` parameter we keep throwing around?

The amount of time it takes an event loop to complete is dependent on a *lot* of factors. The processor speed, the amount of stuff that needs to happen in a given tick... But this sort of variance is not always acceptable. If our player can only fire his laser cannon twice per second, then that means the interval between laser shots should be no less than 0.5 seconds. It can't be 2 rendering cycles, it can't be 100. It has to be as many rendering cycles as you can fit into 500 milliseconds—no more, no less.

The trick is, you need some way of communicating how much time has passed to the game loop. Sure, one option would be to set a timer to fire once every millisecond, and decrement a variable or something... but that timer might not get to fire (pun not intended) as often as we want it to. Comparing the CPU clock times of the last rendering cycle with the current cycle is a much safer option.

So, the average speed of our asteroids, aliens, bullets and players, and the average firing rate of the player and aliens, will be as close to whatever we want as they can be. There may still be some stuttering if the framerate is too low... but when you average out all the stutters, you'll always get roughly the same speeds and firing intervals.

What are we doing with the player's gun, though?

If enough time passes between render cycles, it's possible that the player might have been supposed to have fired several shots between this frame and the next. Maybe it took us 2.2 seconds to draw the last frame, so we should allow the player to have fired 4 shots, and say that he'll be able to fire again in approximately 0.3 seconds. We probably don't want him to have fired those bullets *simultaneously*, though, so we *also* make the new bullets tick as if they really *had* been fired 2.2, 1.7, 1.2, and 0.7 seconds ago, respectively, moving them into the appropriate positions.

Of course, this is not always ideal. Sometimes, we may want to separate ourselves from real time, at least to some extent. Maybe we don't want the player to drain all of his ship's power by accident, or we don't want the asteroids and aliens to teleport across the screen between frames. However, now we know how to control time to achieve one effect or the other.

Part 2. Messaging Systems

Chapter 7

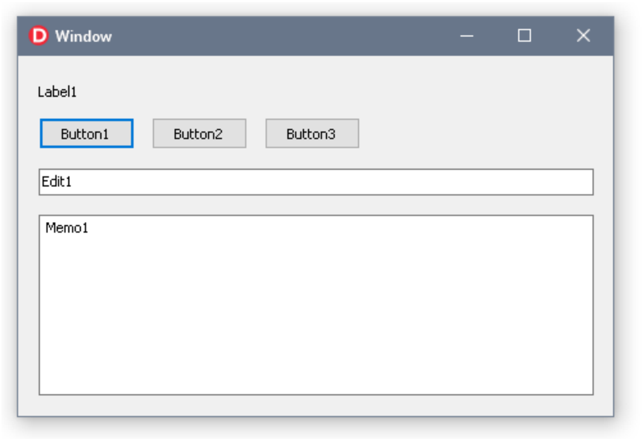
Windows Messaging System

The basic logic of the Windows message (event) loop is simple. Unless you have to directly interact with that system, you don't need to know more than what was explained in the previous chapters. Attaching event handlers to buttons and other controls is happening on a layer above the OS, and you don't need to know about any of the hidden actors involved. But where's the fun in that?

One of the basic building blocks in Windows is a *window*.

But what is a window?

Stupid question… it is quite obvious what a window is.



One window is pretty obvious in the above image… but, it is not the only one. Every button is a window, the edit is a window, the memo is a window. No, I didn't forget about the label, the label is not a window. And there are also invisible windows not meant to display anything.

A window in Windows is the primary building block of a GUI. The form window from the previous image is a parent window, and other windowed controls sitting on that form are child windows. The form window can also be a child of another window. Windows that don't have a parent, or have the desktop window as their parent, are called top-level windows.

Most Delphi VCL controls are thin wrappers around native Windows controls. So `TButton` is a wrapper around the Windows button control. `TLabel` does not have a window, and is considered to be a non-windowed control in Delphi. As far as Windows is concerned, it doesn't know that your label is a control, it only knows about the existence of its parent windowed control. All the handling required to paint and otherwise manage such non-windowed controls is implemented purely by Delphi, in the VCL framework.

When it comes to FireMonkey on Windows, only forms are windowed controls, because they are top-level entities needed for interaction with the OS. All other controls behave just like `TLabel`—they don't have any window attached, and FMX takes care of everything.

When I said that only forms are windowed controls in FireMonkey, that was not entirely true, but it was necessary to understand the architectural difference between VCL and FMX. FMX also supports platform-native controls, and such controls are backed up by the platform (OS) on which the application runs. So if you use such a native control in FMX on Windows, it will have a window attached, and will behave like any other windowed control.

While most of the controls in VCL are windowed controls, most of their FMX counterparts are not windowed controls.

What makes windows interesting in terms of event-based programming is that windows are also entities that can send and receive messages, and that makes them active participants in the Windows messaging system.

Each window has its own ID number, which is automatically assigned when the window is created by the OS. That ID, known as a window *handle*, is then used by the messaging system to identify the recipients of messages. If you want to send some message to a specific window, you need to know its ID.

Because you cannot send messages to anything but windows, Windows has support for non-GUI, invisible windows for messaging purposes. For instance, when you create a timer, that is a non-GUI control. It will have an invisible window attached, that will enable the timer to receive `WM_TIMER` messages and run its `OnTimer` event handler.

Some window properties are immutable, and when you create a window, they cannot be changed later on. Delphi controls conveniently allow you to change such properties, but when you do so, that will trigger recreation of the associated window. When a window is recreated, from the OS' point of view it will represent a completely new window, and this window will have a different ID—a different handle.

This is important to know, because from a communication perspective, you cannot just take any window ID and expect that it will remain the same during the lifetime of a Delphi VCL control. You need to be able to react to such changes when they happen, and refresh the handle you stored for communication purposes.

7.1 Windows Message Queue

Every thread created in Windows can have its own message queue, but a queue is not automatically created for every thread. If a thread does not have any windows, it does not need a communication system. Only threads that have a window will have a message queue.

You cannot explicitly create a message queue, it will be created for a particular thread when you first create a window from that thread. Basically, every thread with windows will have its own message queue, and all windows belonging to that thread will use the same message queue for communication.

Once you have a window and a message queue, you can write your message loop.

The following example creates a basic Windows application from scratch, without using any Delphi-provided frameworks. It directly uses Windows API calls and types, provided by the `Winapi` units. It creates the main application window, which will automatically trigger the creation of a Windows message queue for the application thread (the only running thread). After we successfully create a window, we will also have created the main application loop, and `AppWndProc` defines our main application event handler.

```
program WinApp;

{$R *.res}

uses
  System.SysUtils,
  Winapi.Windows,
  Winapi.Messages;

var
  AppHandle: HWND;
  AppWndClass: WNDCLASS;
  AppMsg: MSG;

const
  APP_CLASS_NAME = 'HelloWorldClass';
  APP_TITLE = 'Hello World';


function AppWndProc(WndHandle: HWND; uMsg: UINT; wp: WPARAM; lp: LPARAM): LRESULT;
  stdcall;
begin
  case uMsg of
    WM_DESTROY:
      begin
        if WndHandle = AppHandle then
```

```
          PostQuitMessage(0);
        Result := 0;
      end;
    else Result := DefWindowProc(WndHandle, uMsg, wp, lp);
  end;
end;

begin
  FillChar(AppWndClass, SizeOf(AppWndClass), 0);
  AppWndClass.lpfnWndProc := @AppWndProc;
  AppWndClass.hInstance := hInstance;
  AppWndClass.lpszClassName := APP_CLASS_NAME;

  RegisterClass(AppWndClass);

  AppHandle := CreateWindowEx(0, // Optional window styles.
    APP_CLASS_NAME,             // Window class
    APP_TITLE,                  // Window title
    WS_OVERLAPPEDWINDOW,        // Window style

    // Size and position
    CW_USEDEFAULT, CW_USEDEFAULT, 480, 320,

    0,                          // Parent window
    0,                          // Menu
    hInstance,                  // Instance handle
    nil                         // Additional application data
    );

  if AppHandle = 0 then
    Exit;

  ShowWindow(AppHandle, CmdShow);

  // Message Loop
  while GetMessage(AppMsg, 0, 0, 0) do
    begin
      TranslateMessage(AppMsg);
      DispatchMessage(AppMsg);
    end;

end.
```

Compared to our Windows VCL Hello World example, which took only four lines of code to create and run an empty Windows application, this one took a bit more. Now is a good time to say *Thank you, VCL!*. Without it, we would have to write insane amounts of code just to do basic things.

Very simple basic logic, with just a few basic building blocks, suddenly does not look so simple anymore. Well, the basic simple logic is still there, but in order to create complex applications, we need a different set of functionalities, and bit by bit, our application castle is being built.

In the above example, we have used exact type names as defined in the Windows API: `HWND`, `MSG`, `WPARAM`, `LPARAM`... To make those Windows type names more Delphi-like, the `Winapi` units also define type aliases that can be used instead of the Windows-defined types. So you can use `THandle` instead of `HWND`, `Cardinal` for `UINT`, `UIntPtr` for `WPARAM`, and `IntPtr` for `LPARAM` and `LRESULT`.

If you are writing your own translation of some Windows API, it is always good to start with Windows-declared types, as you will be less likely to make an error in translation, especially with some lesser-used, obscure types. And it is quite easy to make a mistake with some commonly-used types. Unless handling the Windows API is your daily bread and butter, I dare you to give me the correct substitute for `WPARM` or `LPARAM`. You will probably use the wrong one...

On the other hand, using `THandle` and the various `TMessage` record declarations is more common.

Now, back to the Windows message queue...

When we created a window, we also attached a message (event) handler function to that window. Every window can have its own message handler, and the Windows API also defines a default message handler, `DefWindowProc`, that you can attach or call for messages which you don't know how to or don't *need* to handle yourself.

To send and receive messages, you obviously need to know what the message type looks like. `hwnd` is the handle of the receiver window, `message` is the ID of the sent message, and there are also two additional parameters, wParam and lParam, that can hold various data depending on the message type. `time` is the time at which the message is posted to the queue, and `pt` contains the coordinates of the mouse cursor at the time of posting. The last two parameters are not significant for message handling, and the event handler function will receive only the first four.

```
MSG = record
  hwnd: HWND;
  message: UINT;
  wParam: WPARAM;
  lParam: LPARAM;
  time: DWORD;
  pt: TPoint;
end;
```

The declarations of Windows message record types have changed with time, more precisely with operating system bitness. The days of 16-bit Windows are long gone, so you don't need to worry about it. `WPARAM` is an unsigned type, while `LPARAM` is signed, and on both 32-bit and 64-bit Windows, both types share the bitness of the operating system. In Windows messaging, when they are not just plain numbers, `WPARAM` is used for passing handle values, while `LPARAM` is used for pointers. And by now, it should be blatantly obvious that the Windows messaging system is built upon sending numbers (IDs) and pointers.

If your curiosity does not let you sleep, `W` in `WPARAM` comes from `word`, since in 16-bit Windows, that parameter contained a 16-bit word value, and `L` comes from `long`, as this parameter could hold a 32-bit long value back then.

We already know what `GetMessage` does—it retrieves a message from the message queue if there is one, or waits for the arrival of the next one. The retrieved message will be stored in a `var` parameter—the record we passed to `GetMessage`.

After that, we need to call `TranslateMessage`, which will translate the virtual key messages into char messages that are then passed to the message queue. This is special handling required by Windows, and not something generally required by the *event loop* design pattern.

The next call, `DispatchMessage`, *is* part of the *event loop* pattern. This will invoke the Windows message dispatching mechanism, and execute the event handler attached to the appropriate window—the one whose handle is specified by the `hwnd` parameter. After the message is received in our window's message handler, we can implement all of the logic needed for every specific message ID we want to handle.

In a slightly different loop variant, we can use `PeekMessage`, which will tell us whether there are messages in the queue. Depending on the passed parameters, `PeekMessage` can either retrieve and remove a message from the queue for further processing (similarly to what we do after message is retrieved by `GetMessage`), or it can leave the message in the queue and let us handle further retrieval and processing in a separate call.

7.2 Posting and sending messages

All user interactions, as well as other OS-specific messages, will be posted to message queues. But Windows also allows applications to post messages through the `PostMessage` function to other windows that are part of their process, or even to send messages to other processes (some security restrictions may apply with interprocess communication). `PostMessage` will post a message to the queue, and will return immediately before the message is handled by the receiving window.

```
function PostMessage(hWnd: HWND; Msg: UINT;
  wParam: WPARAM; lParam: LPARAM): BOOL; stdcall;
```

Besides queued messages, Windows supports sending messages directly to a particular window with `SendMessage`. Such messages are called non-queued messages, and the attached window message handler will run immediately upon receiving such a message. Directly sending a message is a blocking call and will return only after the message handler has finished executing.

```
function SendMessage(hWnd: HWND; Msg: UINT;
  wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
```

If you pass `HWND_BROADCAST` as the `hWnd` parameter when posting or sending messages, the messages will be broadcasted to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows.

`PostMessage` and `SendMessage` are not the only functions in Windows API that enable you to deliver queued or non-queued messages, respectively. While they will suffice for the most common use cases, there are other variants that may be more appropriate when sending messages to windows created on different threads, or when broadcasting messages.

7.3 Message priority

Not all messages are made equal, and the Windows message queue does not work like a FIFO (first-in first-out) queue.

`GetMessage` and `PeekMessage` implement filtering, so you can specify which types - which range of messages—you would like to retrieve:

```
function GetMessage(var lpMsg: TMsg; hWnd: HWND;
  wMsgFilterMin, wMsgFilterMax: UINT): BOOL; stdcall;

function PeekMessage(var lpMsg: TMsg; hWnd: HWND;
  wMsgFilterMin, wMsgFilterMax, wRemoveMsg: UINT): BOOL; stdcall;
```

No matter what you specify in the filtering parameters, a `WM_QUIT` message will always be retrieved. During a `GetMessage` call, any directly sent, non-queued messages will be immediately delivered, then all messages specified by the filter.

If there is no filtering, messages will be received in the following order:

- Sent messages
- Posted messages
- Input (hardware) messages and internal system events
- Sent messages (again)
- WM_PAINT messages
- WM_TIMER messages

`WM_PAINT` and `WM_TIMER` messages are considered to be low priority messages, and if there are other types of messages in the queue, they will have priority over those two.

Note: The fact that `WM_TIMER` messages have lower priority than other messages has a few consequences. First, you cannot rely on timers to fire at an exactly specified interval. Next, because you cannot rely on the timer event firing at exactly the moment specified, you cannot use a timer event to count elapsed time.

For instance, if you have timers set to fire once every second, you cannot say that after it fired X times, X seconds had elapsed in the meantime. This approach will never yield correct results, and even if at some point it may seem such code is working, it can break at any time.

Incorrect code

```
procedure TMainForm.Timer1Timer(Sender: TObject);
begin
  ElapsedSeconds := ElapsedSeconds + 1;
  Label1.Caption := ElapsedSeconds.ToString;
end;

procedure TMainForm.StartCounting(Sender: TObject);
begin
  ElapsedSeconds := 0;
  Timer1.Enabled := True;
end;
```

Correct code

```
procedure TMainForm.Timer1Timer(Sender: TObject);
begin
  ElapsedSeconds := SecondsBetween(Now, StartTime);
  Label1.Caption := ElapsedSeconds.ToString;
end;

procedure TMainForm.StartCounting(Sender: TObject);
begin
  StartTime := Now;
  Timer1.Enabled := True;
end;
```

Another consequence of the previously mentioned inaccuracy, is that timer-driven animations may not always run smoothly, and they can last much longer than the anticipated time. If you need smooth and precisely-timed animations, you will need to use the `DeltaTime` approach mentioned in the *Game loop* chapter.

7.4 Custom messages

There is a whole range of Windows-defined messages related to various controls and specific purposes. The name of each predefined message ID constant begins with a few letters that denote its group. For instance, general window messages start with `WM`, button control-related messages are prefixed with: `BCM`, `BCN`, `BM`, and `BN`, and so on... When you read the Windows documentation for a particular control or functionality, you will also find all of the related message IDs.

You can also define your own custom message IDs by declaring your own ID numbers, or for inter-process communication, you can register a commonly shared message ID via the `RegisterWindowMessage` function, that takes a string as a parameter. Multiple applications using the same string for registration will receive the same message ID number for that string, and will be able to communicate using that ID while sending messages. Such an ID is only valid until the OS is restarted, and you cannot take that number as a hardcoded ID that will never change.

When registering your own message IDs, you must make sure that such an ID does not overlap with any predefined Windows ID, nor other IDs used in 3rd-party libraries. To prevent messaging chaos, Windows have several ranges of message IDs—some are reserved, and some are available for custom purposes.

Range	Meaning
0 through WM_USER –1	Messages reserved for use by the system
WM_USER through 0x7FFF	Integer messages for use by private window classes
WM_APP through 0xBFFF	Messages available for use by applications
0xC000 through 0xFFFF	String messages for use by applications
Greater than 0xFFFF	Reserved by the system

A declaration example for custom messages:

```
const
  CM_MyMessage = WM_USER + 200;
  APP_Message = WP_APP + 100;
```

Registering a system-wide message:

```
var
  MyMessageID: UINT;
  ...
  MyMessageID := RegisterWindowMessage('MyMessage');
```

Chapter 8

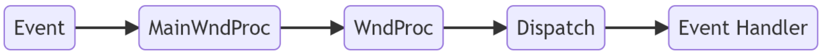
Windows Messaging System in VCL

As seen in the previous chapter, attaching message handlers to Windows controls is rather simple, but it raises a few questions.

Firstly, what about inheritance? If you have two windowed controls, one inheriting from the other, how can you change the message handling for a particular message. And next, what can you do to get rid of that ugly `case`?

Also, what about non-windowed controls? If they have no attached window, how can they receive messages and user interactions?

Delphi implements its own message dispatching mechanism on top of Windows-based messaging. That enables messaging support on non-windowed VCL child controls, through their parent control window, as well as a much simpler way of handling specific messages and the ability to declare a separate event handler for each message ID in the control class definition.



The following are the basic building blocks you need to know when using the Windows messaging system through VCL. If the control is not a windowed control—in other words, if it is not a descendant of `TWinControl`, but a descendant of `TControl`—it will have to have its `Parent` property set.

`WndProc` is the main entry point for all messages, and it can be written in a similar way to a regular window message handler with a `case` block, handling all message IDs you want to handle. Since it is a virtual method, you can override it in descendant classes for handling specific messages. For handling the rest, just call `inherited`.

```
type
  TControl = class(TComponent)
  private
    FParent: TWinControl;
    ...
    procedure DefaultHandler(var Message); override;
    procedure WndProc(var Message: TMessage); virtual;
    ...
    procedure CMVisibleChanged(var Message: TMessage); message CM_VISIBLECHANGED;
  end;

  TWinControl = class(TControl)
  ...
```

But the real power comes from the `message` directive, accompanied by a message ID. This is how you can easily declare specific message handlers for specific message IDs, even custom ones.

```
    procedure CMVisibleChanged(var Message: TMessage); message CM_VISIBLECHANGED;
```

The message handler method must have a single `var` parameter of the `TMessage` record type, as defined in the `Winapi.Messages` unit. Besides this generic message record, you can also use and define custom message records that match `TMessage` in size and general parameter layout.

```
  TMessage = record
    Msg: Cardinal;
    case Integer of
      0: (
        WParam: WPARAM;
        LParam: LPARAM;
        Result: LRESULT);
      1: (
        WParamLo: Word;
        WParamHi: Word;
        WParamFiller: TDWordFiller;
        LParamLo: Word;
        LParamHi: Word;
        LParamFiller: TDWordFiller;
        ResultLo: Word;
        ResultHi: Word;
        ResultFiller: TDWordFiller);
  end;
```

For instance, when handling mouse messages, it is simpler to use the `TWMMouse` record instead of `TMessage`.

```
TWMMouse = record
  Msg: Cardinal;
  MsgFiller: TDWordFiller;
  Keys: Longint;
  KeysFiller: TDWordFiller;
  case Integer of
    0: (
      XPos: Smallint;
      YPos: Smallint;
      XYPosFiller: TDWordFiller;);
    1: (
      Pos: TSmallPoint;
      PosFiller: TDWordFiller;
      Result: LRESULT);
end;
```

```
procedure WMLButtonUp(var Message: TWMMouse); message WM_LBUTTONUP;
```

If you don't want to add separate message handlers, you can also override the `WndProc` method and add custom handling for specific messages in additional `case` blocks. Just make sure that you call `inherited` for any messages that you don't handle. If you want to fallback to default behavior at any particular moment, you can also call the `DefaultHandler` method.

Note: It is customary to use `case` for handling different message IDs, because it is easier to add new IDs if necessary, and it can be easier to follow the code's logic. But, that does not mean you **must** use `case`, and you can freely use any appropriate control flow when inspecting message IDs.

8.1 Window recreation

VCL controls allow changing the otherwise-immutable properties of the underlying window. Changing such properties requires the window to be recreated with a new set of properties. When that happens, the window handle will also change.

Code that takes a VCL control's window handle and stores it somewhere else can be easily broken by the above mechanism. If you really need to take a window handle for communication purposes, you have a few ways of solving the above problem.

First, you can detect changes in the window handle by extending the control class, and notifying interested parties that the window handle they stored is about to be changed.

The following methods are responsible for creating and destroying window handles, so you can add notification system there:

```
    procedure CreateWindowHandle(const Params: TCreateParams); virtual;
    procedure DestroyWindowHandle; virtual;
```

However, the above methods are also part of a larger window recreation system, so trying to ask for a new handle from inside the `DestroyWindowHandle` method will not end well. There are other virtual methods involved in window recreation that are better suited for requesting a new handle after the old one is destroyed, like:

```
    procedure CMRecreateWnd(var Message: TMessage); message CM_RECREATEWND;
    procedure DestroyHandle; virtual;
```

Another thing you need to pay attention to is that window handle destruction is not triggered when the control is being destroyed. In such cases, if the `csDestroying` flag is set in `ComponentState`, you should not ask for handle recreation.

Also, the above approach will not work in a multithreaded scenario—when a handle is needed for inter-thread communication.

A better option than tracking window handle recreation from external code, one that also works in multithreading, is to have a separate window handle for communication purposes that will not change. You can extend the control to allocate an additional handle that will not be recreated just because the visual presentation of the control has changed.

An even better option, especially if you need to communicate with many controls, where creating extra handles would be a waste of resources, is to have a dedicated component that will have access to other controls as objects, not window handles, and that can then receive communication and safely pass it on from the context of the main thread.

The following chapter shows an example of allocating and deallocating a dedicated, non-visual window.

8.2 Window for non-visual components

To use the Windows messaging system, you need a window and its handle, but you don't necessarily need a visual control. To create a custom component capable of processing Windows messages, you need a few things: the ability to create and destroy the window handle, and a message handler function that will be attached to the window.

Doing all that is rather simple. RTL has the basic building blocks provided in `System.Classes`, in the form of the `AllocateHWnd` and `DeallocateHWnd` functions, that will take any appropriate method that matches the signature of the `TWndMethod` procedure and create a window with that method attached as a message handler.

In the message handler, you can handle any particular custom or predefined message, but you should always call `DefWindowProc` for any messages you don't specifically handle yourself. Even if your window is meant to process only your custom messages, it may receive a number of system messages that need to be properly handled.

The base class for such a non-visual component can be literally any Delphi class. But if you want to have the ability to drag and drop such a component on the form in the form designer, then you might use `TComponent` as a base class instead of `TObject`.

That also means you can take any existing class and attach window messaging functionality with just a few lines of code. Of course, such an extension will only work on Windows:

```
TWndMethod = procedure(var Message: TMessage) of object;
function AllocateHWnd(const AMethod: TWndMethod): HWND;
procedure DeallocateHWnd(Wnd: HWND);
```

```
uses
  Winapi.Windows,
  Winapi.Messages,
  System.SysUtils,
  System.Classes;

const
  CM_MY_EVENT = WM_USER + 100;

type
  TGear = class(TObject)
  protected
    FWndHandle: THandle;
    procedure WndProc(var Msg: TMessage); virtual;
  public
    constructor Create;
    destructor Destroy; override;
    property WndHandle: THandle read FWndHandle;
  end;

implementation

constructor TGear.Create;
begin
  inherited;
  FWndHandle := AllocateHWnd(WndProc);
end;

destructor TGear.Destroy;
```

```
begin
  DeallocateHWnd(FWndHandle);
  inherited;
end;

procedure TGear.WndProc(var Msg: TMessage);
begin
  case Msg.Msg of
    CM_MY_EVENT :
      begin
        OutputDebugString('GEAR');
      end;
    else
      // mandatory call to default message handler
      Msg.Result := DefWindowProc(FWndHandle, Msg.Msg, Msg.wParam, Msg.lParam);
  end;
end;
```

```
var
  Gear: TGear;
  ...
  PostMessage(Gear.WndHandle, CM_MY_EVENT, 0, 0);
```

8.3 Message handler inheritance

Delphi message handlers are not marked as virtual, so you may get the impression that you cannot override their behavior. But, this is not true, and all you need is to redeclare the same method, with the same message ID attached, in the descendant control.

For instance, if you want to change how some control responds to “erase background” messages. In the following example, we have completely changed the control’s behavior, and the logic implemented in the ancestor classes will not be called at all:

```
  TMyControl = class(TPanel)
    ...
    procedure WMEraseBkgnd(var Message: TWmEraseBkgnd); message WM_ERASEBKGND;
  end;

procedure TMyControl.WMEraseBkgnd(var Message: TWMEraseBkgnd);
begin
  Message.Result := 1;
end;
```

On the other hand, if you want to only partially change behavior, and still run the code implemented in the ancestors, you can just call inherited in the appropriate places:

```
procedure TMyControl.WMEraseBkgnd(var Message: TWMEraseBkgnd);
begin
  // always call inherited handler
  inherited;
  if SomeCondition then
    begin
    end;
end;

procedure TMyControl.WMEraseBkgnd(var Message: TWMEraseBkgnd);
begin
  if SomeCondition then
    begin
    end
  else
    begin
      // call inherited handler only in some cases
      inherited;
    end;
end;
```

Chapter 9

Custom messaging systems and patterns

Messaging systems are a core part of event-based architecture. While you can always use platform-provided systems, it is not always practical to do so. For one thing, platform-based messaging systems are not cross-platform, and such code cannot be used for cross-platform applications.

Besides not being cross-platform, OS messaging is usually quite complex and has many slightly different ways of doing things that you may never need. This is necessary for catering to the needs of a broad range of different applications that can run on the platform.

The basic principles of messaging systems are rather simple, and creating your own system that will cover your specific needs is not a very hard exercise. The complexity involved depends solely on your needs, and you can make it as simple or as complex as you want.

All you need is a message queue, a mechanism for retrieving and sending messages to the queue, and a mechanism for dispatching and handling messages. As we have seen, the Windows messaging system operates on a rather low level, and sending anything more complex than an integer number requires fiddling with pointers.

Custom messaging systems are not restricted by the underlying platform's messaging systems, and they can freely implement support for different types of messages, like records, strings, interfaces, objects...

There are a number of variations in messaging systems and patterns. We already covered the *event loop* pattern, that particularly covers central application control flow. Other messaging patterns include, but are not limited to: the *publish-subscribe pattern*, the *observer pattern*, *message queues*, and *event buses.*

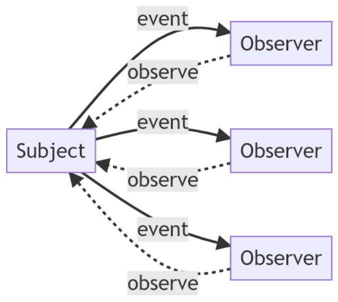
There is some amount of overlap in the functionality and features in all of the mentioned patterns, and often implementations will be some sort of hybrids. Some patterns also serve as building blocks for others. As you might have noticed, a message queue is also one of the building blocks of an event loop.

Let's start with event buses (also known as message buses) and message queues. The main differences are fairly subtle, and gradually being eroded by technological development. However, in principle, a message queue is used to notify a single recipient, while an event bus broadcasts to any recipients listening for a certain message type. This is usually called a *topic*, in a *publish-subscribe* relationship.

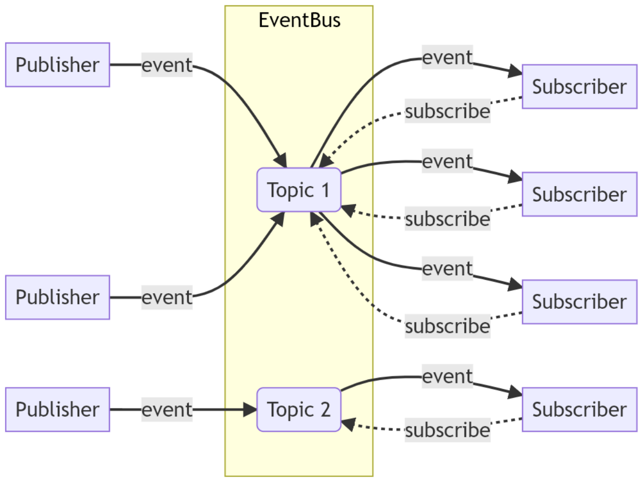
Of course, one can make a new queue for every new recipient, or use a topic for only one recipient. However, generally speaking, the publisher in a publish-subscribe relationship has no idea who's listening for the messages, whereas a message queue knows exactly who it's talking to. One pushes messages, the other polls them.

A good analogy to message queues and event buses might be an email account and a radio broadcast. If you give someone your email address, they will send emails to you, specifically. They may *also* send emails to other people... but those emails won't go to you. (... Carbon copying is a bit of a loophole here, until you consider the word "copying".) Meanwhile, the radio station has no idea who's listening (or even *if* they're listening), nor does it care. You simply tune in, subscribe to the topic, and listen to the broadcast, no questions asked.

The next question is, what is the publish-subscribe pattern? But to answer that, we must first look at the observer pattern. The observer pattern maintains a close relationship between subjects and observers. An object called the *subject* maintains a list of *observers*, and when the subject's state changes, it notifies its observers about the change. In the observer pattern, the subject is not interested about who its observers are, and it is the observer's responsibility to attach and detach themselves from the observed subject.



Unlike observers, the publish-subscribe pattern keeps a greater distance between publishers and subscribers. Publishers don't know anything about their subscribers, and subscribers don't need to know anything about publishers, they just know which topics they are interested in. An event bus is often used as a mediator—a communication channel—between publishers and subscribers.



Whether you are implementing your own messaging system or using an existing one, keep in mind that implementations can offer hybrid approaches. This is not necessarily a violation of design principles, but rather an adaptation to real-life needs where you can often use the same infrastructure for slightly different use cases.

For instance, the Windows messaging system, while primarily used for one-to-one notifications, can also be used to broadcast messages to a wider audience… except in Windows' case, that audience will not have the ability to mute certain topics, it can only choose not to react to some.

Chapter 10

System.Messaging

Besides supporting the Windows messaging system, the Delphi RTL has its own custom messaging system that is supported on all platforms, to cater for cross-platform messaging needs. You can find it in the `System.Messaging` unit.

Its main drawback is that its primary purpose is backing FireMonkey and other UI-related messaging needs, and it is not thread-safe. So if you need cross-platform messaging for inter-thread communication, you are out of luck and you will have to write your own system, or use some 3rd-party library.

The base message type used in `System.Messaging` is just an empty class. There is ambiguity in the class name, `TMessage`, with the type used in `Winapi.Messages`, but you will most likely not need to use both in the same unit, and if you do, fully qualified names will solve the problem.

When you define your own message classes, you can use `TMessage` as the base, but if you also need to send some accompanying value, you can also use the predefined `TMessage<T>` or `TObjectMessage<T>` classes.

```
  TMessageBase = class abstract;
  TMessage = TMessageBase;
  TMessage<T> = class(TMessage)
  TObjectMessage<T: class> = class(TMessage<T>)
```

For message handlers, you can register either anonymous methods or regular methods. Similarily to `TComponent` event handlers, the first parameter is the `Sender` object so you can use the same handler for receiving messages from multiple senders. The second parameter is the message itself.

```
TMessageListener = reference to procedure(const Sender: TObject;
  const M: TMessage);
TMessageListenerMethod = procedure (const Sender: TObject;
  const M: TMessage) of object;
```

A message is an object, and it needs to be manually released somewhere. When sending a message, you can specify whether you want the object to be automatically released after the message is sent to all recipients, or if you will take care of that yourself.

Manual memory management of such a message object requires that you don't grab and hold on to the sent message object instance in your message handler. After your message handler completes, the object might get killed. If you need to take some particular data from the message, you will need to create a local deep copy in the message handler.

Since the same message can be sent to more than one recipient, don't ever free the message in your message handler.

Incorrect code

```
procedure TMainForm.HandleMessage(const Sender: TObject; const M: TMessage);
begin
  // do something with message
  ...
  M.Free;
end;
```

The `TMessageManager` class handles everything else: providing methods for subscribing and unsubscribing to particular message classes, and sending messages. When you subscribe to some message class, you will be given a unique number `Id`, that you can use to unsubscribe the attached message handler from the class.

The subscribing/unsubscribing methods:

```
function SubscribeToMessage(const AMessageClass: TClass;
  const AListener: TMessageListener): Integer;

function SubscribeToMessage(const AMessageClass: TClass;
  const AListenerMethod: TMessageListenerMethod): Integer;

procedure Unsubscribe(const AMessageClass: TClass; Id: Integer;
  Immediate: Boolean = False);

procedure Unsubscribe(const AMessageClass: TClass;
  const AListener: TMessageListener; Immediate: Boolean = False);
```

```
procedure Unsubscribe(const AMessageClass: TClass;
  const AListenerMethod: TMessageListenerMethod; Immediate: Boolean = False);
```

Methods for sending messages:

```
procedure SendMessage(const Sender: TObject; AMessage: TMessage);
procedure SendMessage(const Sender: TObject; AMessage: TMessage;
  ADispose: Boolean);
```

You can create your own instances of `TMessageManager`, or you can use the default singleton instance, accessible through the `TMessageManager.DefaultManager` property.

`TMessageManager` is not a thread-safe class, and you cannot use it across thread boundaries. `DefaultManager` can only be used from the context of the main thread. However, you can safely create and use your own dedicated instances of `TMessageManager` in background threads, as long as you use any particular instance exclusively for communication within that thread.

The message manager is extremely simple to use, as can be seen from the following simple example of sending string messages that are automatically destroyed upon reception:

```
type
  TStringMessage = TMessage<string>;

var
  Id: Integer;

procedure TMainForm.SubscribeButtonClick(Sender: TObject);
begin
  Id := TMessageManager.DefaultManager.SubscribeToMessage(TStringMessage,
    procedure(const Sender: TObject; const M: TMessage)
    begin
      Memo1.Lines.Add(TStringMessage(M).Value);
    end);
end;

procedure TMainForm.UnsubscribeButtonClick(Sender: TObject);
begin
  TMessageManager.DefaultManager.Unsubscribe(TStringMessage, Id);
end;

procedure TMainForm.SendButtonClick(Sender: TObject);
begin
  TMessageManager.DefaultManager.SendMessage(Self, TStringMessage.Create('Test'));
end;
```

Part 3. Life, the Universe, and Everything

Chapter 11

Life, the Universe, and Everything

You are probably wondering what *Life, the Universe, and Everything* have to do with asynchronous programming. The answer is not 42, but everything and nothing.

This is one of the unexpected curveballs you have to deal with if you want to use some more sophisticated code flow, such as the *fluent interface* approaches commonly used in other languages. Actually, there are multiple curveballs and some can affect you even when using simple constructs like callbacks:

- lack of automatic memory management for object references
- mixing object and interface references
- limited generics support for interfaces
- lack of interface helpers
- lack of generic helpers
- lack of lambda expressions and anonymous method reference cycles

Before tackling some topics like tasks and futures, it is necessary to explain the above limitations in more detail and suggest possible workarounds. This will allow focusing on the particular topic without digressing into discussions about some limitations of the language and why some piece of code is written the way it is. If you ever want to dive into the waters of reactive programming, you will find that hard to do without being well versed in the subtleties of the Delphi language.

This is also a good place to jump into *Part 5. Thread Safety*, for readers who are beginners or not so proficient in multithreading.

When deciding which is the best option for implementing a *fluent interface* in some code, or if you should even use it at all, there is no absolute and simple answer. It is always the API itself and its requirements that will provide us with the best guidance. While keeping some consistency between different APIs is also good for coding speed and readability, it does not make sense to use a particular style when it does not fit well.

Keep in mind that the main point of *fluent interfaces* is the ability to write a concise, expressive code flow that will be easy to write, read and understand. If the shortcomings of the compiler are just too great to make a clean fluent API, then it is better to use the classic approach than to make a fluent mess just for the sake of being fluent.

One of the downsides of method chaining is harder debugging. You cannot simply place a breakpoint in the middle of the chain at the call site. You need to walk through every call from the beginning of the chain, or use breakpoints inside the functions you want to inspect.

There is also a difference in requirements between server and client applications. Client applications are usually more forgiving when performance is considered. If some piece of code is not called a million times per second, it does not really matter whether some action done on a button click will run for 100 or 200 miliseconds. On the server side, such differences in performance can be fatal for the server's health.

Some issues are more general issues, and are present when designing any kind of API. The same rules still apply. Let the API requirements guide you, and use what fits the best for that particular use case. The implementation details of complex frameworks can get ugly sometimes. That is fine, as long as the public API is as clean and as simple to use as possible. In other words, if using it gets too complicated or convoluted, you are probably doing it wrong.

Computer languages are living creatures, they evolve, bugs are being made and being fixed. That means some of the following issues might no longer be an issue in newer versions or even updates to the current version. Before applying any of the workarounds, it would be prudent to check how a particular block of code behaves in the targeted Delphi version.

Chapter 12

Automatic memory management

Fluent interfaces consist of method (function) chaining achieved by returning `Self`. Some API can require transforming one type to another. In such cases, the chaining function will not return `Self`, but a value of some other type. Commonly, there is no need to explicitly store the final result, and the chained sequence is logically treated as a single statement. While automatic memory management is not a requirement to implement such a sequence, it is highly desirable, especially in more complex scenarios involving type transformations, or multiple threads and a variable capture mechanism.

Being value types, records are not quite suitable for `Self` chaining, because the contents of the record will be copied on every return. Also, records with managed field types will introduce implicit variables for every call, and both of these behaviors can have a negative impact on performance.

Additionally, value type copying can make the code inside the *fluent* API awkward and error-prone, because it makes a difference whether the code first modifies its own fields and then assigns `Self` to the function's `Result`, or first assigns to the `Result` and then performs all operations on the `Result`.

While object references don't suffer from excessive copying and implicit references, they don't have automatic memory management in Delphi. So that leaves us with interface references. However, just like records, they also have downsides. Every call will trigger reference counting and create implicit references. Again, performance will suffer a bit compared to object references, but not as much as it can with larger records.

Another significant difference between records and classes is polymorphism. If some API can have many implementers, then using records is not a viable option.

Since Delphi 10.1 Berlin, you can mark function results as unsafe using the `[Result:unsafe]` attribute. This will prevent implicit variables and reference counting when using an interface-based *fluent* API. You will still need an initial strong reference to properly initialize the reference counting mechanism.

If you cannot use unsafe function results, Delphi does have another solution that prevents implicit variables, record copying, and reference counting hell. That is the dreaded `with` block—it does not provide all aspects and functionalities of a *fluent interface*, but in many cases where performance is paramount, it can be a suitable substitute.

If you really want to avoid using `with` in cases where its functionality is sufficient, and still want to have more readable code, the only option left is to use really short—specifically, one-letter—variable names. If you only need one such variable in a local method, you can use `_` as a really short and unobtrusive identifier.

There is one thing to keep in mind. Not all chaining functions will return the same object instance as their result. If you have an API that, for some reason, must create and return a new instance, then you cannot use unsafe on such a function.

If performance is absolutely critical, a *fluent interface* in Delphi might not even be the best choice. However, using any more sophisticated framework is usually a tradeoff between *speed of code* and *speed of coding*.

Generally, an interface-based *fluent* API, with some help from unsafe results, or even without it, will perform just fine.

The following is an extremely simplified example of a *fluent interface*-based XML builder using various possible implementations, including the classic procedural version for comparison. What all fluent versions have in common is that when using the builder API, all variants have exactly the same code—the following `Build` function will always look the same:

```
function Build: string;
begin
  Result := TXMLBuilder.New
    .Open('root')
    .Open('items')
    .Add('item', '123')
    .Add('item', '456')
    .Close('items')
    .Add('xxx', 'yyy')
    .Close('root')
    .SaveToString;
end;
```

We can even make the above function even more readable by additionally indenting code belonging to different levels:

```
function Build: string;
begin
  Result := TXMLBuilder.New
    .Open('root')
      .Open('items')
        .Add('item', '123')
        .Add('item', '456')
      .Close('items')
      .Add('xxx', 'yyy')
    .Close('root')
    .SaveToString;
end;
```

The procedural version is a bit more cramped, but the underscore identifier saves the day:

```
function Build: string;
var
  _: IXMLBuilder;
begin
  _ := TXMLBuilderProcedural.New;
  _.Open('root');
    _.Open('items');
      _.Add('item', '123');
      _.Add('item', '456');
    _.Close('items');
    _.Add('xxx', 'yyy');
  _.Close('root');
  Result := _.SaveToString;
end;
```

The procedural variant using `with` is also pretty good:

```
function Build: string;
begin
  with TXMLBuilderProcedural.New do
    begin
      Open('root');
        Open('items');
          Add('item', '123');
          Add('item', '456');
        Close('items');
```

```
          Add('xxx', 'yyy');
        Close('root');
        Result := SaveToString;
      end;
  end;
```

Looking at the above procedural examples, it can be easy to conclude that the *fluent interface* style is overhyped. It does slightly increase code readability, but just slightly. But this is just a very simple example. The real power of *fluent interfaces* and function chaining will become more obvious in the *Futures* chapter.

And now to examine the declarations and implementations of all the mentioned variants, and their differences. What is common to all *fluent* implementations is that the chaining functions always assign `Self` as the function's `Result`, except for the `SaveToString` function that is used to retrieve the final result that the builder produced in the form of a string.

First is the record-based variant. Because of the previously mentioned record copying (and implicit references), this variant is probably the least suitable one in real life scenarios:

```
    TXMLBuilder = record
    private
      Buf: string;
    public
      class function New: TXMLBuilder; static;
      function Open(const Tag: string): TXMLBuilder;
      function Close(const Tag: string): TXMLBuilder;
      function Add(const Tag, Value: string): TXMLBuilder;
      function SaveToString: string;
    end;

  class function TXMLBuilder.New: TXMLBuilder;
  begin
    Result := Default(TXMLBuilder);
  end;

  // change value first, then assign Self to Result
  function TXMLBuilder.Open(const Tag: string): TXMLBuilder;
  begin
    Buf := Buf + '<' + Tag + '>';
    Result := Self;
  end;

  // or assign Self to Result first, then change the value on Result
  function TXMLBuilder.Open(const Tag: string): TXMLBuilder;
  begin
```

```
    Result := Self;
    Result.Buf := Result.Buf + '<' + Tag + '>';
  end;

  function TXMLBuilder.Close(const Tag: string): TXMLBuilder;
  begin
    Buf := Buf + '</' + Tag + '>';
    Result := Self;
  end;

  // this function can be simplified to directly implement Open and Close
  // functionality, but it is not recommended for complex Open and Close functions
  function TXMLBuilder.Add(const Tag, Value: string): TXMLBuilder;
  begin
    Result := Open(Tag);
    Result.Buf := Result.Buf + Value;
    Result := Result.Close(Tag);
  end;

  function TXMLBuilder.Add(const Tag, Value: string): TXMLBuilder;
  begin
    Buf := Buf + '<' + Tag + '>' + Value + '</' + Tag + '>';
    Result := Self;
  end;

  function TXMLBuilder.SaveToString: string;
  begin
    Result := Buf;
  end;
```

Now, let's take a look at the class-based implementation. Basically, the type declaration is the same as it was with records, except with the `class` keyword instead of `record`. The implementation is also quite similar. Besides the fact that we don't need to pay attention to where we will assign `Self` to the function's `Result`, the major differences are in the `New` and `SaveToString` functions. `New` creates a new object instance, and `SaveToString` will `Free` that instance, since it is the logical end of the builder chain. This is a kind of *fluent interface* that can get away with using regular, non-managed class. If there were no particular function responsible for ending a function chain, we could not include `Free` to properly release memory. The ability to use a regular class for *fluent* design is therefore rather limited.

```
  TXMLBuilder = class
  private
    Buf: string;
  public
    class function New: TXMLBuilderObject; static;
    function Open(const Tag: string): TXMLBuilder;
    function Close(const Tag: string): TXMLBuilder;
    function Add(const Tag, Value: string): TXMLBuilder;
    function SaveToString: string;
  end;

class function TXMLBuilder.New: TXMLBuilder;
begin
  Result := TXMLBuilder.Create;
end;

function TXMLBuilder.Open(const Tag: string): TXMLBuilder;
begin
  Result := Self;
  Buf := Buf + '<' + Tag + '>';
end;

function TXMLBuilder.Close(const Tag: string): TXMLBuilder;
begin
  Result := Self;
  Buf := Buf + '</' + Tag + '>';
end;

function TXMLBuilder.Add(const Tag, Value: string): TXMLBuilder;
begin
  Result := Open(Tag);
  Buf := Buf + Value;
  Close(Tag);
end;

function TXMLBuilder.SaveToString: string;
begin
  Result := Buf;
  Free;
end;
```

When it comes to interface-based *fluent* implementations, we have two variants: with and without using the unsafe result attribute. The implementation part will be exactly the same as the above example with regular classes, except for the `SaveToString` function, which will not need to perform any memory management:

```
  IXMLBuilder = interface
    function Open(const Tag: string): IXMLBuilder;
    function Close(const Tag: string): IXMLBuilder;
    function Add(const Tag, Value: string): IXMLBuilder;
    function SaveToString: string;
  end;

  TXMLBuilder = class(TInterfacedObject, IXMLBuilder)
  private
    Buf: string;
  public
    class function New: IXMLBuilder; static;
    function Open(const Tag: string): IXMLBuilder;
    function Close(const Tag: string): IXMLBuilder;
    function Add(const Tag, Value: string): IXMLBuilder;
    function SaveToString: string;
  end;
  ...

function TXMLBuilder.SaveToString: string;
begin
  Result := Buf;
end;
```

```
  IXMLBuilder = interface
    [Result:unsafe] function Open(const Tag: string): IXMLBuilder;
    [Result:unsafe] function Close(const Tag: string): IXMLBuilder;
    [Result:unsafe] function Add(const Tag, Value: string): IXMLBuilder;
    function SaveToString: string;
  end;

  TXMLBuilder = class(TInterfacedObject, IXMLBuilder)
  private
    Buf: string;
  public
    class function New: IXMLBuilder; static;
    [Result:unsafe] function Open(const Tag: string): IXMLBuilder;
    [Result:unsafe] function Close(const Tag: string): IXMLBuilder;
    [Result:unsafe] function Add(const Tag, Value: string): IXMLBuilder;
    function SaveToString: string;
  end;
```

That leaves us with the good old-fashioned procedural approach. The main implementation difference is that there are no functions, so there is nothing to assign to a function's `Result`.

```
IXMLBuilder = interface
  procedure Open(const Tag: string);
  procedure Close(const Tag: string);
  procedure Add(const Tag, Value: string);
  function SaveToString: string;
end;

TXMLBuilder = class(TInterfacedObject, IXMLBuilder)
private
  Buf: string;
public
  class function New: IXMLBuilder; static;
  procedure Open(const Tag: string);
  procedure Close(const Tag: string);
  procedure Add(const Tag, Value: string);
  function SaveToString: string;
end;
```

Chapter 13

Interfaces

13.1 Mixing object and interface references

This is a rather common issue when using reference-counted classes in Delphi. In the context of reference counting, object references represent unsafe references, and interface references (unless explicitly marked as unsafe or weak) represent strong references.

The problem with mixing object and interface references is not so much in the *mixing* itself, but in the fact that proper initialization of the reference counting mechanism requires that the reference-counted object instance is stored in at least one strong interface reference immediately after construction. Failure to do so can cause memory leaks or crashes in subsequent code. Even if such code works without issues, it is only by chance, and can easily be broken simply by introducing code changes in seemingly unrelated places.

While this is a general issue when using interfaces, a *fluent interface* in combination with inline variables or no explicit variables is especially prone to broken reference counting. However, this can also be easily solved by using factory functions for instantiating reference-counted objects instead of directly calling constructors.

In the previous example using the *fluent* XML builder, this problem is solved by having the class function `New`, that returns an interface and as such, ensures proper reference count initialization:

```
class function TXMLBuilder.New: IXMLBuilder;
begin
  Result := TXMLBuilder.Create;
end;
```

The only problem with such an approach is that there is no way to enforce calling such a function in code. In other words, you can always inadvertently directly call the constructor of a class and potentially break the reference counting mechanism. The only way to prevent direct calls to the constructor is declaring the *fluent interface* implementation class in the implementation section of the unit, and hiding it from the outside world:

```
unit BuilderU;

interface

type
  IXMLBuilder = interface
    [Result:unsafe] function Open(const Tag: string): IXMLBuilder;
    [Result:unsafe] function Close(const Tag: string): IXMLBuilder;
    [Result:unsafe] function Add(const Tag, Value: string): IXMLBuilder;
    function SaveToString: string;
  end;

  TXMLBuilder = class
  public
    class function New: IXMLBuilder; static;
  end;

implementation

type
  TXMLBuilderImpl = class(TInterfacedObject, IXMLBuilder)
  private
    Buf: string;
  public
    [Result:unsafe] function Open(const Tag: string): IXMLBuilder;
    [Result:unsafe] function Close(const Tag: string): IXMLBuilder;
    [Result:unsafe] function Add(const Tag, Value: string): IXMLBuilder;
    function SaveToString: string;
  end;

class function TXMLBuilder.New: IXMLBuilder;
begin
  Result := TXMLBuilderImpl.Create;
end;

...

end.
```

Reference counting initialization can also be solved by explicitly typecasting the newly created object instance as an interface, but this approach is more error-prone than using a factory function everywhere:

```
TXMLBuilder.Create as IXMLBuilder;
```

The most common places where a reference-counted object instance is not properly initialized are:

- Storing an instance reference in a object reference
- Constructing an instance without assignment to any reference
- In-place construction in a `const` parameter
- Using an inline variable with type inference

13.2 Storing an instance reference in a object reference

Incorrect code

```
var
  Builder: TXMLBuilder;
begin
  Builder := TXMLBuilder.Create;
  ...
end;
```

Correct code

```
var
  Builder: IXMLBuilder;
begin
  Builder := TXMLBuilder.Create;
  ...
end;
```

13.3 Constructing an instance without assignment to any reference

After the Delphi constructor call is completed, the reference-counted object instance will have a reference count of 0. The first assignment to a strong reference (or some other reference counting trigger) will increase that count to 1, and make such an object properly initialized for reference counting. If there is no assignment, such an object instance will cause a memory leak:

Incorrect code

```
begin
  TXMLBuilder.Create;
  ...
end;
```

An explicit type cast, or using a factory function that returns an interface reference, will create a hidden (implicit) interface reference and ensure proper initialization:

Correct code

```
begin
  TXMLBuilder.New;
  //
  // or
  //
  TXMLBuilder.Create as IXMLBuilder;
  ...
end;
```

13.4 In-place construction in a `const` parameter

A common practice in languages with automatic memory management is constructing an object instance while it is being passed as a parameter to some method (function, procedure). Since reference-counted instances are automatically managed, it is tempting to do the same in Delphi and avoid unnecessary local variables:

Incorrect code

```
procedure Build(const Builder: IXMLBuilder);
...

begin
  Build(TXMLBuilder.Create);
  ...
end;
```

But, the instance constructed in the above code will not be properly initialized, and depending on the rest of the code inside the `Build` procedure, it can leak or cause a crash.

Why?

In Delphi, some parameter semantics, like `const`, do not trigger the reference counting mechanism. In such cases, in-place construction will not trigger the reference counting mechanism for such a parameter (as designed), and the object instance's reference counting will not be correctly initialized.

This is a rather unexpected compiler behavior (bug), because at that point, the compiler has all the necessary information to prevent problematic behavior and insert a hidden (implicit) interface reference to properly initialize such an instance:

Correct code

```
procedure Build(const Builder: IXMLBuilder);
...

begin
  Build(TXMLBuilder.New);
  //
  // or
  //
  Build(TXMLBuilder.Create as IXMLBuilder);
  ...
end;
```

13.5 Using an inline variable with type inference

Delphi 10.3 Rio introduced inline variables with type inference. This extremely useful feature needs to be used with some caution in combination with reference counted classes.

Namely, when constructing an object instance, the compiler will infer the exact type used to call the constructor and will create an object reference instead of, what we might quite easily assume, an interface reference. And of course, for such an object instance, the reference counting mechanism will not be properly initialized:

Incorrect code

```
begin
  var Builder := TXMLBuilder.Create;
  ...
end;
```

Correct code

```
begin
  var Builder := TXMLBuilder.New;
  //
  // or
  //
  var Builder: IXMLBuilder := TXMLBuilder.Create;
  //
  // or
  //
  var Builder := TXMLBuilder.Create as IXMLBuilder;
  ...
end;
```

13.6 Interfaces and generics

The moment you opt for using interfaces as the backbone of your fluent API, you will face another set of issues. If your API is using generics, you may find the support of Delphi generics for interfaces quite limiting. This is especially visible if you need to allow transformations from one generic type to another.

The first potentially limiting factor with generic interfaces is that you cannot use the `as` operator nor the `Supports` function on a generic interface. Actually, you can, but you probably will not like the results (see the output of the `GenIntf` program below).

All of the above functionality is using the interface GUID to distinguish one interface from another. However, when you add a GUID to a generic interface, the same GUID will match every specialization of that interface. If you want the ability to check for a specific one, you will need to explicitly declare all those interfaces and add a GUID to each one.

```
program GenIntf;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  System.Classes;

type
  IGen<T> = interface
  ['{AF9649A9-B068-442A-9A7C-1BD52B4047D8}']
    function Description: string;
  end;

  TGen<T> = class(TInterfacedObject, IGen<T>)
  public
    function Description: string;
  end;

function TGen<T>.Description: string;
begin
  Result := ClassName;
end;

function IsStringIntf(const Intf: IInterface): boolean;
var
  IG: IGen<string>;
begin
  Result := Supports(Intf, IGen<string>, IG);
  if Result then
    Writeln(IG.Description)
  else
    Writeln('Not String');
end;

procedure Test;
var
  IntegerIntf: IGen<Integer>;
  StringIntf: IGen<string>;
  ObjectIntf: IGen<TObject>;

  Intf: IInterface;
begin
  IntegerIntf := TGen<Integer>.Create;
```

```
    StringIntf := TGen<string>.Create;
    ObjectIntf := TGen<TObject>.Create;

    IsStringIntf(ObjectIntf);
    IsStringIntf(IntegerIntf);
    IsStringIntf(StringIntf);

    Intf := StringIntf;
    ObjectIntf := Intf as IGen<TObject>;

    Writeln('This is Object :', ObjectIntf.Description);
  end;

  begin
    try
      Test;
    except
      on E: Exception do
        Writeln(E.ClassName, ': ', E.Message);
    end;
  end.
```

Guess what the output of the above program will be?

If `Supports` and `as` worked properly, we would expect the following output:

```
Not String
Not String
TGen<System.string>
EIntfCastError: Interface not supported
```

What we got instead of the correct output is:

```
TGen<System.TObject>
TGen<System.Integer>
TGen<System.string>
This is Object :TGen<System.string>
```

Another interface issue, the inability to convert from one generic type to another, comes from the fact that in Delphi, interfaces don't support generic parameterized methods.

So you will be able to have a class-based *fluent* API with the following declaration:

```
TFoo<V> = class
public
  function First(const Value: V): TFoo<V>;
  function Then_<T>(const Value: V): TFoo<T>;
end;
```

But the interface-based equivalent will cause the following compiler error:

```
E2535 Interface methods must not have parameterized methods
```

```
IFoo<V> = interface
  function First(const Value: V): IFoo<V>;
  function Then_<T>(const Value: V): IFoo<T>;
end;

TFoo<V> = class(TInterfacedObject, IFoo<V>)
public
  function First(const Value: V): IFoo<V>;
  function Then_<T>(const Value: V): IFoo<T>;
end;
```

Generics can cause issues not only in interface-based APIs, but in class-based APIs, too. A rather good substitute for generics, if needed, is using the `TValue` type.

For a real life example of how `TValue` can be used, you can jump to the *Futures* chapter.

Chapter 14

Class helpers

14.1 Where class helpers help?

If we say that class inheritance is a way for vertical class extension, then class helpers provide a way for horizontal extension. In other languages, similar features are commonly called class extensions or extension methods. Simply put, they allow adding additional methods to the original class, without the need for altering its source code.

When it comes to *fluent interface* APIs, this allows expanding the API with additional methods without breaking the method chain. This kind of expansion can reduce the complexity of the original class, because it does not have to contain every possible method for every possible use case. It also helps in reducing unnecessary dependencies.

For instance, if you have a document writer fluent API that allows writing some documents using a set of predefined rules, then you can easily extend such an API to save such documents in different formats like PDF, Rich Text, HTML, without the need to include every particular library format as a dependency for your document writer API.

With a bit of help from class helpers, we can easily extend our object-returning XML builder:

```
type
  TXMLBuilderHelper = class helper for TXMLBuilder
  public
    function AddValues(const Tag: string; Values: array of string): TXMLBuilder;
  end;

function TXMLBuilderHelper.AddValues(const Tag: string; Values: array of string)
  : TXMLBuilder;
var
  Value: string;
```

```
begin
  Result := Self;
  Open(Tag);
  for Value in Values do
    Add('value', Value);
  Close(Tag);
end;
```

And then we can still chain methods, including new functionality, without breaking the chain:

```
function Build: string;
begin
  Result := TXMLBuilder.New
    .Open('root')
    .Open('data')
    .Close('data')
    .Open('items')
    .Add('item', '123')
    .Add('item', '456')
    .Close('items')
    .Add('xxx', 'yyy')
    .AddValues('values', ['aaa', 'bbb'])
    .Close('root')
    .SaveToString;
end;
```

But, in Delphi there is no helper support for extending interfaces. So our interface-based version cannot be extended in such a way.

We would need to use inheritance to implement additional functionality or introduce a standalone utility function. In any case, we would need to break the chain to insert typecasting or calls to utility functions, which will have a negative impact on code readability.

When designing a fluent API based on interfaces, the above limitation can have a significant impact on the design itself. It can be a tradeoff between adding functionality to the base interface even when it does not perfectly fit, not using a fluent API, or even creating separate APIs for different use cases, where duplicating small parts of code is a better choice than making a kitchen sink API that will cover all use cases and drag in unnecessary dependencies.

If you encounter a fluent API that feels a bit off and seems to suffer from any of the above-mentioned problems, don't immediately blame the developer. The whole thing may as well be a perfectly-crafted crooked house to work around the language limitations.

14.2 Interfaces don’t have helpers

The lack of helper support is another problem when using interfaces. That seriously limits the ability to extend an API as needed, and pushes developers to break the *YAGNI* (You aren’t gonna need it) principle. This may be less of a problem for in-house frameworks that can more easily be changed, but it is certainly a limiting factor for public ones, where a stable API is desirable.

14.3 Generics don’t have helpers

Another missing feature is generic helpers for both classes and interfaces. Similarly to the lack of interface helpers, the lack of generic helpers makes extending an API more difficult, although casting support for classes with generics is a bit more versatile than casting support for interfaces.

In general, generics can cause problems in older Delphi versions, and some code can trigger compiler errors. If backward compatibility is important, testing the API in older versions that need to be supported is mandatory during the design and development process. It can be very hard to work around compiler errors, and sometimes they can have a radical impact on the API design—even to the point where you will have to completely remove generic support.

While the compile-time type safety generics provide is always preferable to runtime errors, if generics are causing too much trouble, maybe the API would be better off without them.

Chapter 15

Anonymous methods and lambdas

Lambda expressions are extremely useful in reducing verbosity. This is especially visible in *fluent interfaces*, where anonymous methods obstruct readability. In Delphi, due to the `begin` and `end` keywords, anonymous methods are even more verbose than in other languages, where you can usually squeeze a one-line method body together with the parameter declaration without making an unreadable mess of it. Having lambda expressions in the language would drastically improve readability for many use cases.

But verbosity is not the primary issue when Delphi anonymous methods are concerned. It is so easy to fall into hidden traps of variable capture, creating reference cycles or memory leaks even in fairly trivial code.

Anonymous methods are not a new feature, as they were introduced quite some time ago in Delphi 2009, but are still rather *new* for many Delphi developers either because they are sparsely used in their code, or because many hidden aspects and pitfalls of this feature are not fully known and understood.

While I can certainly say that anonymous methods are no stranger to me, now and then I find myself baffled by some code behavior, where only after detailed inspection does it become clear what is happening. Nothing undocumented and nothing I really don't already know—usually rather trivial and obvious once you get to the bottom of it, which only proves this feature must be used with a healthy dose of caution.

On the bright side, once you establish some working patterns, you can rather safely use them without the need to inspect every line of code a dozen times.

Despite some flaws, anonymous methods and their ability to capture variables from outside contexts are an extremely important and almost indispensable feature even in rather simple scenarios and frameworks. Because of that importance, it is vital to cover in more depth how they work—even though covering language features is not the primary goal of this book.

They are also a prime example of how language features are more than just syntactic sugar, and how they can represent significant building blocks that help developers on a daily basis. Building blocks that would be otherwise very hard to achieve or would create intangible, messy code.

15.1 What exactly is an anonymous method?

Delphi has two procedural types capable of storing and executing plain functions and procedures, as well as methods—that is, functions and procedures declared within the context of a class.

Anonymous methods add a third one. As their name suggests, they are methods that do not have an associated identifier—they have no name. They represent blocks of code that can be assigned to a variable or passed as a parameter and executed later on, in the same manner as other procedural types. Anonymous methods are similar to closures in other languages, as they implement variable capture, which enables the usage of variables from the context in which they are defined.

The type declaration of anonymous methods follows the rules for declaration of plain functions and procedures, prefixed with `reference to`, and is called a *method reference type*:

```
type
  TAnonymousProc = reference to procedure;
  TAnonymousFunc = reference to function: Integer;

  TAnonymousProcWithParameters = reference to procedure(x, y: Integer);
  TAnonymousFuncWithParameters = reference to function(x, y: Integer): Integer;
```

15.2 What is it good for?

Functionally, anonymous methods in Delphi give us two things:

The first is the ability to write method code inline—passing it directly as a parameter to another method (function/procedure) or directly assigning it to a variable of the appropriate anonymous method type. On its own, that feature is neat, but it does not really give us way more than we already have. You can always write your function/procedure separately and then use it the same way you would use an anonymous method. Naming is hard, so not having to name something is a plus, and writing code closer to the place where it will be used is an additional plus because you don't have to jump much around the code, but still, all that does not make anonymous methods *worth the trouble.*

But the most important and distinguishing feature of anonymous methods is the ability to capture (use in the method body) variables from the context in which a particular anonymous method is defined. This is especially useful for various event handlers, or callback- and task-related patterns, because we can standardize (simplify) the method signature and still have access to all necessary data from the outer context. Simply put, for every variable needed to perform a particular functionality inside the method, we don't have to introduce another parameter.

For instance, with capture, we can have a task method in our imaginary task framework defined as a simple parameterless procedure for all use cases:

```
type
  TTaskProc = reference to procedure;

  TTask = class
  public
    class procedure Run(const Proc: TTaskProc); static;
  end;

class procedure TTask.Run(const Proc: TTaskProc);
begin
  // setup some code...
  ...
  Proc(); // and then run task proc
end;

begin
  TTask.Run(
    procedure
    begin
      Foo;
    end);
end;
```

If we need to use some string parameter inside `Task`, the capture mechanism allows us to use it directly:

```
var
  Data: string;
begin
  Data := 'Some data';
  TTask.Run(
    procedure
    begin
      Foo(Data);
    end);
end;
```

Without the capture mechanism, we would need to have additional declarations in the task framework in order to pass that string. Imagine doing that for every parameter type you might expect. With generics, you can reduce that number, but you would still need to make separate Run methods for different numbers of parameters. Or you would need to use some universal wrapper object, into which you could stuff everything you need to pass to the task. Now the complexity moves from the task framework to your code. Instead of having really clean code when you want to run a task, you would need to declare a wrapper object somewhere, then you'd need to create it, fill it up with data, eventually take care of its release, and what's worst, you would need to typecast that wrapper object in the task method body to get the appropriate data:

```
type
  TTaskProc = reference to procedure;
  TStringTaskProc = reference to procedure(const Value: string);

  TTask = class
  public
    class procedure Run(const Proc: TTaskProc); overload; static;
    class procedure Run(const Value: string; const Proc: TStringTaskProc);
      overload; static;
  end;

class procedure TTask.Run(const Proc: TTaskProc);
begin
  // setup some code...
  ...
  // run task proc
  Proc();
end;

class procedure TTask.Run(const Value: string; const Proc: TStringTaskProc);
begin
  // setup some code...
  ...
  // run task proc
  Proc(Value);
end;

var
  Data: string;
begin
  Data := 'Some data';
  TTask.Run(Data,
    procedure (const Value: string)
    begin
      Foo(Value);
    end);
end;
```

The capturing mechanism allows us to write much simpler and cleaner code and frameworks. Without it, anonymous methods would be nothing more than almost useless syntactic sugar.

15.3 How are anonymous methods implemented?

Capturing variables from an outside context implies extending their lifetime, as they may be long gone by the time the anonymous methods get to use them. Capturing the variables and extending their lifetime is only one side of the capturing mechanism—it also needs to perform proper cleanup after the variables are no longer needed.

Delphi already uses reference counting to maintain the lifetime of some types, so using automatic reference counting as the lifetime management mechanism in anonymous methods is a logical choice.

Anonymous methods are basically defined as interfaces with a single method - `Invoke`—implemented by a hidden reference-counted class, and captured variables are stored as fields of that class. When an anonymous method is accessed, an instance of that class is constructed behind the scenes, and it is kept alive through reference counting for as long as is required by the anonymous method it wraps.

If we translated the previously declared method reference types into the appropriate interface declarations, they would look like this:

```
type
  IAnonymousProc = interface
    procedure Invoke;
  end;

  IAnonymousFunc = interface
    function Invoke: Integer;
  end;

  IAnonymousProcWithParameters = interface
    procedure Invoke(x, y: Integer);
  end;

  IAnonymousFuncWithParameters = interface
    function Invoke(x, y: Integer): Integer;
  end;
```

Variable capture solved, and they all lived happily ever after.

Well no, not really. It all works well, until you start capturing reference-counted object instances exposed through interfaces. Actually, you can create nice cycles with records, too.

Anonymous methods (closures), in combination with ARC, can form strong reference cycles and cause memory leaks. That is an inevitable side effect of the inner workings of the reference counting mechanism. This is not a Delphi-specific issue, though the Delphi implementation throws a few curveballs of its own.

A strong reference cycle is created when a class (or record) stores a closure in its field, and at the same time, that closure captures some other fields, or calls some other method from that class. That is called *capturing self*, and it is an inevitable consequence of capturing variables from the surrounding context in combination with the reference counting mechanism. Besides cycles created between an object instance (or record) and an anonymous method, cycles can also be created when an anonymous method references itself—the variable where it is stored, even if it is a local variable.

One of the issues related to the capture mechanism comes from the fact that multiple anonymous methods defined in the same context will be backed up by the same auto-generated class, opening up a way to have less obvious reference cycles.

15.4 Anonymous method variable capture

Let's shed some light on the variable capture process. After all, this is the process that can create reference cycles. Obviously, capturing locally-declared integers, strings and similar types cannot have a negative impact on memory management in any way, nor can it create reference cycles. But, before one can run, one should learn how to walk.

`System.SysUtils` contains a number of predefined anonymous method types, but to make the code easier to follow, I will declare the appropriate anonymous method types and some utility methods, instead of using pre-declared types:

```
type
  TAnonymousProc = reference to procedure;

  TAnonymousStringFunc = reference to function: string;

procedure Execute(Proc: TAnonymousProc);
begin
  // Execute passed anonymous method
  Proc;
end;
```

The first example is a bit contrived, as there is really no need to use anonymous methods in this manner. Running the code would produce the same result as we would get if we just skipped all the anonymous methods and directly used a local variable.

```
...
procedure Test;
var
  Number: Integer;
begin
  Number := 42;

  Execute(procedure
    begin
      Writeln('Anything ', Number);
    end);

  Number := 0;

  Execute(procedure
    begin
      Writeln('Bang ', Number);
      Number := 42;
    end);

  Execute(procedure
    begin
      Writeln('Resurrected ', Number);
    end);
end;

begin
  Test;
end.
```

```
Anything 42
Bang 0
Resurrected 42
```

Not only can we capture a variable, but we can change its value from within the anonymous methods.

Let's change the `Test` method, and instead of executing methods inline, store them into variables for later execution:

```
procedure Test;
var
  Number: Integer;
  Proc1, Proc2, Proc3: TAnonymousProc;
begin
  Number := 42;

  Proc1 := procedure
    begin
      Writeln('Anything ', Number);
    end;

  Number := 0;

  Proc2 := procedure
    begin
      Writeln('Bang ', Number);
      Number := 42;
    end;

  Proc3 := procedure
    begin
      Writeln('Resurrected ', Number);
    end;

  Execute(Proc1);
  Execute(Proc2);
  Execute(Proc3);
end;
```

```
Anything 0
Bang 0
Resurrected 42
```

Now what? Why is `Anything 0` printed?

Well, obviously, for the same reason that we were able to change values from within the anonymous methods—because we captured the location (pointer) of the variable and not its value. Before we called any of our anonymous methods, we executed the following code:

```
  Number := 42;
  Number := 0;
```

That is why when we started executing anonymous methods, the value in `Number` was `0`. Whatever you do with the captured variable inside or outside anonymous methods will influence that variable in order of execution, not order of definition.

To move things further, let's declare `Proc1`, `Proc2` and `Proc3` outside the `Test` procedure, and call them outside its scope:

```
var
  Proc1, Proc2, Proc3: TAnonymousProc;

procedure Test;
var
  Number: Integer;
begin
  Number := 42;

  Proc1 := procedure
    begin
      Writeln('Anything ', Number);
    end;

  Number := 0;

  Proc2 := procedure
    begin
      Writeln('Bang ', Number);
      Number := 42;
    end;

  Proc3 := procedure
    begin
      Writeln('Resurrected ', Number);
    end;
end;

begin
  Test;
  Execute(Proc1);
  Execute(Proc2);
  Execute(Proc3);
end.
```

```
Anything 0
Bang 0
Resurrected 42
```

As expected, we had the same output as when executing anonymous methods within the `Test` procedure, as they have captured the local `Number` variable and extended its lifetime.

Now that it is clearer how capture works, we can understand why capturing object instances will not work as we might wish, at least not when they are not backed up by the reference counting mechanism. Anonymous methods can extend the lifetime of the reference, but they cannot extend the lifetime of the associated object instance itself.

The following is a simple example where composing strings is done within the anonymous method. By using different types of local variables, we can observe and compare their behavior.

What exactly are we testing in the `Test` procedure? First, we will acquire an anonymous method by using the `Composer` function. Then, we will execute that function two times to observe what is happening with the captured variables, and then we will acquire the anonymous method again and execute it once—merely to prove that each time we call the anonymous method factory, we will be served with a brand new anonymous method, which captured a fresh set of local variables. This is expected behavior, because every time you call a function, a new stack frame will be created to serve that call, and local variables will be allocated on that stack frame:

```
function Composer: TAnonymousStringFunc;
var
  Text: string;
  Number: Integer;
begin
  Text := 'Number';
  Number := 5;
  Result := function: string
    begin
      Inc(Number);
      Result := Text + ' ' + IntToStr(Number);
    end;
end;

procedure Test;
var
  Ref1, Ref2: TAnonymousStringFunc;
  Str: string;
begin
  // Get anonymous method
  Ref1 := Composer();
  // Call anonymous method
  Str := Ref1;
  Writeln(Str);
  // Call anonymous method
  Str := Ref1;
  Writeln(Str);
  // Get anonymous method again
  Ref2 := Composer();
  // Call new anonymous method
  Str := Ref2;
  Writeln(Str);

  // Call first anonymous method again
  Str := Ref1;
  Writeln(Str);
end;

begin
  ReportMemoryLeaksOnShutdown := true;
  Test;
end.
```

```
Number 6
Number 7
Number 6
Number 8
```

Looking at the output confirms what we already know. The anonymous method captures local variables and extends their lifetime beyond the lifetime they would have had while confined to the local function call—that is visible from the first two lines. The third one shows that the new anonymous method comes with a new set of variables, and the last one confirms that the variables captured by the first method are not influenced by the second one. They are independent sets.

Records are value types, so capturing records should work just about the same way:

```
  TContentRec = record
    Text: string;
    Number: Integer;
    constructor Create(const AText: string; ANumber: Integer);
  end;

constructor TContentRec.Create(const AText: string; ANumber: Integer);
begin
  Text := AText;
  Number := ANumber;
end;

function Composer: TAnonymousStringFunc;
var
```

```
    Content: TContentRec;
  begin
    Content := TContentRec.Create('Number', 5);
    Result := function: string
      begin
        Inc(Content.Number);
        Result := Content.Text + ' ' + IntToStr(Content.Number);
      end;
  end;
```

Calling our Test procedure will yield the same result:

```
Number 6
Number 7
Number 6
Number 8
```

Now let's try objects. As we already know, changes to captured variables happen sequentially, following the code's order of execution. The problem with objects is that we have to manage their memory. When should we release the local object instance, and is it even possible to use it in this manner?

```
    TContentObject = class(TObject)
    public
      Text: string;
      Number: Integer;
      constructor Create(const AText: string; ANumber: Integer);
    end;

  constructor TContentObject.Create(const AText: string; ANumber: Integer);
  begin
    Text := AText;
    Number := ANumber;
  end;

  function Composer: TAnonymousStringFunc;
  var
    Content: TContentObject;
  begin
    Content := TContentObject.Create('Number', 5);
    Result := function: string
      begin
        Inc(Content.Number);
        Result := Content.Text + ' ' + IntToStr(Content.Number);
    end;
end;
```

Calling Test again shows the same output. Good. But we leaked two TContentObject instances. Not so good.

```
Number 6
Number 7
Number 6
Number 8
```

Let's try to handle the leak:

```
function Composer: TAnonymousStringFunc;
var
  Content: TContentObject;
begin
  Content := TContentObject.Create('Number', 5);
  Result := function: string
    begin
      Inc(Content.Number);
      Result := Content.Text + ' ' + IntToStr(Content.Number);
    end;
  Content.Free;
end;
```

No leak, but what happened to our output? Well, we are accessing a dangling reference since by the time we call the function, our Content object instance has been released. Since the string variable is managed by the compiler, it has been cleaned after the instance was released, and the Text field contains an empty string. On the other hand, the integer field Number will hold a leftover value:

```
6
2
6
2
```

If you are still not convinced we are dealing with a dangling pointer, just nil the Content variable, and enjoy the access violation exception that causes:

```
  ...
  Content.Free;
  Content := nil;
```

So, this approach obviously does not work. What if we `Free` the `Content` from within the body of the anonymous method? By now, it should be pretty obvious that that approach will not work either, but...

```
function Composer: TAnonymousStringFunc;
var
  Content: TContentObject;
begin
  Content := TContentObject.Create('Number', 5);
  Result := function: string
    begin
      Inc(Content.Number);
      Result := Content.Text + ' ' + IntToStr(Content.Number);
      Content.Free;
      Content := nil;
    end;
end;
```

The end result—`Number 6`, followed by an access violation exception caused by the second execution of the anonymous method. Well, at least the first call produced an appropriate result.

If we also add a different `Test2` procedure that only executes each anonymous method once, we will get the correct result and no leaks:

```
procedure Test2;
var
  Ref1, Ref2: TAnonymousStringFunc;
  Str: string;
begin
  Ref1 := Composer();
  Str := Ref1;
  Writeln(Str);

  Ref2 := Composer();
  Str := Ref2;
  Writeln(Str);
end;
```

Generally speaking, there is no reason to use this kind of code. We cannot reuse a locally constructed object, and if we really need one, we should move it to the body of the anonymous method. That way, it will not matter how many times we have to call the procedure.

```
function Composer: TAnonymousStringFunc;
begin
  Result := function: string
    var
      Content: TContentObject;
    begin
      Content := TContentObject.Create('Number', 5);
      try
        Inc(Content.Number);
        Result := Content.Text + ' ' + IntToStr(Content.Number);
      finally
        Content.Free;
      end;
    end;
end;
```

Of course, if you want to have the same output as before, `6 7 6 8`, you will not like this one:

```
Number 6
Number 6
Number 6
Number 6
```

If you need to have a local object instance for whatever reason, and you also need to update its fields, you will need additional local variables (value types) that can handle being captured by the anonymous methods:

```
function Composer: TAnonymousStringFunc;
var
  Number: Integer;
begin
  Number := 5;
  Result := function: string
    var
      Content: TContentObject;
    begin
      Inc(Number);
      Content := TContentObject.Create('Number', Number);
      try
        Result := Content.Text + ' ' + IntToStr(Content.Number);
      finally
        Content.Free;
      end;
    end;
  end;
```

And the expected output is back again:

```
Number 6
Number 7
Number 6
Number 8
```

There is another solution to our object instance problem. If the object instance we have to construct locally is reference-counted, its memory will be automatically managed. The anonymous method will be capable of holding that instance and releasing it when it is no longer needed. Since this is a locally created object instance that does not reference the anonymous method in any way, there is no possibility of creating reference cycles.

The only thing we have to keep in mind here is that a locally constructed object instance will be kept alive as long as the anonymous method referencing it is alive. In our example, that will happen in the epilogue of the `Test` procedure. If we needed to release it sooner, we would have to set the `Refxxx` variable holding that particular instance to `nil`.

Also, if an anonymous method reference is passed on to other parts of the framework that might hold that reference, or if we move that reference declaration into the global scope, our local object might not be promptly released.

Keeping track of and knowing what happens with your anonymous method references when interacting with other code is important, as that code might have issues or bugs of their own that will prevent the timely release of your method and all captured variables.

```
    TContentObject = class(TInterfacedObject)
    public
      Text: string;
      Number: Integer;
      constructor Create(const AText: string; ANumber: Integer);
    end;

  function Composer: TAnonymousStringFunc;
  var
    Intf: IInterface;
  begin
    Intf := TContentObject.Create('Number', 5);
    Result := function: string
      var
        Content: TContentObject;
      begin
        Content := TContentObject(Intf);
        Inc(Content.Number);
        Result := Content.Text + ' ' + IntToStr(Content.Number);
      end;
  end;
```

15.5 Capturing and loops

While looping, anonymous method capturing does not behave any differently than in the previous explanation, but loops are a frequently used feature and warrant a specific example:

```
uses
  System.SysUtils;

procedure Test;
var
  Functions: array of TFunc<Integer>;
  Func: TFunc<Integer>;
  i: Integer;
begin
  SetLength(Functions, 5);

  for i := 0 to High(Functions) do
    Functions[i] :=
      function: Integer
      begin
        Result := i;
      end;

  for Func in Functions do
    Writeln(Func());
end;

begin
  Test;
end.
```

If you happily run the above code, expecting integers from 0 to 4 as output, you will be surprised to see the following output:

```
5
5
5
5
5
```

If you were not surprised, congratulations. You understood how capture works and that it captures the location of variables, not their values at a specific point during code execution.

If you were surprised, don't worry… Loops are tricky. When you are reading the above code, there is a discrepancy between the actual value of `i` outside the anonymous function and inside the anonymous function for every loop iteration, and it is very easy to get wrong assumptions.

In the above example, by the time the declared anonymous functions are called, the `for` loop where the functions are defined is finished, and its loop variable `i` contains the value 5, so that is the value our anonymous functions will return when called.

If we call the function inside the `for` loop, it will return the current value of the `for` loop variable `i`, and output the numbers 0 to 4:

```
  for i := 0 to High(Functions) do
    begin
      Functions[i] :=
        function: Integer
        begin
          Result := i;
        end;
      Writeln(Func());
    end;
```

However, calling the function during the loop defeats the purpose of creating an anonymous function in the first place. Also, if you use anonymous methods to execute some parallel tasks, you cannot count on the captured variables to have the expected values at the moment of task execution.

To solve that problem, you have to capture the actual value of the loop variable `i` during the loop. Wrapping the anonymous function declaration into a regular function and passing all necessary variables as parameters (in this example, there's only one) will create copies of their values on the stack, and allow the anonymous method capture mechanism to capture each particular copy (and its value) instead of the original:

```
uses
  System.SysUtils;

function CreateFunction(Value: Integer): TFunc<Integer>;
begin
  Result :=
    function: Integer
    begin
      Result := Value;
    end;
end;

procedure Test;
var
  Functions: array of TFunc<Integer>;
  Func: TFunc<Integer>;
  i: Integer;
begin
  SetLength(Functions, 5);

  for i := 0 to High(Functions) do
    Functions[i] := CreateFunction(i);

  for Func in Functions do
    Writeln(Func());
end;

begin
  Test;
end.
```

```
0
1
2
3
4
```

Using reference types in a loop is not more complicated than using value types if the reference itself is passed as a value. Actually, the compiler will not even allow capturing reference types that are not passed as value or const parameters:

```
E2555 Cannot capture symbol ‘Value’
```

Incorrect code

```
function CreateFunction(var Value: TContentObject): TFunc<TContentObject>;
function CreateFunction([ref] Value: TContentObject): TFunc<TContentObject>;
```

Correct code

```
function CreateFunction(const Value: TContentObject): TFunc<TContentObject>;
function CreateFunction(Value: TContentObject): TFunc<TContentObject>;
```

```
function CreateFunction(Value: TContentObject): TFunc<TContentObject>;
begin
  Result :=
    function: TContentObject
    begin
      Result := Value;
    end;
end;

procedure Test;
var
  List: array of TContentObject;
  Functions: array of TFunc<TContentObject>;
  Func: TFunc<TContentObject>;
  i: Integer;
begin
  SetLength(List, 5);
  for i := 0 to High(List) do
    List[i] := TContentObject.Create(i);

  SetLength(Functions, 5);
  for i := 0 to High(Functions) do
    Functions[i] := CreateFunction(List[i]);

  for Func in Functions do
    Writeln(Func.Number);

  for i := 0 to High(List) do
    List[i].Free;
end;
```

15.6 Anonymous method reference cycles

As previously mentioned, variable capture can cause reference cycles. If you avoid capturing variables, you will successfully avoid creating cycles. :)

If you think that was just a joke... well, it is a joke, but only to a point. Capturing variables is a powerful feature, but also one that can easily be abused, leading to bad code, regardless of cycles. So when you use anonymous methods and their ability to capture variables from the surrounding context, make sure that you are not making a real mess of your code along the way.

Since the main point of anonymous methods in various frameworks is relying on variable capture, you probably cannot and should not avoid capture at all cost, but you can minimize it and use it only when necessary.

For instance, moving local variable declarations that are used only inside an anonymous method to its variable section, limiting their scope and avoiding the capture.

In the following example, the `Abc` variable is not used anywhere outside the anonymous method, so it is better to have it moved inside, even though such local variables don't usually cause any trouble in terms of reference cycles:

```
procedure Foo;
var
  Abc: string;
begin
  Execute(
    procedure
    begin
      Abc := 'abc';
    end);
end;
```

Correct code

```
procedure Foo;
begin
  Execute(
    procedure
    var
      Abc: string;
    begin
      Abc := 'abc';
    end);
end;
```

Probably the most devious reference cycles are ones created with several anonymous methods. Other kinds of reference cycles involving anonymous methods are more obvious and are more easily recognized, as they resemble reference cycles created by regular strong references. You just have to remember that any anonymous method reference is an interface reference in disguise.

And now, let's make some cycles...

Incorrect code

```
procedure Test;
var
  Proc1, Proc2: TAnonymousProc;
begin
  Proc1 :=
    procedure
    begin
      Writeln('Procedure 1');
    end;

  Proc2 :=
    procedure
    begin
      Proc1;
      Writeln('Procedure 2');
    end;

  Proc2;
end;

begin
  ReportMemoryLeaksOnShutdown := true;
  Test;
end.
```

The above code will leak the hidden object instance used to back up those two anonymous methods. This leak happens because of specific implementation details—in this case, the fact that both of our methods are backed up by the same hidden object instance. Since the second method references the first one, it effectively references its own *supporting instance*.

Since `Proc1` is the one causing the cycle, we can solve this leak by explicitly setting it to `nil` before the end of the local procedure:

Correct code

```
...
  Proc2;
  Proc1 := nil;
end;
```

Even if we removed the explicit `Proc2` variable, we would still create a leak, because the parameter passed to the `Execute` procedure would be implicitly captured within the same context:

Incorrect code

```
procedure Test;
var
  Proc1: TAnonymousProc;
begin
  Proc1 :=
    procedure
    begin
      Writeln('Procedure 1');
    end;

  Execute(
    procedure
    begin
      Proc1;
      Writeln('Procedure 2');
    end);
end;
```

However, if we move the first procedure to the second one, they will be defined in different code blocks, and supported by two different instances:

Correct code

```
procedure Test;
begin
  Execute(
    procedure
    begin
      Execute(
        procedure
        begin
          Writeln('Procedure 1');
        end);
      Writeln('Procedure 2');
    end);
end;
```

Coming up next, a cycle created by a record or class storing a reference to an anonymous method that captures other members of that record or object instance:

Incorrect code

```
type
  TAnonymousRec = record
    Number: Integer;
    Proc: TAnonymousProc;
  end;

procedure Test;
var
  Rec: TAnonymousRec;
begin
  Rec.Number := 5;
  Rec.Proc := procedure
    begin
      Writeln(Rec.Number);
    end;
  Rec.Proc();
```

```
end;

begin
  ReportMemoryLeaksOnShutdown := true;
  Test;
end.
```

Since records are value types, the above code will create a cycle, because the whole record value (holding a reference to the anonymous method) will be captured. Converting `TAnonymousRec` to a class will not create a reference cycle, since object references are not strong references in terms of reference counting:

Correct code

```
type
  TAnonymousObject = class(TObject)
  public
    Number: Integer;
    Proc: TAnonymousProc;
  end;

procedure Test;
var
  Obj: TAnonymousObject;
begin
  Obj := TAnonymousObject.Create;
  try
    Obj.Number := 5;
    Obj.Proc := procedure
      begin
        Writeln(Obj.Number);
      end;
    Obj.Proc();
  finally
    Obj.Free;
  end;
end;
```

However, if we convert the above class to a reference-counted class, the interface reference, in combination with the anonymous method, will create a strong reference cycle:

```
type
  IAnonymousObject = interface
    function GetNumber: Integer;
    function GetProc: TAnonymousProc;
    procedure SetNumber(Value: Integer);
    procedure SetProc(Value: TAnonymousProc);
    property Number: Integer read GetNumber write SetNumber;
    property Proc: TAnonymousProc read GetProc write SetProc;
  end;

  TAnonymousObject = class(TInterfacedObject, IAnonymousObject)
  private
    FNumber: Integer;
    FProc: TAnonymousProc;
    function GetNumber: Integer;
    function GetProc: TAnonymousProc;
    procedure SetNumber(Value: Integer);
    procedure SetProc(Value: TAnonymousProc);
  public
    property Number: Integer read GetNumber write SetNumber;
    property Proc: TAnonymousProc read GetProc write SetProc;
  end;
```

Incorrect code

```
procedure Test;
var
  Obj: IAnonymousObject;
begin
  Obj := TAnonymousObject.Create;
  Obj.Number := 5;
  Obj.Proc := procedure
    begin
      Writeln(Obj.Number);
    end;
  Obj.Proc();
end;
```

Obviously, `Proc` is the culprit here and should be nilled to break the cycle. We can also go the other way around and `nil` the `Obj` reference, following a `try...finally` code pattern similar to the one we would use with a regular object instance. However, while calling `Free` on an object instance would not raise any eyebrows, nilling an interface reference just before it goes out of scope might. It would be prudent to comment such code to prevent someone from clearing it out by mistake, as it might seem like redundant cruft:

Correct code

```
procedure Test;
var
  Obj: IAnonymousObject;
begin
  Obj := TAnonymousObject.Create;
  try
    Obj.Number := 5;
    Obj.Proc := procedure
      begin
        Writeln(Obj.Number);
      end;
    Obj.Proc();
  finally
    Obj := nil;
  end;
end;
```

If you cannot wrap your head around why setting an interface reference to `nil` immediately before it goes out of scope can clear the above cycle, the secret is in the implicit code added by the compiler in the epilogue—another call to `_IntfClear` to finalize the local variable. Because of this, the reference count of the `Obj` instance will be decreased twice, and that is what breaks the cycle in this code.

The previously presented code not only has obvious flaws, but it is also quite pointless. Its purpose is not to be a pattern you would use, but to be a learning model.

A commonly used pattern used with anonymous methods is to not execute them immediately after defining them; they are usually either executed at a later time, or executed asynchronously. In that case, you cannot rely on a non-reference-counted object instance being alive when the anonymous method executes, and depending on the situation, it might not be possible to release that object from within the anonymous method.

In such cases you have to either use reference-counted object instances, or you must release the object from within the method; but in the latter case you must call such an anonymous method only once.

In terms of the previous example, if we imagine that the code within `Obj.Proc` was executed asynchronously under manual memory management, then calling `Obj.Free` after we started execution of `Obj.Proc` would release the `Obj` instance before the anonymous method had the chance to complete.

Such an asynchronous scenario can be created with `TTask`, from the RTL Parallel Programming Library:

```
type
  TProcessor = class(TObject)
  protected
    FData: TData;
    ...
  public
    procedure Run;
  end;

procedure TProcessor.Run;
var
  Task: ITask;
begin
  FData.Prepare;
  Task := TTask.Create(
    procedure
    begin
      FData.Process;
      FData.Processed := true;
    end);
  Task.Start;
end;
```

The above `Run` method will not create any strong cycles even though we are accessing `TProcessor`'s fields inside an anonymous method, because `Task` is a local variable. However, if for any reason we have to make that `Task` variable accessible to a broader context, even temporarily, we will have a strong reference cycle.

In such a case, we have to explicitly break the cycle at the appropriate point. In the following example, it is logical to clear the `FTask` field after the task has completed its job, from within the anonymous method:

```
type
  TProcessor = class(TObject)
  protected
    FData: TData;
    FTask: ITask;
    ...
  end;

procedure TProcessor.Run;
begin
  FData.Prepare;
  FTask := TTask.Create(procedure
    begin
      FData.Process;
      FData.Processed := true;
      FTask := nil;
    end);
  FTask.Start;
end;
```

15.7 Using weak references to break anonymous method reference cycles

A strong reference cycle is a strong reference cycle, regardless of how it is created. Breaking strong reference cycles created by anonymous methods is not too different from breaking other kinds of strong references. Besides explicitly breaking cycles at a certain point during execution, you can figure out which reference is the owning one and which one is not, and make the non-owning one weak, or instead of capturing a strong reference, you can capture a weak one.

Of course, capturing weak instead of strong references can only be used if the lifetime of the original reference is longer than the lifetime needed for the captured variable, or if the required functionality allows you to have `nil` checks inside the anonymous method and do nothing if the original reference is already gone.

If the above requirement cannot be satisfied, you will have to break the cycle by some other method: separating dependencies, limiting the scope of anonymous methods if you have more of them, using different transient variables for storing values that must be captured...

If you can break a cycle with weak references, great; you just have to use the standard methods of making weak references and you are done...

```
var
  [weak] Task: ITask;
begin
  Task := TTask.Create(
    procedure
    begin
      ...
      Task.CheckCanceled;
    end);
  Task.Start;
end;
```

Another type of strong reference cycle involving anonymous methods that can be solved by using the `[weak]` or `[unsafe]` attributes is recursion. If your anonymous method calls a reference to itself, it will create a strong cycle:

```
function Factorial(x: Integer): Integer;
var
  [weak] Func: TFunc<Integer, Integer>;
begin
  Func :=
    function(Value: Integer): Integer
    begin
      if Value = 0 then
        Result := 1
      else
        Result := Value * Func(Value - 1)
    end;
  Result := Func(x);
end;
```

Note: In older Delphi versions, there was a compiler bug that would turn a `[weak]` reference captured by anonymous method into a strong one, and the above code would leak because of it. This issue is partially fixed in 10.3 Rio, so the above example works correctly, but it still affects inline variable capture in 10.4 Sydney. If you are using a Delphi version affected by any variant of this bug, you can also use the `[unsafe]` attribute but then you must make sure you will not accidentally access a dangling reference. In the event of a inline variable leak, the easiest fix is to move the inline variable to local declaration.

QP report RSP-19204

This is a good place for another reminder that before establishing particular coding patterns, it is prudent to verify whether they work correctly in a particular Delphi version. There have been numerous improvements and fixes in the compiler over the years, so any example code that works correctly in the most recent versions might not work properly in older versions.

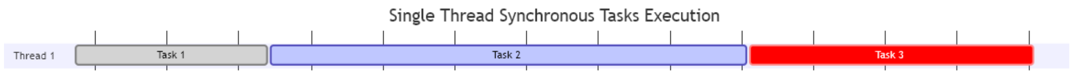
Part 4. Asynchronous Programming and Multithreading

Chapter 16

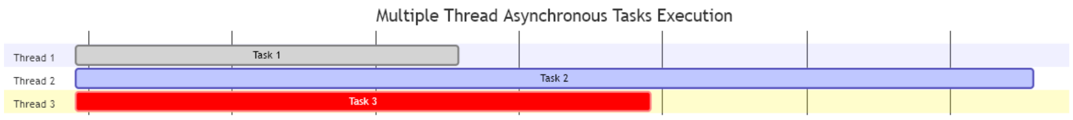
Asynchronous programming and multithreading

Asynchronous programming and multithreading are conflated quite often. While asynchronous programming commonly involves multiple threads, it is also possible to execute multiple tasks asynchronously in a single thread.

Synchronous programming consists of serially running some tasks, one by one. After one task completes, another one runs. In other words, while one task is running, all other tasks will wait in line for their turn.



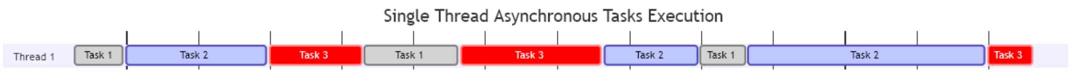
Asynchronous programming using multiple threads allows independent tasks to run in parallel—simultaneously—using different threads. How many threads will actually be able to run simultaneously depends on the machine and number of CPU cores available, but this will only have an impact on speed, not on how developers write their multithreading code—at least not in general. Depending on whether an application is a server or client application, and the expected specifications of the machine running it, multithreaded code can be slightly tweaked and optimized to run faster and better utilize the resources available to it.



The reason why asynchronous programming and multithreading are often used interchangeably is that the overwhelming majority of code uses one of the previously mentioned approaches. It either uses synchronous execution in a single thread, or uses multiple threads which by definition run asynchronously.

The third possibility, however, is to split tasks into multiple steps and asynchronously run one such step after another. This is a particularly useful approach for any non-CPU-bound operations—tasks that don't need to run on the CPU, but are using some other *slow* resource and will spend most of their time waiting for feedback from that resource. For instance, you can start a read operation on some file, and there is no need for your app to wait, twiddling its thumbs while the disk reads the requested data and returns with results. The same goes for doing network-related requests or interacting with other hardware devices. If you specify that a file is to be read in asynchronous mode, the file read function will return immediately, and you can do other tasks in the meantime. When the file read operation is completed, the OS will fire an event to notify your application that the requested data is ready. That event will sit in line, waiting to be processed along with other events. When its turn comes up, you can continue with the original task that issued the file read request.

Now, you can argue that the task that needs to read the file and process the data is actually two tasks combined, but you can also force your currently running task to yield control and run other pending tasks, all in the context of a single thread. In Delphi, there are several ways of achieving that, like, for instance, using the `TThread.ForceQueue` or `Application.ProcessMessages` methods.



The following example with `ForceQueue` shows splitting the task of counting (adding numbers to output) into single steps that will execute asynchronously. We can run more than one such task, and their output will be intertwined. Also, while the task runs asynchronously, it can be interrupted by other events like the user pressing the Cancel button:

```
var
  Canceled: boolean;

procedure TMainForm.AddLine(x, n: integer);
begin
  if (x > n) or Canceled then Exit;
  Memo1.Lines.Add(x.ToString);
  inc(x);
  TThread.ForceQueue(nil,
    procedure
    begin
      AddLine(x, n);
    end, 5);
end;
```

```
procedure TMainForm.RunButtonClick(Sender: TObject);
begin
  Canceled := False;
  AddLine(1, 100);
  AddLine(201, 300);
end;

procedure TMainForm.CancelButtonClick(Sender: TObject);
begin
  Canceled := true;
  Memo1.Lines.Add('Button Cancel');
end;
```

Similar functionality can be achieved by using `Application.ProcessMessages`. The main difference between this and the previous example is that the initial code in the `RunButtonClick` event handler will run synchronously, the counting tasks will run one after another, and the output results will not be intertwined like in the `ForceQueue` example.

```
procedure TMainForm.AddLine(x, n: Integer);
var
  i: integer;
begin
  for i := x to n do
    begin
      if Canceled then Exit;
      Memo1.Lines.Add(i.ToString);
      Application.ProcessMessages;
    end;
end;
```

Simply put, asynchronous (synchronous) programming is focused on tasks, and multithreading focuses on the workers—threads—that will run those tasks. Synchronicity tells you whether executing a particular task will block the current worker (thread) or not. And threading tells you how many workers you are using to run tasks. Singlethreaded—one worker, multithreaded—multiple workers. And of course, when multiple threads are involved, you will need to take care of thread safety, and make sure that those workers don't walk all over each other's feet and make a mess.

Chapter 17

Making the first dive...

Multithreading does not make your code run faster, it makes your code run slower.

Locking, synchronization, thread switching, all that extra code needed to make your code thread-safe takes time to execute, and will make your original code run slower, not faster.

So what is the point?

Every application, sooner or later, comes to a point where some operation will take more than a split second to run, and during that time it will block the main thread, leaving the application in an unresponsive state. I don't have to explain why having *Not Responding* glued to the title bar of your Windows application makes for a bad user experience, but it also goes beyond that. If the application is in such a state for too long, Windows will show a dialog to the user, offering to kill such an application (even though it might not be really dead, just really busy working on some long task). On some platforms like Android and iOS, the user might not even get the chance to decide what to do. The OS can decide to nuke such an unresponsive application at any time (usually sooner rather than later).

Besides offloading long-running tasks to the background threads to make the UI responsive, another purpose is spreading the workload across multiple cores, so that while the total execution time of all the threads combined will be longer, ideally each of them will run in less time than a single thread would need.

Ideally is the key word here. If the task is too small, it can potentially run faster in a single thread than the whole process of moving it into the background and sending the results back. However, the decision about moving to the background or not should depend on the worst-case scenario, not the best or even the most common one.

Some tasks are also suitable for a *divide and conquer* approach, and can be divided into subtasks that can run in parallel using multiple threads.

As processor evolution has shifted from a single-core processor to multiple cores, software nowadays is also making the necessary shifts toward multithreading and parallel algorithms in order to run faster.

Multithreading is not an easy topic to master. Because it is so complex with many pitfalls lurking around, many developers are reluctant to dive into those stormy waters. But once you make the dive, you will find out that it is not all so bad and that deep waters offer a calm not found on the surface.

Until the monsters from the deep suddenly appear and swallow you in one bite...

MWAHAHAHA...

Chapter 18

Application.ProcessMessages

Well, you have to start somewhere, and the logical place for transforming any singlethreaded application is finding long-running tasks and making the UI responsive in such places.

If an application is pumping the message queue, that is a signal that it is alive. If some long task blocks the main thread, there is no queue processing. It has been a long-standing practice in Delphi (and a really bad one) to call `Application.ProcessMessages` in such situations.

This is bad for many reasons. For starters, you need to find the right places to insert calls to `Application.ProcessMessages` inside your task code. If you call it too often, it will slow down actual work, but if you call it too infrequently, it will not be enough to prevent "not responding" messages or sluggishness.

Your code can also call other code, that other code might also call `Application.ProcessMessages`, and before you know it, you will spend most of the time pumping the message queue instead of performing the actual task.

Even if your code allows nice and even message processing, there is another pit that many naive Delphi developers have fallen into. While your code will look linear, it is not. `Application.ProcessMessages` makes your code re-entrant. That means that if you have started your task on a button click, and the user clicks that same button again, calling `Application.ProcessMessages` will process that click and call the button click handler in the middle of the code that is already handling the previous click. And not only that particular button will be affected. The user can trigger literally any other action in your application.

If you think that you can convince your users to keep their hands off the mouse and keyboard...

Well, first, that never works, and second, if you have any timers in your code, their events can also fire during a call to `Application.ProcessMessages` and make a fine mess.

On top of all that, in any more complex scenario, it will create a debugging nightmare. Actually, even simple scenarios are debugging nightmares.

Trust me, I know what I'm doing.

No, you don't...

If you don't fall into the reentrancy trap, and you don't care how much time you will spend debugging, sooner or later your application will choke on some task that will take way too long, and where you will not be able to do a thing about it.

But all of my tasks run fast and can be finetuned with `ProcessMessages`!

So you say...

Have you ever tried saving a file on a damaged drive, where the Windows API will choke on writing for more than 5 minutes and it will seem like it is trying to chisel the data in? Or have you ever encountered a network connection or server that is alive enough to prevent throwing a timeout exception, but so slow that using snail mail would be faster?

I don't care about exceptional situations, I just want to finish my new Android app!

Wait, what... did you say Android?

Well, sorry to break it to you, but using `Application.ProcessMessages` on Android is not supported and will deadlock. Not only that, but if you try to invoke any networking—read: Internet—related code from the main thread, the Android application will crash with `NetworkOnMainThreadException`.

The right thing to do is to move such long (or even potentially long) tasks into a background thread. That way, the application will remain responsive, and the OS will not be tempted to kill it before its time.

One important thing should be noted here: While using a background thread solves sluggishness and unresponsiveness when a long task is outside of our control, and where it is not possible to pump messages in small time intervals, it will not solve code reentrancy.

One of the reasons `Application.ProcessMessages` has been abused for so long by so many developers is that multithreading is not so easy. Not only does it require protecting access to shared data, but also adapting to a non-linear way of writing program logic. Add not solving the reentrancy problem to the above, and it's no wonder `Application.ProcessMessages` still appeals to so many people.

While changing to non-linear logic seems like a huge leap and represents a huge hurdle, in reality it is not all that hard. It is just an extension of the event-based programming every Delphi developer is already used to.

When the user clicks on a button, you can add an event handler that will execute some code on that click, and when the user clicks on another button, you will have a different event handler that will execute different code.

A similar logic works with background tasks. A task will run in the background, and when it is completed, it will trigger some event (completion) handler that will execute the appropriate code at that moment.

Chapter 19

Application.ProcessMessages reentrancy problem

Let's visualize different patterns and anti-patterns with some code.

This is the long task sequence we want to run on a button click:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  DoFoo;
  DoBar;
  DoFork;
end;
```

And we immediately run into another common Delphi coding misuse. All our logic is located in the button event handler. But, to simplify following the code flow and focus on the main differences between `Application.ProcessMessages` and background tasks, I will leave it like that.

The code in the above `ButtonClick` event handler is blocking. It will run sequentially, blocking execution of all other code—including processing Windows messages—and thus preventing us from processing user actions. Of course, if our application has some other code running in background threads, that code will still run, but we will assume that at this point we have a simple, singlethreaded application.

Because that code is blocking, clicking on the button will always result in the same output, no matter how many times you click in a row, nor how fast:

```
  Foo
  Bar
  Fork
```

Let's say that the above task is a long-running task, and you want to make the application responsive during that time by adding `Application.ProcessMessages`. Now your code may look like the following, and you may have some extra calls to `Application.ProcessMessages` inside any of the `DoFoo`, `DoBar` or `DoFork` methods—exactly how many you have does not really matter, because only a single call is sufficient to make a mess out of your sequence:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  DoFoo;
  Application.ProcessMessages;
  DoBar;
  Application.ProcessMessages;
  DoFork;
end;
```

If you click on the button only once, and let the code finish running before you click again, you will still get the correct output:

```
  Foo
  Bar
  Fork
```

But if you click on the button twice in a row, before the `DoFork` method has had the chance to run, you can get something like:

```
  Foo
  Foo
  Bar
  Fork
  Bar
  Fork
```

To prevent reentrancy, you can disable the button immediately after entering the button event handler, and enable it before exiting:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  Button.Enabled := False;
  try
    DoFoo;
    Application.ProcessMessages;
    DoBar;
    Application.ProcessMessages;
    DoFork;
  finally
    Button.Enabled := True;
  end;
end;
```

But, the above approach only works if clicking on that Button was the only entry point. If you can invoke this method in any other way—for instance through some menu—disabling the button will not be enough.

If you have other actions that should not be activated while the above event handler is running, it might be better to have an additional field in the form that will show what the current status of your operation is, whether it is running or not.

Having a `Processing` field does not exclude disabling the UI or showing the user in some other way that the application is currently working on something. Let's move all that UI-related code into separate methods. Make sure your UI-related methods don't raise exceptions or you will need to handle them separately, as the `finally` block in the `ButtonClick` handler will not be able to properly set the `Processing` variable.

```
procedure TMainForm.DisableUI;
begin
  Button.Enabled := False;
  // and any other related UI action
end;

procedure TMainForm.EnableUI;
begin
  Button.Enabled := True;
  // and any other related UI action
end;

procedure TMainForm.ButtonClick(Sender: TObject);
begin
  if Processing then
    Exit;

  Processing := True;
  try
    DisableUI;
    DoFoo;
    Application.ProcessMessages;
    DoBar;
    Application.ProcessMessages;
    DoFork;
  finally
```

```
      EnableUI;
      Processing := False;
    end;
  end;
```

You might be wondering, why are we still discussing `Application.ProcessMessages` related code if this is the wrong approach altogether? Well, if you move code to a background task, you will still need to solve the reentrancy problem, and you will still need to give some kind of feedback to the user. So the above code is just another step forward in moving your task to a background thread.

Another reason is that it is very likely that some will already have code similar to the above. Following the logical evolution of code that uses `Application.ProcessMessages`, step by step, will make it easier to recognize the above patterns in existing code and apply the changes needed to achieve the final goal of moving the task to the background.

Chapter 20

Moving long operations to a background thread

Just like the core Delphi frameworks neatly wrap the Windows messaging system and other complexities of OS interaction, the `TThread` class from the `System.Classes` unit wraps threading support and makes it easier to use. Besides the `TThread` class, newer Delphi versions also have the Parallel Programming Library in the `System.Threading` unit. And, of course, you can always use other third-party threading libraries, or make your own.

The simplest way to start moving long operations to a background thread is to use the `TThread.CreateAnonymousThread` class function, or to extend the `TThread` class, overriding its `Execute` method to implement your task code.

20.1 Using an anonymous thread

Now, let's move the example from the previous chapter to a background thread using the anonymous thread function. Looks rather simple, doesn't it? To be fair, moving code to a background thread is a simple task. Making sure that code runs properly in the context of a background thread—in other words, making it thread-safe—can be a completely different story.

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  TThread.CreateAnonymousThread(
    procedure
    begin
      DoFoo;
      DoBar;
      DoFork;
    end).Start;
  OutputDebugString('Button click exit');
end;
```

Since thread safety is a complex matter, for the moment we will pretend that our example doesn't contain any code that would not work properly in the context of a background thread. We will deal with thread safety issues later, for now we will focus on the code flow.

When executed, the above code will yield the following output, and as we can see, our button event handler will finish executing before the thread had a chance to run and perform its task:

```
Button click exit
Foo
Bar
Fork
```

`TThread.CreateAnonymousThread` creates and returns an instance of the internal class `TAnonymousThread`, that will execute the code provided in the anonymous method inside its execute method. It is created in suspended mode, so don't forget to call `Start` after creating it. Also, the thread is created with `FreeOnTerminate` set, so after you call `Start`, you should not do anything else with that thread if you have stored its reference in some variable.

With auto-destroying threads that have `FreeOnTerminate` set, not using a thread variable in any way after the thread starts running is a general rule valid for all threads. Since the thread will self-destruct, it is always possible that the thread has finished running and has already been destroyed at that point:

Incorrect code

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  Thread: TThread;
begin
  Thread := TThread.CreateAnonymousThread(
    procedure
    begin
      DoFoo;
      DoBar;
      DoFork;
    end);
  Thread.Start;
  Thread.OnTerminate := ThreadTerminatedCallback;
end;
```

Correct code

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  Thread: TThread;
begin
  Thread := TThread.CreateAnonymousThread(
    procedure
    begin
      DoFoo;
      DoBar;
      DoFork;
    end);
  Thread.OnTerminate := ThreadTerminatedCallback;
  Thread.Start;
  // It is not safe to use the Thread variable from this point on
end;
```

20.2 Extending the TThread class

Creating a custom extension of the TThread class is not much harder than using an anonymous thread, even though it requires way more ceremony. If there are no particular reasons to create a new class, like storing additional fields or any other functionality that might be impossible or hard to achieve using an anonymous thread, there is not much reason not to use anonymous threads as the simplest solution.

```
type
  TFooBarForkThread = class(TThread)
  protected
    procedure Execute; override;
  public
    constructor Create;
  end;

constructor TFooBarForkThread.Create;
begin
  // start automatically, unlike an anonymous thread,
  // which is constructed in suspended mode
  inherited Create(False);
  FreeOnTerminate := True;
end;
```

```
procedure TFooBarForkThread.Execute;
begin
  DoFoo;
  DoBar;
  DoFork;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
var
  Thread: TFooBarForkThread;
begin
  Thread := TFooBarForkThread.Create;
end;
```

If you fully configure everything in your thread class, and the thread has its `FreeOnTerminate` set, you don't even need a variable:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  TFooBarForkThread.Create;
end;
```

While creating and running our imaginary FooBarFork thread has been a resounding success, our troubles are just starting.

The first issue that may arise is that the `Do...` methods are not standalone procedures, but belong to `TMainForm`. In that case, it is probably time to separate UI and business logic, and move them into their own classes—or make standalone procedures, whichever is more suitable in a particular case. However, this move is only necessary when extending the `TThread` class, because using an anonymous thread can access those main form methods.

If you are refactoring existing code, and need to move slowly, using anonymous threads will be easier to refactor. Once you have all the nuts and bolts figured out, it will also be easier to take additional steps and make a total separation.

20.3 Thread reentrancy

Threads, just like `Application.ProcessMessages`, suffer from reentrancy. We solved that by disabling and reenabling the UI, and having an additional `Processing` flag. Now we need to make the same adjustments to our multithreaded code.

The first part—disabling the UI—is easy, but reenabling it again is trickier. An additional problem is that any UI-related code must run in the context of the main thread:

Incorrect code

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  if Processing then
    Exit;

  Processing := True;
  try
    DisableUI;

    TThread.CreateAnonymousThread(
      procedure
      begin
        DoFoo;
        DoBar;
        DoFork;
      end).Start;

  finally
    EnableUI;
    Processing := False;
  end;
end;
```

The above code is obviously wrong, because the `finally` block will execute while the thread is still running, and that is not what we need.

We could dump that anonymous thread approach and use a slightly modified `TFooBarForkThread`, remove `FreeOnTerminate` so we can freely access the thread variable, and just wait for the thread to finish before manually releasing it:

Incorrect code

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  Thread: TFooBarForkThread;
begin
  if Processing then
    Exit;

  Processing := True;
  try
```

```
    DisableUI;
    Thread := TFooBarForkThread.Create;
    try
      Thread.WaitFor;
    finally
      Thread.Free;
    end;
  finally
    EnableUI;
    Processing := False;
  end;
end;
```

Err… this works, kind of… but, `WaitFor` is a blocking call and will block the main thread, which is exactly the thing we are trying to prevent in the first place.

So, forget about waiting…

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  if Processing then
    Exit;
  Processing := True;
  DisableUI;
  TThread.CreateAnonymousThread(
    procedure
    begin
      try
        DoFoo;
        DoBar;
        DoFork;
      finally
        TThread.Synchronize(nil,
          procedure
          begin
            EnableUI;
            Processing := False;
          end);
      end;
    end).Start;
end;
```

Now, that works better, but it is getting really ugly. Besides being ugly, we also have one more tiny problem. If the thread construction code fails for some reason, the UI will never be reenabled.

You can handle that part by wrapping the whole `TThread.CreateAnonymousThread...` in a `try...except` block, and handling any exception if it occurs. This is also a good place to note that the `Application.ProcessMessages` variant just had a `try...finally` block where the raised exception was not handled, but we only cared about the UI part and the `Processing` flag.

If you want to properly handle an exception there, and show an appropriate message to the user rather than some unparsable error dialog, you will also need a `try...except` block there, as shown a few sections later in *Handling thread exceptions*, or you can assign your exception handler on the application level.

20.4 `OnTerminate` event handler

Another way of getting feedback when a thread is finished with its work is using the thread's `OnTerminate` event handler. It will execute in the context of the main thread, so no additional synchronization will be necessary. It can be used with both anonymous and custom thread classes, you just need to make sure it is assigned before the thread starts running:

```
procedure TMainForm.ThreadTerminateEvent(Sender: TObject);
begin
  EnableUI;
  Processing := False;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
var
  Thread: TThread;
begin
  if Processing then
    Exit;

  Processing := True;
  DisableUI;

  Thread := TThread.CreateAnonymousThread(
    procedure
    begin
      DoFoo;
      DoBar;
      DoFork;
    end);
  Thread.OnTerminate := ThreadTerminateEvent;
  Thread.Start;
end;
```

While the above code is easier to follow because there is no callback nesting, a `ThreadTerminateEvent` that is separated from the main `ButtonClick` method is a bit harder to follow than code that has all the pieces in one place. For simplicity, this code has not implemented any of the required exception handling.

Eventually, you will have to decide which kind of code works best for you. Keeping code organized in logical layers also helps to track what is happening where. `DoFoo`, `DoBar` and `DoFork` already achieved that, but if they can be combined together in an additional method that makes a logical whole, it would be prudent to do so.

20.5 Handling thread exceptions

Handling exceptions in multithreading can get a tad ugly. First you need to take care of exceptions raised during thread construction, and then you need to take care of exceptions raised within the thread's execute method itself.

We already know how to handle construction exceptions... or at least any Delphi developer should. But, a short reminder never hurts.

Exceptions are handled with `try...except` blocks, and `try...finally` blocks are used for cleanup code that must always run, regardless of whether an exception is raised or not:

```
try
  DoFoo;
  DoBar;
  DoFork;
except
  // this code is executed only if an
  // exception is raised in the try part
  HandleException;
end;
```

```
try
  DoFoo;
  DoBar;
  DoFork;
finally
  // this code is always executed regardless of
  // whether an exception was raised in the try block or not
  DoCleanup;
end;
```

If no exceptions were raised, the output of the above code would be:

```
try...except

Foo
Bar
Fork

try...finally

Foo
Bar
Fork
Cleanup
```

And if some method raised an exception (for instance, `DoBar`), the output would be:

```
try...except

Foo
Handle Exception

try...finally

Foo
Cleanup
```

Unless you explicitly raise a new exception or re-raise the existing one, a raised exception will never propagate outside the *except* block. In other words, the outside code will no longer be affected by the exception and will continue to run in regular order. And in the case of the *finally* block, after the code in `finally` runs, the exception will always be propagated further until the first outer `try...except` block that can handle it.

Note: On Delphi platforms where the LLVM compiler backend is used, hardware exceptions can be caught by `try...except` blocks only if they are raised within a function or method. In other words, if they are raised directly within the `try...except` block, they will be propagated further up to the next exception handler (but not the immediate one, rather the one that wraps the whole *broken* `try...except` block in the function or method).

If you are using an anonymous thread, or any other thread with `FreeOnTerminate` set, you don't need to perform any cleanup around constructing that thread instance, but you will need to handle reenabling the UI (if it was previously disabled) and eventually showing the user an appropriate message:

```
  try
    Thread := TThread.CreateAnonymousThread(
      procedure
      begin
        DoFoo;
        DoBar;
        DoFork;
      end);
    Thread.OnTerminate := ThreadTerminateEvent;
    Thread.Start;
  except
    EnableUI;
    Processing := False;
    ShowErrorMessage;
  end;
```

But, what happens if an exception is raised within the thread—for instance, if `DoBar` raises an exception?

If that happens, that exception will be caught by the exception handler put around the `Execute` call within the `TThread` class. If you leave the above code as-is, you will not even know that there was an exception, unless you were running the code through the debugger. Or more correctly, you may notice that something is wrong, because `DoBar` may have been interrupted halfway through, and `DoFork` never had a chance to run. But that is it. The thread's `OnTerminate` event will still run like nothing bad happened.

So how to solve this problem? You can put the handling code directly in the thread's `Execute` method, or the anonymous method if you are using an anonymous thread, but you will also need to synchronize showing any error message with the main thread.

Fortunately, there is a better option that produces cleaner code. The previously mentioned exception handler around the `Execute` call within the `TThread` class will take the raised exception and store it into its `FatalException` property. You can examine that property in the `OnTerminate` event handler, and if it is assigned, you will not know only that something bad happened, you will also know what it was, and you can notify the user accordingly:

```
procedure TMainForm.ThreadTerminateEvent(Sender: TObject);
var
  E: TObject;
begin
  EnableUI;
  Processing := False;
  E := TThread(Sender).FatalException;
```

```
    if Assigned(E) then
      begin
        if E is Exception then
          Application.ShowException(Exception(E))
        else
          ShowMessage(E.ClassName);
      end;
  end;
```

The fact that `OnTerminate` will always run, also makes it a good place to perform any necessary cleanup. But, whatever you need to do in the `OnTerminate` event handler, make sure that you catch and handle all exceptions there, because an unhandled exception in the `OnTerminate` event handler can bring your whole application down.

When you are making proof of concept code and testing rough designs, it is always good to explicitly raise exceptions in places where you are not completely sure how that part of code will behave and whether you have properly handled all of the necessary code paths.

Besides executing cleanup code directly in the `OnTerminate` event, it can be used only as a pit stop from where you will use some other notification system and notify the application that executing some particular piece of code is necessary.

20.6 Canceling the task

If you don't care about canceling in-progress tasks—after all, our `Application.ProcessMessages` code variant didn't care either—then you don't need to read this... I am not sure that your users will approve, though.

Canceling tasks is not an easy thing to do, no matter which approach you take. For some short tasks, you really don't need to bother with canceling. Anyone can wait a few seconds, but for anything longer, canceling is more than nice to have.

Canceling tasks is not a multithreading problem. It is equally hard to cancel a task in code with `Application.ProcessMessages` as it is to cancel a task executing in a background thread. Of course, if you don't use either—and execute your code in a blocking manner—you won't be able to cancel the task at all. Simply put, you will have no means to process user input and act accordingly.

Canceling the task is not very complicated to implement, but it uses rather primitive mechanisms. Also, for more complex code, the ability to cancel some operation on a higher level is directly related to the ability to cancel other operations on lower levels.

Canceling the task requires an additional flag, and some UI control that will set that flag:

```
procedure TMainForm.CancelButtonClick(Sender: TObject);
begin
  Canceled := True;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
begin
  Canceled := False;

  TThread.CreateAnonymousThread(
    procedure
    begin
      if Canceled then Exit;
      DoFoo;
      if Canceled then Exit;
      DoBar;
      if Canceled then Exit;
      DoFork;
    end).Start;
end;
```

Basically, the above is how you implement canceling. If, for instance, `DoFoo` is a long-running task with plenty of code, you will need to add checks for the `Canceled` flag in there, too.

If you are not using anonymous threads, you can also use the thread's `Terminated` flag to check whether to exit the `Execute` method or not, and call the thread's `Terminate` method to terminate the thread.

```
procedure TFooBarForkThread.Execute;
begin
  if Terminated then Exit;
  DoFoo;
  if Terminated then Exit;
  DoBar;
  if Terminated then Exit;
  DoFork;
end;

procedure TMainForm.CancelButtonClick(Sender: TObject);
begin
  Thread.Terminate;
end;
```

Of course, instead of calling `Exit`, you can also raise some custom `ETaskCanceled` exception.

Since our code is rather simple and our `Do...` methods currently belong to a form (or are standalone procedures), currently the only way to cancel the task they perform is to use a global `Canceled` flag they can also access and check. Real-life code should be more self-contained in an object or record instance, and not use global state. Another way is to use some framework that will give you the ability to define tasks and provide all of the necessary functionality.

Delphi XE7 introduced the Parallel Programming Library (PPL) in the `System.Threading` unit, which, among other things, implements task support. More about tasks in general, as well as the PPL, can be found in the *Tasks* chapter.

Note: Whatever you do, don't ever use the WinAPI `TerminateThread` function to forcefully terminate a thread. This function is extremely dangerous and will not only leave your application in an unstable state, but it can also have a negative impact on the rest of the system. Basically, after you call `TerminateThread`, not only do you need to kill and restart your application, but the whole OS needs to be restarted, too.

20.7 Passing the data

At this point, you may think that our threading troubles are over, and that we have covered all the appropriate coding patterns for moving tasks into background threads, but we have only scratched the surface of what is needed to really move some task to a background thread. What we have already covered may be sufficient for some, but usually tasks involve some data and we haven't touched that part, yet.

Sharing data requires knowing how to do that in a thread-safe manner. This is a complex topic in its own right, and is covered in *Part 5. Thread Safety*, where you can also find more thorough examples for thread-safe data sharing.

The following examples focus more on the mechanics of sharing data with background threads than thread safety, so if you use those examples as a template for your code, make sure that you have considered thread safety and that you prevented simultaneous access to the shared data.

If you are transitioning from `Application.ProcessMessages` and you have data locally stored in form fields, you can use those directly in a background thread, as long as you have some mechanism that prevents that data from being used simultaneously. Since we have the `Processing` flag for preventing reentrancy, we can use it as a poor man's lock to protect access to the data. Keep in mind that accessing includes both reading and writing in the context of the following example.

In the following example, only `DoFoo`, `DoBar`, `DoFork` and `DisplayData` are allowed to access the `FData` object. And only in the places shown in the example. In other words, those methods can be called only when the `Processing` flag is set, and `DisplayData` can be called only from the context of the main thread, because it must interact with the UI controls:

```
type
  TMainForm = class(TForm)
  ...
  private
    FData: TDataObject;
    Processing: boolean;

    procedure DoFoo;
    procedure DoBar;
    procedure DoFork;

    procedure DisableUI;
    procedure EnableUI;
    procedure DisplayData;
  end;

procedure TMainForm.DoFoo;
begin
  // change content of FData
  ...
end;

// DoBar and DoFork can also work on FData like DoFoo does
...

procedure TMainForm.DisableUI;
begin
  Button.Enabled := False;
  ...
end;

procedure TMainForm.EnableUI;
begin
  Button.Enabled := True;
  ...
end;

procedure TMainForm.DisplayData;
begin
  // present content of FData through UI controls
  ...
end;

procedure TMainForm.ThreadTerminateEvent(Sender: TObject);
```

```
begin
  DisplayData;
  EnableUI;
  Processing := False;
end;

procedure TMainForm.ButtonClick(Sender: TObject);
var
  Thread: TThread;
begin
  if Processing then
    Exit;

  Processing := True;
  DisableUI;

  Thread := TThread.CreateAnonymousThread(
    procedure
    begin
      DoFoo;
      DoBar;
      DoFork;
    end);
  Thread.OnTerminate := ThreadTerminateEvent;
  Thread.Start;
end;
```

Another option is using local data that is thread-safe, so as long as you create an instance of some object or have other kind of data locally, you can safely use it within the thread. What is not safe is taking references to other data:

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  Data: TDataObject;
begin
  Data := TDataObject.Create;
  Data.InitData(...);
  TThread.CreateAnonymousThread(
    procedure
    begin
      DoFoo(Data);
      DoBar(Data);
      DoFork(Data);
      TThread.Synchronize(nil,
        procedure
        begin
          DisplayData(Data);
          Data.Free;
        end);
    end).Start;
end;
```

If you are using a custom `TThread` class, you can store data as a field of that class, set it through the constructor, and present it to the UI in the `OnTerminate` event.

20.8 Pitfalls

The main purpose of this chapter is to show a step-by-step transition from using `Application.ProcessMessages` to background threads, and covering some basic thread mechanisms. Generally, the code examples shown will be an adequate substitute to `Application.ProcessMessages`, but they also suffer from an additional set of issues that you need to be aware of, and that may require a different approach.

Some issues arise from the need to safely share data—which can be mitigated by avoiding simultaneous access to shared data, synchronization and locking mechanisms. Threading also requires a slightly more thoughtful approach to exception handling, but probably the most devious problem is notifying the UI after the thread has finished running.

You are probably confused now. How is notifying the UI an issue, when the above examples show how to do that?

Well, the above examples work well, unless you literally *pull the rug out from under the thread's feet*. The rug being the UI, here. What can easily happen is that the user can destroy the UI (close the form or application) while the thread is still working in the background.

If that happens, all kinds of exceptions can happen during application shutdown, and in such situations, the integrity of the data can also be compromised. Similar scenarios can happen in singlethreaded applications, but threads introduce an additional level of complexity, and if you are creating and destroying parts of the UI while it is being shown and hidden from the user, you can have a more widespread problem and a more serious one than if you were using `Application.ProcessMessages`:

```
procedure TMainForm.ThreadTerminateEvent(Sender: TObject);
begin
  EnableUI;
end;
```

The main problem arises in the above code—imagine you have destroyed the form or frame or any of the UI controls used in the `EnableUI` method, and the thread is still not finished when you did that. After the thread finishes, it will call its `OnTerminate` event handler and bang, it will try to do something with invalid pointers, namely references to already-released UI controls.

There are several ways to handle such situations: The first is to prevent the user from terminating and closing the UI while the thread is running. Another one is to initiate the shutdown process, but then wait for all threads to finish before you kill the UI.

Another way is to use messaging systems to avoid direct communication between the GUI and threads. If you close parts of the UI, the thread can still finish the job, and fire a notification, but if the UI part is no longer there listening, nothing bad will happen. Of course, this only works while the application is still running and the user is just closing parts of the UI, not the whole application.

If the user closes the application, your only option is to wait for threads to gracefully exit. If you have implemented canceling tasks, you can also use that to speed up the process.

A more detailed explanation and examples about communication with the GUI can be found in *Part 6. GUI and Multithreading*.

Chapter 21

Asynchronous message dialog

One of the more common patterns and coding practices, that has worked well for Delphi Windows applications, is using modal dialogs. A modal dialog is a dialog (window) that forces the user to interact with that window, choose some option, or simply close it, before the application can continue running. Another aspect of that modality, is that invoking a modal window is a blocking call. Your code will execute in a linear fashion and wait for user input.

That allows you to write code like:

```
procedure TMainForm.OpenButtonClick(Sender: TObject);
begin
  if OpenDlg.Execute then
    begin
      // do something with selected file
    end;
end;
```

```
procedure TMainForm.DeleteButtonClick(Sender: TObject);
begin
  if MessageDlg('Are you sure you want to delete the selected item?',
    mtConfirmation, [mbOK, mbCancel], 0) = mrOK then
    begin
      Items.DeleteSelected;
    end;
  // Do something in any case after dialog is closed
  ...
end;
```

```
procedure TMainForm.DeleteButtonClick(Sender: TObject);
begin
  if MessageDlg('Are you sure you want to delete the selected item?',
   mtConfirmation, [mbOK, mbCancel], 0) <> mrOK then Exit;

  Items.DeleteSelected;
  ...
end;
```

But, with the introduction of other platforms, the ability to use modal dialogs has changed a bit. Namely, Android as an operating system does not have the same concept of modal dialogs as Windows does. On Windows, invoking modal dialogs is a blocking call. Subsequent code in the main thread will not execute until the dialog is closed and returns a result. On Android, showing a dialog (while you can make it modal in the sense that the user must interact with the dialog before they can continue using the application) will not be a blocking call, all code will just continue execution, and the dialog will return a result through an event handler.

In practical terms, you can no longer write code involving modal dialogs the way you used to. You cannot have a simple function that will return some result to you, and where you can proceed depending on that result. You need to use a completion handler (callback)—an anonymous method that will execute after the user selects some option and the dialog closes.

Depending on the platform, such a message dialog can execute synchronously or asynchronously, but the completion handler will run in context of the main thread regardless, so there is no need to think about thread safety:

```
procedure TMainForm.DeleteButtonClick(Sender: TObject);
begin
  Log.D('Button click enter');
  TDialogService.MessageDialog(
    'Are you sure you want to delete the selected item?',
    TMsgDlgType.mtConfirmation, [TMsgDlgBtn.mbOK, TMsgDlgBtn.mbCancel],
    TMsgDlgBtn.mbOK, 0,
    procedure(const AResult: TModalResult)
    begin
      // this handler runs in the context of the main thread
      Log.D('Completion handler');
      if AResult = mrOK then
        begin
          Items.DeleteSelected;
        end;
    end);
  Log.D('Button click exit');
end;
```

There is also the option of changing the default platform behavior and explicitly specifying whether such a dialog will run synchronously or asynchronously, using the `TDialogService.PreferredMode` class property. Since Android does not support synchronous (blocking) dialog execution, it will always run asynchronously regardless of the setting.

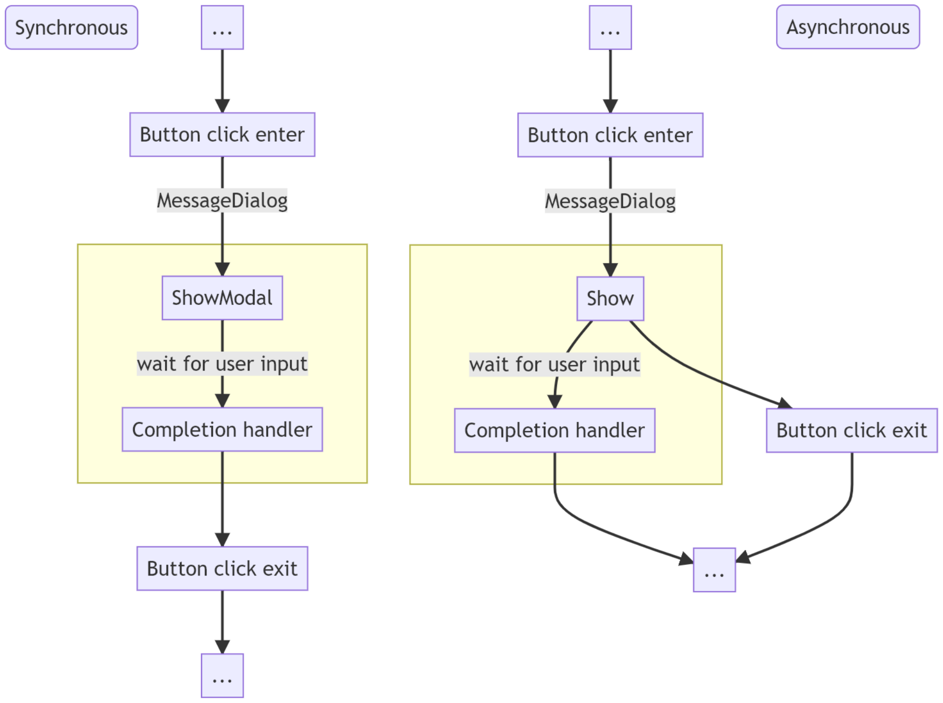
When the above example is executed in synchronous mode, the button event code will execute in a synchronous, linear manner, and the log output will be:

```
Button click enter
Completion handler
Button click exit
```

However, in asynchronous mode, the button event code will completely run before the dialog completion handler has had a chance to run:

```
Button click enter
Button click exit
Completion handler
```

The obvious difference in execution order between synchronous and asynchronous dialogs should be taken into consideration when coding for multiple platforms, or when overriding the default behavior on Windows by forcing the dialog to run asynchronously.



Chapter 22

Tasks

As previously mentioned, using multithreading does make your code run slower. Creating and destroying threads is a rather slow process. If you have tasks that create and destroy many rather short-lived threads, reusing those threads and just feeding them with new tasks would be a faster approach than recreating them.

This is the basic idea behind any task-oriented frameworks. Such frameworks consist of several basic parts: tasks that wrap task code and other task-related data and status information, a task queue for managing a queue of tasks, and a thread pool that provides reusable threads that can run tasks as they come to the queue.

The Parallel Programming Library (PPL) in `System.Threading` implements such a task-oriented framework, and the following code examples use PPL-provided classes and interfaces, while the explanations about the workflow and principles are not only related to PPL, and are applicable to task-based frameworks in general, even though some finer implementation details may differ.

The following is a task-based equivalent of the example used in the `Application.ProcessMessages` chapter:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  if Processing then
    Exit;

  Processing := True;
  DisableUI;

  TTask.Run(
    procedure
    begin
      try
```

```
          DoFoo;
          DoBar;
          DoFork;
        finally
          TThread.Synchronize(nil,
            procedure
            begin
              EnableUI;
              Processing := False;
            end);
        end;
      end);
  end;
```

The above code is very similar to TThread.CreateAnonymousThread, but there are significant differences in what happens in the background. Using an anonymous thread will create a new thread every time its code runs; the same is true for the example with the extended TThread class that is created on the spot. On the other hand, the above TTask code will create a new task object instance and add it to the queue, and if there are idle threads sitting around in the thread pool, the task will be able to run faster, because creating and starting a new thread instance is a time-consuming process.

Instead of an anonymous method, you can also use TNotifyEvent and the Sender object in the task factory function. The TaskData passed in the following example can be a locally created object, or some other data object accessible at that point. If your task does not need any data, and does not use the Sender parameter, you can also safely pass nil:

```
procedure TMainForm.TaskProc(Sender: TObject);
begin
  try
    DoFoo;
    DoBar;
    DoFork;
  finally
    TThread.Synchronize(nil,
      procedure
      begin
        EnableUI;
        Processing := False;
      end);
  end;
end;
```

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  if Processing then
    Exit;

  Processing := True;
  DisableUI;

  TTask.Run(TaskData, TaskProc);
end;
```

22.1 Thread pool

A thread pool can dynamically adjust the number of worker threads in the pool, depending on the workload, so there is always a possibility that running some task will also require creating a new thread, but when there is a steady load of tasks coming in, any thread-pool-based solution will perform better.

One of the problems that arise in multithreading, is optimizing the usage of the finite resources available on the computer. This is one place where more is not necessarily better. If you have too many threads running at some point, the context switching between those threads, in order to get their share of CPU time and run some work, has a negative impact on performance, and having fewer threads can sometimes actually do the job faster.

Even if you don't care much about the initial cost of creating threads, the cost of handling many threads under a heavy workload will be too high.

That does not mean that you need to choose one approach over the other. You can apply different approaches in different parts of your application, depending on their particular needs. Using tasks is good for CPU-intensive tasks, where a thread will actually do some work and not wait too long for some I/O operation. If you have too many active threads with attached tasks that are just waiting for something, you can starve your pool resources, and tasks that could actually run in the meantime will be just sitting in the queue, waiting for a better moment.

Having dedicated threads for slow, non-CPU-bound tasks, is a more suitable option. Of course, if you create too many of those you might get in trouble again. If the application also has a lot of such slow tasks, you can also use a dedicated thread pool for running such tasks instead.

The algorithms used to dynamically adjust the number of worker threads use a set of predetermined parameters to perform their calculations. Each thread pool can be configured with its own parameter set, so they can better adapt to the specific workloads we know they will have to serve.

22.2 Managing task lifetime

So far, writing task-related code is quite similar to writing thread-related code, especially when using anonymous methods. Just as you may need to access the thread instance inside the body of an anonymous method, you may need to do the same with the task instance.

But, unlike threads, tasks are reference-counted objects, and that fact has some repercussions on their memory management and variable capture, where you can accidentally create reference cycles. On the other hand, you don't have to worry about accessing an already-destroyed object instance, unless you have been using the `[weak]` or `[unsafe]` attribute on your task variables.

The following example creates such a reference cycle, and the task instance will never be released:

Incorrect code

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  Task: ITask;
begin
  Task := TTask.Create(
    procedure
    begin
      DoBar;
      Task.CheckCanceled;
      DoFoo;
      Task.CheckCanceled;
      DoFork;
    end);
  Task.Start;
end;
```

Because the `Task` variable in the following example is used only within the task code, while we are absolutely certain that the task object instance is alive, we can safely use the `[unsafe]` attribute to break the strong reference cycle:

Correct code

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  [unsafe] Task: ITask;
begin
  Task := TTask.Create(
    procedure
    begin
      DoBar;
      Task.CheckCanceled;
      DoFoo;
      Task.CheckCanceled;
      DoFork;
    end);
  Task.Start;
end;
```

Another way of breaking the cycle is explicitly setting the `Task` variable to `nil` at the point we no longer need it, or at the end of the task's anonymous method:

Correct code

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  Task: ITask;
begin
  Task := TTask.Create(
    procedure
    begin
      DoBar;
      Task.CheckCanceled;
      DoFoo;
      Task.CheckCanceled;
      DoFork;
      Task := nil;
    end);
  Task.Start;
end;
```

22.3 Waiting for task

Just like you can wait for a thread to finish its work, you can wait on a task. If you have multiple tasks, you can wait on any one of them, or all of them.

Just like waiting for threads is a blocking call, waiting for tasks is also blocking. If you cannot avoid waiting, you should run that kind of code from an additional thread or task to keep the UI responsive:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      t1, t2, t3: ITask;
    begin
      t1 := TTask.Run(
        procedure
        begin
          OutputDebugString('TASK 1 RUNNING');
          Sleep(1000);
          OutputDebugString('TASK 1 COMPLETED');
        end);
      t2 := TTask.Run(
        procedure
        begin
          OutputDebugString('TASK 2 RUNNING');
          Sleep(2000);
          OutputDebugString('TASK 2 COMPLETED');
        end);
      t3 := TTask.Run(
        procedure
        begin
          OutputDebugString('TASK 3 RUNNING');
          Sleep(500);
          OutputDebugString('TASK 3 COMPLETED');
        end);

      TTask.WaitForAll([t1, t2, t3]);
    end).Start;
end;
```

22.4 Handling task exceptions

If you wait for a task to finish, and that task raises an exception, you might be unpleasantly surprised when your `Wait...` call raises an exception, too. That is different behavior from a thread, where waiting alone does not re-raise any exceptions. Since checking whether the task is canceled within the task code also raises an exception if the task is canceled, waiting for a canceled task will also raise one.

You can always handle exceptions within each task's anonymous method body to prevent an exception while waiting, but except for some specific scenarios, a better place is to handle them all together when waiting for completion. This approach will simplify code, remove unnecessary repeating, and allow you to propagate exceptions or show them to the user.

Because each task can raise an exception independently, `Wait...` will combine them all under an `EAggregateException`, which allows you to iterate through its InnerExceptions and inspect them separately:

EAggregateException with message 'One or more errors occurred'.

```
try
  TTask.WaitForAll([t1, t2, t3]);
except
  on E: EAggregateException do
    begin
      for i := 0 to E.Count - 1 do
        begin
          // handle specific exceptions E.InnerExceptions[i]
        end;
    end;
end;
```

22.5 Canceling the task

Unlike threads, tasks directly implement canceling via the `Task.Cancel` method. All you need to do is periodically call `Task.CheckCanceled` in the task body (where `Task` is the task variable), and if the task is canceled, this check will raise:

EOperationCancelled with message 'Operation Cancelled'.

If you are waiting on a single task with `Task.Wait`, then `EOperationCancelled` will be the actual exception raised by the `Wait` method. If you are waiting for multiple tasks, then an

EAggregateException will be raised, and if you need to know whether an operation was canceled or not, you will not be able to do that based on the combined exception alone.

The feature that the PPL is missing is a feature called a *cancellation token*, that can be used to cancel multiple tasks with the same token, and as an additional separation of concerns. Just like tasks add better abstraction and decoupling over the TThread class that just stuffs everything needed—code, data, and thread handling—into a single class, adding separate cancellation (and continuation) tokens would significantly simplify and improve code using the PPL.

Calling Task.CheckCanceled creates a bit of a "chicken or the egg" problem. If you use factory functions that automatically start the task, it is possible to run into a situation where CheckCanceled will be called on a nil reference, because the assignment to the Task variable was not yet completed:

Inorrect code

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  Task: ITask;
begin
  Task := TTask.Run(
    procedure
    begin
      // it is possible that Task is not yet assigned here
      Task.CheckCanceled;
      ...
      Task := nil;
    end);
end;
```

In such cases, you should first create the task using the Create factory function instead of using Run, and then call Start on the already-assigned Task variable. That way, when Task.CheckCanceled is executed inside the anonymous method, the Task variable will always hold the correct value:

Correct code

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  Task: ITask;
begin
  Task := TTask.Create(
    procedure
```

```
      begin
        Task.CheckCanceled;
        ...
        Task := nil;
      end);
    Task.Start;
  end;
```

22.6 Passing the data

When creating a custom thread class, you could also add fields in that class for handling necessary data. If you were using anonymous threads, you could capture data from an outside context. With tasks, using the field approach—while possible—is rather convoluted. The `TTask` class is not meant to be used as a classical inheritable class, so while you can technically make your own class that inherits from `TTask`, any pain you may face along the way will be purely self-inflicted. So unless you have a really compelling reason to do so, don't make your own `TTask` class extensions, and especially not if all you want to do is dump some data inside.

Anonymous method capturing for actual data works the same for tasks and threads, so you can use the same approach you used with threads. Using tasks with `TNotifyEvent` and the `Sender` object gives you the ability to have a separate class that will wrap all your data with processing (task) code, and use those instead of an anonymous method. You just need to pay attention to your data object instance's lifetime, and make sure that it will live longer than the associated task instance.

Just like data handling with threads needs to consider potential thread safety issues, data handling with tasks is no different. How to handle thread safety, and how to communicate with the UI thread, is covered in *Part 5. Thread Safety* and *Part 6. GUI and Multithreading*.

22.7 Parallel Join & For

All tasks can run in parallel (simultaneously) depending on the available hardware resources, but setting up multiple tasks can be a bit verbose. If you want to streamline configuring separate tasks that can run in parallel, you can use the `TParallel.Join` factory functions. This way you will get a single joined task as a result, which you can then use instead of handling multiple task variables.

To simplify the code, the following example will perform waiting on the main thread and block the UI, just like any other `Wait` call. To prevent blocking the UI, you will need to run such code in a background thread or another task.

Just like `TTask.Run` automatically starts the task, `TParallel.Join` does the same. This can also create a problem with `Task.CheckCanceled`, because `Task` could be `nil` if you check it too soon in the process. You can use `Assigned(Task)` to avoid potential problems:

```
procedure TMainForm.JoinButtonClick(Sender: TObject);
var
  Task: ITask;
begin
  Task := TParallel.Join([
    procedure
    begin
      OutputDebugString('TASK 1 RUNNING');
      if Assigned(Task) then Task.CheckCanceled;
      OutputDebugString('TASK 1 COMPLETED');
    end,

    procedure
    begin
      OutputDebugString('TASK 2 RUNNING');
      if Assigned(Task) then Task.CheckCanceled;
      OutputDebugString('TASK 2 COMPLETED');
    end,

    procedure
    begin
      OutputDebugString('TASK 3 RUNNING');
      if Assigned(Task) then Task.CheckCanceled;
      OutputDebugString('TASK 3 COMPLETED');
    end]);
  try
    Task.Wait;
  except
    // handle task exception
  end;
  // break reference cycle
  Task := nil;
end;
```

Often, you could also split some iterational, computation-intensive task that uses the same algorithm into multiple smaller ones, in order to speed up computations that are suitable for parallel execution. To simplify code setup for such parallel algorithms, the PPL has additional functionality wrapped up in the `TParallel` class in the form of various overloads for class `&For` iterator functions, that return a `TLoopResult` record containing information about whether all loop iterations have successfully completed, and if not, which is the lowest index that broke the loop.

```
  TLoopResult = record
  private
    FCompleted: Boolean;
    FLowestBreakIteration: Variant;
  public
    property Completed: Boolean read FCompleted;
    property LowestBreakIteration: Variant read FLowestBreakIteration;
  end;
```

`TParallel.&For` is a blocking call, and will block the current thread until all iteration tasks are completed, or an exception is raised in one of them. To prevent blocking the main thread, you should use `TParallel.&For` from background threads, or within another task.

`TParallel.&For` simulates `for`-loop iteration, but besides the lower and upper bound, you can also specify a stride parameter. Besides several variants of anonymous methods, similarly to tasks, you can also use iterator events that can be used in combination with object parameters.

The `Integer` parameter passed to iterators represents the current iteration index.

```
TProc<Integer>
TProc<Integer, TLoopState>

TProc<Int64>
TProc<Int64, TLoopState>

TIteratorEvent = procedure (Sender: TObject; AIndex: Integer) of object;
TIteratorStateEvent = procedure (Sender: TObject; AIndex: Integer;
  const LoopState: TLoopState) of object;

TIteratorEvent64 = procedure (Sender: TObject; AIndex: Int64) of object;
TIteratorStateEvent64 = procedure (Sender: TObject; AIndex: Int64;
  const LoopState: TLoopState) of object;
```

The following example has an extremely simple computation that would not justify using *parallel for*, but because it is simple, it is easier to focus on the basic `TParallel.&For` code setup and functionality. To prevent blocking the UI, you will need to run such code in a background thread or another task:

```
procedure TMainForm.ForButtonClick(Sender: TObject);
var
  Res: TParallel.TLoopResult;
  Arr: array of integer;
  x: integer;
begin
  SetLength(Arr, 10);
  Res := TParallel.&For(1, 10,
    procedure(TaskIndex: integer)
    var
      f, i: integer;
    begin
      f := 1;
      for i := 1 to TaskIndex do
        f := f * i;
      Arr[TaskIndex-1] := f;
    end);
  if Res.Completed then
    begin
      for x in Arr do
        Memo1.Lines.Add(x.ToString);
    end;
end;
```

Before using *parallel for*, you should make sure that your problem is suitable for parallelization, meaning it is easy to partition the algorithm into separate subtasks that can run independently. For instance, if your computation heavily depends on the results of previous computations, then such an algorithm will not benefit from being broken down in pieces.

If the algorithm itself is suitable, make sure that you minimize the usage of shared data that will require locking mechanisms. The above example uses a shared array, but it does not require any locking, because the array is fully initialized before the algorithm runs, and each calculation operates on a totally separate chunk of the array.

If you cannot completely avoid locks, make them minimal. There is no point in running parallel threads if they will just keep interrupting each other. Such *parallel for* code can eventually take more time than using a single background thread to do the job.

Partitioning the data into appropriately-sized chunks for each iteration is also extremely important for performance. What works well for one algorithm may not work well for another. Starting every iteration adds some overhead; if the workload for an iteration is too small, your *parallel for* will spend more time starting each iteration than actually executing the code within the iteration.

22.8 Task continuation

Running multiple tasks and running parallel algorithms with the support of the PPL is rather easy. But if you need to wait for one task to complete, in order to start another one, your code will get way more complicated.

There is an additional feature in the PPL called a *future*, which simplifies using tasks that need to return some value. Tasks that don't need to return a value are basically futures that return nothing. If you need to use tasks in a series, starting one after another completes, it is simpler to treat them as futures regardless of whether you actually need some value or not.

Of course, if your tasks, the code you need to run one after another, can be executed in a synchronous manner, maybe you don't need task continuation at all, and you could just put all code in one single task.

Chapter 23

Back to the Future

Our ability to understand some piece of code is inherently linked to how easy or hard is to follow its flow. While the code and logical flow for some tasks in the background will not be too different, nor too much harder to follow than regular event-based code, some code might end up in *callback hell*, or being torn up in little pieces scattered all around. Understanding such code may be hard, and in such code, bugs can have a party for a long time without being detected.

If the code you need to run in the background has a direct flow on its own, then it is easy just to move the whole block to a background task and call some completion handler when you are done.

A single block does not mean that the complete logic has to be dumped in one single method, but rather that we can easily follow the flow going through the layers of code. In other words, each layer and each piece needs to be well-defined and understandable. That also means that each piece needs to be as self-contained as possible, and that understanding the code logic at each layer is not impeded by code intertwined with other layers.

For instance, the handling logic in button event handlers, even if there are many of those, is usually not hard to follow, because each event handler will act as a separate entity. You may have some logic spread around, where some event handlers' logic (like `Cancel` button) will require coordination with others, but in well-written code, those don't represent a huge problem.

However, the more you use it, the more often you will encounter situations where asynchronous programming will rip the program flow apart and scatter logically connected pieces all over the place. If you have ever read one of those *multiple ending* books, where at the end of every few pages you had to make a decision and then jump accordingly to some displaced page number to continue the story, you certainly know how hard it can be to imagine the complete flow and all the possibilities within the book.

You will be able to go one step forward every time and follow a particular story path, but you will have trouble going back, and even more in choosing and understanding different paths. While reading the book one story at a time is something you might consider fun, and you don't have to explore every possible story path, as a developer, you don't have that luxury. As a developer, you need to know what is going on in your application: You must understand the complete program flow and all possibilities.

As previously mentioned, often enough, asynchronous programming with callbacks may tear the flow apart in the middle of some operation, and you will either end up reading callback pyramids of doom, or jumping around between seemingly disconnected blocks of code. Debugging asynchronous code is also much harder, because jumping to the background usually requires a lot of setup code, so you cannot just simply step into some asynchronous code call and hope that you will continue the debugging session in just the right place.

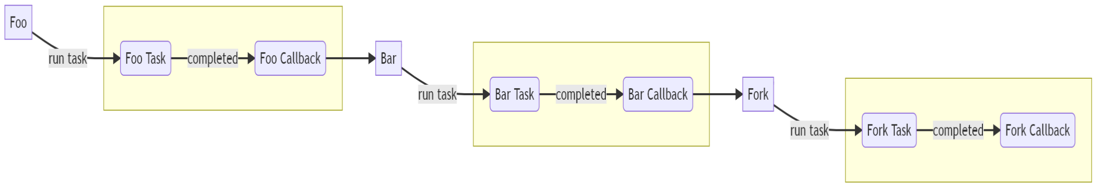
Following asynchronous code is not only a problem because all logically connected pieces of code may not be in the same place, but it also includes a temporal dimension. Understanding some code flow will require you to understand **when** some piece of code will be executed in relation to other pieces. If following *when something happens* is hard, then understanding the overall logic of a particular chunk of code will also be hard.

Even more problems, than just following some hard-to-understand code, can be caused by simply misunderstanding *what* and *when* some code does. When time is an important part, but also a very elusive part, of the equation, you may simply not take it into account by accident.

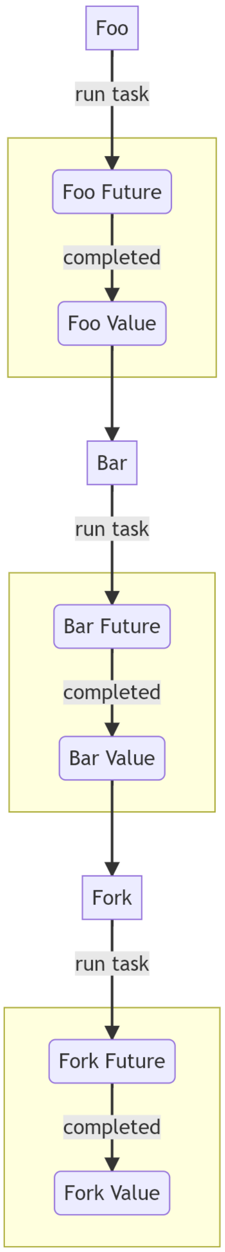
One of the solutions to such problems is *streamlining* asynchronous code flow and merging it back into a more direct one. This feature is called a *future* (and is often accompanied with *promises*, sometimes also called *completable futures*).

Using futures does not mean that the code is not executed asynchronously, only that the code using them has a seemingly linear, direct flow you can easily follow, understanding each layer in one go.

From callback hell:



To a more direct flow:



23.1 What is a *future*

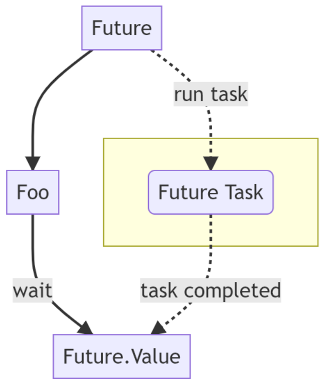
A future is a placeholder for a value that will be provided at a later time, but accessing that value where you need to use it will create a blocking call and wait until the value is actually there, ready to use. Slightly different, non-blocking implementations use a callback mechanism.

While a *future* represents a value placeholder, a *promise* represents the function that will actually set that value. The term *completable future* is shorthand for a *future* that also knows how to complete itself—in other words, it also contains a *promise*.

The Delphi PPL library's *future* implementation is actually a *completable future*, because it always comes with an attached *promise* function that will be used to set its value. It is an extension of the `ITask` interface and `TTask`, and provides factory functions directly on the `TTask` class. Basically, in Delphi PPL terms, a future is a task that returns some value.

Usage is rather simple:

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  Future: IFuture<string>;
begin
  Future := TTask.Future<string>(
    function: string
    begin
      Sleep(10000);
      Result := 'FUTURE';
    end);
  Foo;
  Memo1.Lines.Add(Future.Value);
end;
```



A future is a task, but it also returns a value, so the anonymous method commonly used to define task code is now a generic function that will set our future value. As previously mentioned, *futures* can be implemented using blocking calls or callbacks. The Delphi PPL uses the former approach, and calling `Future.Value` will actually perform a wait on the background task. Since it is a blocking call, code using *futures* can be written like any other code that uses a synchronous approach.

But, because accessing the *future*'s value is blocking, you should not use *futures* from the context of the main thread, or you will make the UI unresponsive. If you run the previous example, your UI will sleep for 10 seconds before you can continue.

Wrapping it in another thread or task is the way to go. Don't forget that interacting with the UI requires synchronization with the main thread, too.

Piece of cake... we have done it plenty of times already...

Now what???? Why is the UI blocked, when we wrapped everything in a background thread?

Incorrect code

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      Future: IFuture<string>;
    begin
      Future := TTask.Future<string>(
        function: string
        begin
          Sleep(10000);
          Result := 'FUTURE';
        end);
      TThread.Synchronize(nil,
        procedure
        begin
          Memo1.Lines.Add(Future.Value);
        end);
    end).Start;
end;
```

This is an easy mistake to make. We forgot that `Future.Value` is actually a blocking call, and placing it directly inside code that runs on the main thread, will do exactly as implied—block the main thread.

To work around that, you need to retrieve the future value in a background thread, store it in another variable, and then you can freely access that variable from the context of the UI thread:

Correct code

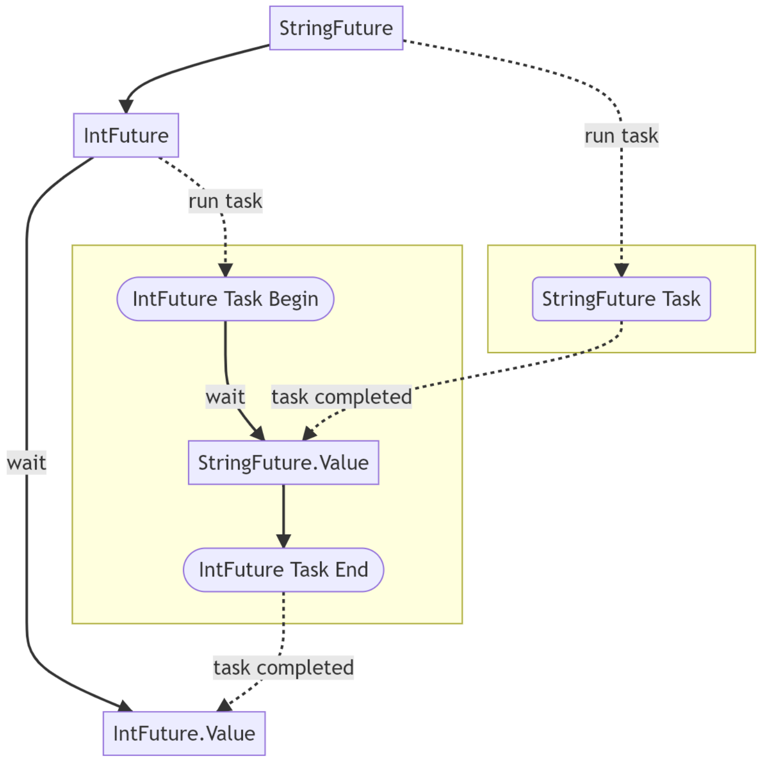
```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      Future: IFuture<string>;
      Value: string;
    begin
      Future := TTask.Future<string>(
        function: string
        begin
          Sleep(10000);
          Result := 'FUTURE';
        end);

      Value := Future.Value;

      TThread.Synchronize(nil,
        procedure
        begin
          Memo1.Lines.Add(Value);
        end);
    end).Start;
end;
```

On the bright side, if you need more than one future, your code does not require wrappers around wrappers for every future:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      Future: IFuture<string>;
      FutureInt: IFuture<Integer>;
      Value: Integer;
    begin
      OutputDebugString('THREAD BEGIN');
      Future := TTask.Future<string>(
        function: string
        begin
```

```
          OutputDebugString('STRING FUTURE BEGIN');
          Sleep(10000);
          Result := 'FUTURE';
          OutputDebugString('STRING FUTURE END');
        end);

      FutureInt := TTask.Future<Integer>(
        function: Integer
        begin
          OutputDebugString('INTEGER FUTURE BEGIN');
          Sleep(3000);
          Result := Length(Future.Value);
          OutputDebugString('INTEGER FUTURE END');
        end);

      OutputDebugString('FUTURE WAIT');
      Value := FutureInt.Value;

      TThread.Synchronize(nil,
        procedure
        begin
          Memo1.Lines.Add(Value.ToString);
        end);
      OutputDebugString('THREAD END');
    end).Start;
end;
```

There are some important considerations when using multiple *futures*. First of all, unlike the single-future example code, where the flow is easy to follow by reading code or observing its diagram, the above multi-future code flow is more convoluted:



As we will see in the following output (also visible in diagram), produced by the above example, all *futures* will start running immediately. It takes some time for future tasks to start running, so `FUTURE WAIT` may end up being called before the first future's code starts, but that is not something you can count on. If the first called future runs quickly, it may also end before the next future task begins. But there is also no guarantee that the first future will actually run before the second one. You cannot depend on anything there running in some defined order, except that each `BEGIN` block will start before its accompanying `END` block:

```
THREAD BEGIN
FUTURE WAIT
STRING FUTURE BEGIN
INTEGER FUTURE BEGIN
STRING FUTURE END
INTEGER FUTURE END
THREAD END
```

This is also a possible output of the above example:

```
THREAD BEGIN
FUTURE WAIT
INTEGER FUTURE BEGIN
STRING FUTURE BEGIN
STRING FUTURE END
INTEGER FUTURE END
THREAD END
```

The main point is that while `Future.Value` is a blocking call, it will only block the thread in which it is currently running. Each future (task) can run on its own thread if there are enough hardware resources available. So in an ideal situation, you will have four threads active when running the above example: the UI thread, the anonymous thread, the string future's thread, and the integer future's thread.

The integer future's thread must wait on the string future to complete in order to do its job. The real question is, when is the string value needed? If it is needed at the end of the integer future's task and there is some other longish job that the integer future can work on until it needs to wait for the string future, then the above setup will be beneficial for performance, because both the string and integer future tasks can do some work in parallel.

On the other hand, if the first thing the integer future needs is to retrieve the string future's value, before continuing with its work, then the above example is really the wrong way to do it, as it would cause one thread (the integer thread) to sit and wait doing nothing.

There are several ways to solve this issue.

The first, and probably best option, is to use a single future to perform the complete task. If there is no work that can be done in parallel, then there is very little point in running multiple futures:

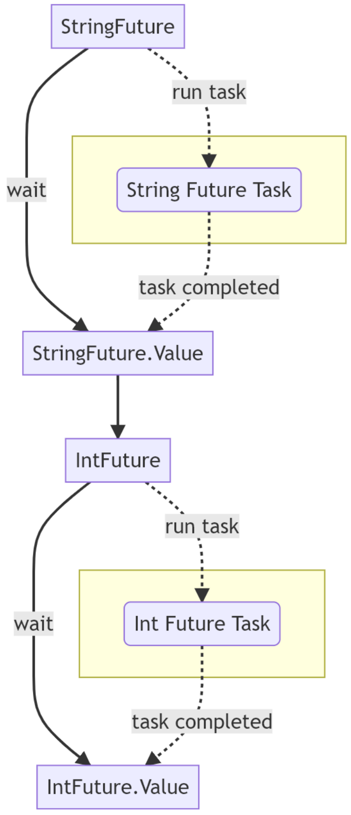
```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      FutureInt: IFuture<Integer>;
      Value: Integer;
    begin
      FutureInt := TTask.Future<Integer>(
        function: Integer
        var
          StringValue: string;
        begin
          // do work with string
          Sleep(10000);
          StringValue := 'FUTURE';
          // do work with integer
          Sleep(3000);
          Result := Length(StringValue);
        end);

      Value := FutureInt.Value;

      TThread.Synchronize(nil,
        procedure
        begin
          Memo1.Lines.Add(Value.ToString);
        end);
    end).Start;
end;
```

Another way to write it, would be explicitly taking the string future's value in the context of the anonymous thread. That way, the anonymous thread will never be able run the integer future's task before it actually retrieves the value of the string future. This is not the best possible code one could write, but it is a working option. Certainly better than the original example, where the integer task is twiddling its thumbs waiting for the string task to finish:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      Future: IFuture<string>;
      FutureInt: IFuture<Integer>;
      StringValue: string;
      Value: Integer;
    begin
      OutputDebugString('THREAD BEGIN');
      Future := TTask.Future<string>(
        function: string
        begin
          OutputDebugString('STRING FUTURE BEGIN');
          Sleep(10000);
          Result := 'FUTURE';
          OutputDebugString('STRING FUTURE END');
        end);

      OutputDebugString('STRING FUTURE WAIT');
      StringValue := Future.Value;
      OutputDebugString('STRING VALUE RETRIEVED');

      FutureInt := TTask.Future<Integer>(
        function: Integer
        begin
          OutputDebugString('INTEGER FUTURE BEGIN');
          Sleep(3000);
          Result := Length(StringValue);
          OutputDebugString('INTEGER FUTURE END');
        end);

      OutputDebugString('INTEGER FUTURE WAIT');
      Value := FutureInt.Value;

      TThread.Synchronize(nil,
        procedure
        begin
          Memo1.Lines.Add(Value.ToString);
        end);
      OutputDebugString('THREAD END');
    end).Start;
end;
```

```
THREAD BEGIN
STRING FUTURE WAIT
STRING FUTURE BEGIN
STRING FUTURE END
STRING VALUE RETRIEVED
INTEGER FUTURE WAIT
INTEGER FUTURE BEGIN
INTEGER FUTURE END
THREAD END
```

Running separate futures, one after another, also has simpler code flow compared to starting multiple future tasks in parallel. Again, a more complex flow in parallel running futures is not a problem on its own, if the second task itself has some independent work to do before it needs the result of the first future.


StringFuture
run task
String Future Task
wait
task completed
StringFuture.Value
IntFuture
run task
Int Future Task
wait
task completed
IntFuture.Value

Chapter 24

There is no Future like your own Future

The Delphi PPL has a few downsides. Firstly, it was added in XE7. Delphi versions tend to live quite long in production, so if, for any reason, you need to stick to an older one, there is no PPL to begin with.

But, my actual *future* story didn't even start with Delphi. I was trying to solve callback hell in Java and Swift for mobile applications (the grass is not greener on the other side). There are many different open-source libraries available, not to mention ReactiveX, which not only gives you *future*-like functionality, but much, much, much more. But, except for RX, which offers a fairly similar API on both platforms, all other solutions are pretty much incompatible with each other.

Of course, you cannot use Java code in Swift or vice versa, but having consistent naming and behavior helps when transferring the same piece of code from one language to another.

My original intent was to get out of callback hell, but I was not really into replacing it with dependency hell, especially not with something as heavy as the whole RX framework.

As a concept, *futures* are not so complicated, and instead of taking time to come to grips with some existing frameworks, I thought making my own implementation to fulfill my rather simple needs would be time better spent.

What does all that have to do with Delphi, you might ask? For starters, callback hell and dependency hell are universal. Quite often, solutions and approaches from one language can fit into another language, too. If I make my own solution in Java and Swift, I could also apply it to Delphi and make my code even more portable, and usable in the older Delphi versions that I still need to support.

What I had in Swift was the following code. You don't need to understand Swift code to see that the `downloadAndProcessData` function is hard to follow. Error handling is repeated in several places, every successful operation is indented a bit more... it's a total nightmare:

```
func downloadData(_ path: String, completionBlock: (result: Data) -> Void) {
   ...
}

func processData(_ data: Data, completionBlock: (result: Data) -> Void) {
  ...
}

func storeData(_ data: Data, completionBlock: (result: String) -> Void) {
  ...
}

func downloadAndProcessData(url: String,
  completionBlock: (result: String?, error: Error?) -> Void) {
    downloadData(url) { data, error in
        guard let data = data else {
            completionBlock(nil, error)
            return
        }
        processData(data) { processedData, error in
            guard let processedData = processedData else {
                completionBlock(nil, error)
                return
            }
            storeData(processedData) { resultData in
                guard let resultData = resultData else {
                    completionBlock(nil, error)
                    return
                }
                completionBlock(resultData)
            }
        }
    }
}
```

And what I wanted to have instead would be the following:

```
  downloadData(url)
  .then { data in
    processData(data)
  }
  .then { processedData in
    storeData(processedData)
  }
```

```
    .onCompleted {
      // do something always
    }
    .onSuccess { resultData in
      print("SUCCESS \(resultData)")
    }
    .onFailure { error in
      print("FAILURE \(error)")
    }
```

Delphi is a bit more verbose than Swift, but generally, I was aiming for achieving the following result:

```
    DownloadData(Url)
    .Then_<TStream>(
      function(const Value: TStream): INxFuture<string>
      begin
        Result := ProcessData(Value);
      end)
    .Then_<string>(
      function(const Value: string): INxFuture<string>
      begin
        Result := StoreData(Value);
      end)
    .Sync
    .OnCompleted(
      procedure()
      begin
        Memo1.Lines.Add('COMPLETED');
      end)
    .OnSuccess(
      procedure(const Value: string)
      begin
        Memo1.Lines.Add('SUCCESS ' + Value);
      end)
    .OnFailure(
      procedure(const Value: string)
      begin
        Memo1.Lines.Add('FAILURE ' + Value);
      end);
```

Why a *fluent interface* and not task+await like the Delphi PPL's *future* approach?

Like I said, I started by solving issues in my Java and Swift code, and *fluent interfaces* are quite commonly used there. Another reason is that many commonly-used frameworks in Java and Swift use a callback mechanism for asynchronous task completion notifications. And once you start calling asynchronous code with callbacks, it requires a lot of ceremony to put that code back into a synchronous flow. And then when you do get all your subtasks into order, you still need to have an asynchronous call at the top level (with a callback), otherwise you will block the GUI thread, which is precisely what you are trying to avoid in the first place.

In other words, fitting synchronous code into a callback-based future is easy. Fitting asynchronous code into a waitable future, not so much. So a callback-based future is more versatile. When you make such a future flow more direct by using a *fluent interface*, you get a fairly flexible framework that can easily adapt to different requirements, and using it produces more maintainable code.

Another point of this whole `NxFuture` exercise is moving one step closer towards creating a Delphi ReactiveX framework that offers way more functionality than the Delphi PPL library has. And polishing designs and solving real-life code challenges is also easier when starting with simpler things, that are still complicated enough to prevent falling into the *too trivial* example trap, where you think some solution is viable and then it starts falling apart when you add some complexity. Abandoning wrong designs is an integral part of the design process, and it is easier when you have to abandon less code rather than more.

24.1 Generic approach to future

If you want to have the *same* type running different code, then you use inheritance. If you want different types running the same code, then you use generics. This is why, quite often, various data-oriented frameworks lean on generics: They want to handle different kinds of data in the same way.

As we have already established, in some cases generics can cause trouble and prevent us from achieving a particular code flow. This is where `TValue` comes in to save the day.

But before calling for help, let's try anyway, and establish what the real issues are when implementing a generic-based future. The following code is rather simplified, to avoid unnecessary distractions from the main code logic and difficulties.

A future represents a value returned after performing some task. In the PPL, a future is defined as an interface, with a `Value` property that contains the result:

```
IFuture<T> = interface(ITask)
  ...
  function GetValue: T;
  property Value: T read GetValue;
end;
```

A callback-based future has a slightly different design. There is no function that returns a result, but the result will be passed to the callback on completion. In order to examine whether the future was successfully retrieved or not, the future's result will not only contain the value, but also the status of the operation.

This is not so different from a task-based future, because tasks also give us status and exception handling capabilities, just more tightly coupled. From the perspective of separating responsibilities, this is a plus—because the result is a rather independent entity and can be used even outside the context of the task/future.

To make our future result type really simple, there will be no advanced exception handling with capturing and forwarding exceptions, and the error will be represented with a string. If the string is empty, there was no exception. If it is not empty, it will contain some error code, message or whatever value we want. Remember, this is prototype code—it will work as intended, but it is stripped of anything that will take us away from the basic goal: testing the viability of prototype code. More sophisticated exception handling can be always added later on.

We need to be able to tell whether the future's value has been set, so `Value` is declared as the `Nullable` type. Any nullable will do, but in this case we could also have used a simple Boolean flag to tell us whether `Value` has been set or not, because as a field of the future class, it will always be properly initialized.

Also, this implementation will only support the completion handler `OnCompleted`, not the `OnSuccess` and `OnFailure` handlers that are usually provided for convenience.

The framework utilizes two classes and their accompanying interfaces: `TNxFuture<V>` and `TNxPromise<V>`. We could combine them both into a single one, but this way their usage is more clear, as there is a clear separation between the *fluent interface* style that strictly operates on futures, and various callbacks that need a promise.

As previously mentioned, promises are also called completable futures—and this is exactly what their purpose is in the framework. Promises are used in places where the future value is determined (calculated) and we need to set that actual value in our future value placeholder. Because calculating the value may fail, the promise has two methods: `Resolve` for setting the actual value, and `Reject` to set the error if there was an error while calculating the value.

When the promise is resolved or rejected, this triggers a call to the `OnCompleted` completion handler and we can inspect the passed result to determine whether the operation was successful and we can use the resulting value, or whether an error occurred.

```
type
  TNxFutureResult<V> = record
  public
    Value: Nullable<V>;
    Error: string;
    function IsNull: boolean;
  end;
```

```
    INxFuture<V> = interface;
    INxPromise<V> = interface;

    TNxCompletedCallback<V> = reference to
      procedure(const aResult: TNxFutureResult<V>);
    TNxThenCallback<V, T> = reference to
      procedure(const aPromise: INxPromise<T>; const aValue: V);
    TNxAsyncCallback<V> = reference to procedure(const aPromise: INxPromise<V>);

    INxFuture<V> = interface
      function Sync: INxFuture<V>;
      function OnCompleted(const aCallback: TNxCompletedCallback<V>): INxFuture<V>;
    end;

    INxPromise<V> = interface(INxFuture<V>)
      procedure Resolve(const aValue: V);
      procedure Reject(const aError: string);
    end;

    TNxFuture<V> = class(TInterfacedObject, INxFuture<V>)
    strict protected
      fResult: TNxFutureResult<V>;
      fSync: Boolean;
      fCompletedCallback: TNxCompletedCallback<V>;
      procedure SetResult(const aResult: TNxFutureResult<V>);
      procedure DoCompleted;
    public
      function Sync: INxFuture<V>;
      function OnCompleted(const aCallback: TNxCompletedCallback<V>): INxFuture<V>;
      function Then_<T>(const aCallback: TNxThenCallback<V, T>): INxFuture<T>;
      class function Async(const aCallback: TNxAsyncCallback<V>): INxFuture<V>;
        static;
    end;

    TNxPromise<V> = class(TNxFuture<V>, INxPromise<V>)
    public
      procedure Resolve(const aValue: V);
      procedure Reject(const aError: string);
    end;

  implementation

  { TNxFutureResult<V> }
```

```
function TNxFutureResult<V>.IsNull: boolean;
begin
  Result := Value.IsNull and Error.IsEmpty;
end;

{ TNxFuture<V> }

procedure TNxFuture<V>.SetResult(const aResult: TNxFutureResult<V>);
begin
  fResult := aResult;
  if fSync then
    TThread.Synchronize(nil,
      procedure
      begin
        DoCompleted;
      end)
  else
    DoCompleted;
end;

function TNxFuture<V>.Sync: INxFuture<V>;
begin
  Result := Self;
  fSync := True;
end;

function TNxFuture<V>.Then_<T>(const aCallback: TNxThenPromiseCallback<V, T>):
  INxFuture<T>;
var
  Promise: INxPromise<T>;
begin
  Promise := TNxPromise<T>.Create;
  Result := Promise;
  OnCompleted(
    procedure(const aResult: TNxFutureResult<V>)
    begin
      if aResult.Error.IsEmpty then
        try
          aCallback(Promise, aResult.Value);
        except
          on E: Exception do
            Promise.Reject(E.Message);
        end
      else
```

```
        Promise.Reject(aResult.Error);
    end);
end;

procedure TNxFuture<V>.DoCompleted;
begin
  if Assigned(fCompletedCallback) then
    fCompletedCallback(fResult);
  fCompletedCallback := nil;
end;

function TNxFuture<V>.OnCompleted(const aCallback: TNxCompletedCallback<V>):
  INxFuture<V>;
begin
  Result := Self;
  // if the result is not yet populated, store the callback for later
  // if we already have the result, we can call the callback immediately
  if fResult.IsNull then
    fCompletedCallback := aCallback
  else
    aCallback(fResult);
end;

class function TNxFuture<V>.Async(const aCallback: TNxAsyncPromiseCallback<V>):
  INxFuture<V>;
var
  Promise: INxPromise<V>;
begin
  Promise := TNxPromise<V>.Create;
  Result := Promise;
  TThread.CreateAnonymousThread(
    procedure
    begin
      try
        aCallback(Promise);
      except
        on E: Exception do
          Promise.Reject(E.Message);
      end;
    end).Start;
end;

{ TNxPromise<V> }
```

```
procedure TNxPromise<V>.Resolve(const aValue: V);
var
  Result: TNxFutureResult<V>;
begin
  Result.Value := aValue;
  SetResult(Result);
end;

procedure TNxPromise<V>.Reject(const aError: string);
var
  Result: TNxFutureResult<V>;
begin
  Result.Error := aError;
  SetResult(Result);
end;
```

And here is a simple example using the above future framework, that we can analyze step by step:

```
begin
  Memo1.Lines.Add('BEFORE');
  TNxFuture<string>.
    Async(
      procedure(const aPromise: INxPromise<string>)
      begin
        Sleep(1000); // simulate work
        aPromise.Resolve('First');
      end)
    .Sync
    .OnCompleted(
      procedure(const Value: TNxFutureResult<string>)
      begin
        Memo1.Lines.Add('COMPLETED');
        if Value.Error.IsEmpty then
          Memo1.Lines.Add('SUCCESS ' + Value.Value)
        else
          Memo1.Lines.Add('FAILURE ' + Value.Error);
      end);
  Memo1.Lines.Add('AFTER');
end;
```

Running the example will produce the following output:

```
BEFORE
AFTER
COMPLETED
SUCCESS First
```

Replacing `aPromise.Resolve` with `aPromise.Reject('Some Error')` yields slightly different results:

```
BEFORE
AFTER
COMPLETED
FAILURE Some Error
```

The first call is to the class function `Async` that returns the initial future, actually a promise that is also passed to a callback, and it is used to resolve the promise with a value or reject it with an error. The callback is where the actual work is being done. Since it is a class function that's returning the interface, we don't need explicit variable declaration, because the implicit one will keep our chain of future objects alive until they go out of scope. However, we could also have assigned the chain to an explicit reference:

```
var
  Future: INxFuture<string>;
begin
  Future := TNxFuture<string>.
    Async(
  ...
```

`Async` runs the callback in an anonymous thread, but a PPL task can be used instead, or any other 3rd-party or custom way of running that callback in a thread. If you are interacting with libraries that already perform tasks in threads and use completion handlers, then running an additional thread is a bit overkill, but you can easily implement a `Run` variant that simply skips creating an anonymous thread. In the completion handler, depending on the result of such an operation, you just need to resolve the promise with an actual value or reject it with an error.

The `Sync` function that is called next in the chain, simply indicates that any further calls should be synchronized with the main thread. This is why it is safe to use GUI controls, like the memo in the `OnCompleted` completion handler. If our completion handler doesn't need to run in the context of the main thread, we can just omit the call to `Sync`.

So far, the generic future concept seems to work as expected. However, problems come further down the line, when you want to chain futures. Since interfaces don't support parameterized methods, in order to call the `Then_` method, we need to break the future chain and use typecasting. Such code can also cause internal compiler errors in older Delphi versions.

Just like in our PPL future examples, we don't always need to use separate futures for every step in our algorithm. We can just run everything sequentially in a single future, but just like in the PPL example, this is only possible if each step doesn't run any asynchronous code on its own, and doesn't rely on additional completion handlers.

The `Then_` method's purpose is similar to the `Async` method—it also creates a new promise and passes it to the provided callback, but `Then_` is also a continuation method, and the result of the previous future's calculations will also be passed on to the callback. If the previous future failed, the `Then_` callback will never run.

When we get to the `Then_` method we are already running code in the context of the initial background thread created in `Async`, so there is no need to start a new one:

```
var
  Future: INxFuture<string>;
begin
  Memo1.Lines.Add('BEFORE');
  Future := TNxFuture<string>
    .Async(
      procedure(const aPromise: INxPromise<string>)
      begin
        Sleep(1000); // simulate work
        aPromise.Resolve('First');
      end);

  TNxFuture<string>(Future)
    .Then_<integer>(
      procedure(const aPromise: INxPromise<integer>; const aValue: string)
      begin
        aPromise.Resolve(Length(aValue));
      end)
    .Sync
    .OnCompleted(
      procedure(const Value: TNxFutureResult<integer>)
      begin
        Memo1.Lines.Add('COMPLETED');
        if Value.Error.IsEmpty then
          Memo1.Lines.Add('SUCCESS ' + IntToStr(Value.Value))
        else
          Memo1.Lines.Add('FAILURE ' + Value.Error);
      end);
  Memo1.Lines.Add('AFTER');
end;
```

This is the point where it becomes obvious that the generic approach has too many downsides, and the alternative solution using `TValue` might make more sense.

24.2 TValue approach to future

The generic approach to future frameworks, was a trimmed-down, extremely simplistic example to show why generics cause problems. On the other hand, the TValue approach works nicely, and it makes sense to expand it with additional functionality that makes its usage more convenient, such as separate OnSuccess and OnFailure callbacks.

Since a TValue-based future does not need parameterized methods in the interface declaration, we can also declare the Then_ method as part of the interface. Besides that, the rest of the implementation logic is basically the same as in the generic variant:

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

type
  TNxFutureResult = record
  public
    Value: TValue;
    Error: string;
    function IsNull: boolean;
  end;

  INxFuture = interface;
  INxPromise = interface;

  TNxCompletedCallback = reference to procedure(const aResult: TNxFutureResult);
  TNxSuccessCallback = reference to procedure(const aValue: TValue);
  TNxFailureCallback = reference to procedure(const aError: string);
  TNxAsyncCallback = reference to procedure(const aPromise: INxPromise);
  TNxThenCallback = reference to
    procedure(const aPromise: INxPromise; const aValue: TValue);

  INxFuture = interface
    [Result:unsafe]
    function Sync: INxFuture;
    [Result:unsafe]
    function OnCompleted(const aCallback: TNxCompletedCallback): INxFuture;
    [Result:unsafe]
    function OnSuccess(const aCallback: TNxSuccessCallback): INxFuture;
    [Result:unsafe]
    function OnFailure(const aCallback: TNxFailureCallback): INxFuture;
    function Then_(const aCallback: TNxThenCallback): INxFuture;
  end;
  INxPromise = interface(INxFuture)
    procedure Resolve(const aValue: TValue);
    procedure Reject(const aError: string);
  end;

  TNxFuture = class(TInterfacedObject, INxFuture)
  strict protected
    fResult: TNxFutureResult;
    fSync: Boolean;
    fCompletedCallback: TNxCompletedCallback;
    fSuccessCallback: TNxSuccessCallback;
    fFailureCallback: TNxFailureCallback;

    procedure SetResult(const aResult: TNxFutureResult);
    procedure DoCompleted;
    procedure DoSuccess;
    procedure DoFailure;
  public
    [Result:unsafe]
    function Sync: INxFuture;
    [Result:unsafe]
    function OnCompleted(const aCallback: TNxCompletedCallback): INxFuture;
    [Result:unsafe]
    function OnSuccess(const aCallback: TNxSuccessCallback): INxFuture;
    [Result:unsafe]
    function OnFailure(const aCallback: TNxFailureCallback): INxFuture;
    function Then_(const aCallback: TNxThenCallback): INxFuture;
    class function Async(const aCallback: TNxAsyncCallback): INxFuture; static;
  end;

  TNxPromise = class(TNxFuture, INxPromise)
  public
    procedure Resolve(const aValue: TValue);
    procedure Reject(const aError: string);
  end;

implementation

{ TNxFutureResult }

function TNxFutureResult.IsNull: boolean;
begin
  Result := Value.IsEmpty and Error.IsEmpty;
end;
```

```
{ TNxFuture }

procedure TNxFuture.SetResult(const aResult: TNxFutureResult);
begin
  fResult := aResult;
  if fSync then
    begin
      TThread.Synchronize(nil,
        procedure
        begin
          if not fResult.Value.IsEmpty then
            DoSuccess
          else
            DoFailure;
          DoCompleted;
        end);
    end
  else
    begin
      if not fResult.Value.IsEmpty then
        DoSuccess
      else
        DoFailure;
      DoCompleted;
    end;
end;

function TNxFuture.Sync: INxFuture;
begin
  Result := Self;
  fSync := True;
end;

function TNxFuture.Then_(const aCallback: TNxThenCallback): INxFuture;
var
  Promise: INxPromise;
begin
  Promise := TNxPromise.Create;
  Result := Promise;
  OnSuccess(
    procedure(const aValue: TValue)
    begin
      try
        aCallback(Promise, aValue);
      except
        on E: Exception do
          Promise.Reject(E.Message);
      end;
    end);
  OnFailure(
    procedure(const aError: string)
    begin
      Promise.Reject(aError);
    end);
end;

procedure TNxFuture.DoCompleted;
begin
  if Assigned(fCompletedCallback) then
    fCompletedCallback(fResult);
  fCompletedCallback := nil;
end;

procedure TNxFuture.DoSuccess;
begin
  if Assigned(fSuccessCallback) then
    fSuccessCallback(fResult.Value);
  fSuccessCallback := nil;
end;

procedure TNxFuture.DoFailure;
begin
  if Assigned(fFailureCallback) then
    fFailureCallback(fResult.Error);
  fFailureCallback := nil;
end;

function TNxFuture.OnCompleted(const aCallback: TNxCompletedCallback): INxFuture;
begin
  Result := Self;
  if fResult.IsNull then
    fCompletedCallback := aCallback
  else
    aCallback(fResult);
end;

function TNxFuture.OnSuccess(const aCallback: TNxSuccessCallback): INxFuture;
begin
  Result := Self;
  if fResult.IsNull then
    fSuccessCallback := aCallback
  else
  if not fResult.Value.IsEmpty then
    aCallback(fResult.Value);
end;

function TNxFuture.OnFailure(const aCallback: TNxFailureCallback): INxFuture;
begin
  Result := Self;
  if fResult.IsNull then
    fFailureCallback := aCallback
  else
  if not fResult.Error.IsEmpty then
    aCallback(fResult.Error);
end;

class function TNxFuture.Async(const aCallback: TNxAsyncCallback): INxFuture;
var
  Promise: INxPromise;
begin
  Promise := TNxPromise.Create;
  Result := Promise;
  TThread.CreateAnonymousThread(
    procedure
    begin
      try
        aCallback(Promise);
      except
        on E: Exception do
          Promise.Reject(E.Message);
      end;
    end).Start;
end;

{ TNxPromise }

procedure TNxPromise.Resolve(const aValue: TValue);
var
  Result: TNxFutureResult;
begin
  Result.Value := aValue;
  SetResult(Result);
end;
```

```
procedure TNxPromise.Reject(const aError: string);
var
  Result: TNxFutureResult;
begin
  Result.Error := aError;
  SetResult(Result);
end;
```

And this is how a TValue-based future example looks. Comparing to generic futures, the visual flow is not interrupted because there was no need for typecasting:

```
begin
  Memo1.Lines.Add('BEFORE');
  TNxFuture
    .Async(
      procedure(const aPromise: INxPromise)
      begin
        Sleep(1000);
        aPromise.Resolve('First');
      end)
    .Then_(
      procedure(const aPromise: INxPromise; const aValue: TValue)
      begin
        aPromise.Resolve(Length(aValue.AsString));
      end)
    .Sync
    .OnCompleted(
      procedure(const Value: TNxFutureResult)
      begin
        Memo1.Lines.Add('COMPLETED');
        if Value.Error.IsEmpty then
          Memo1.Lines.Add('SUCCESS ' + IntToStr(Value.Value.AsInteger))
        else
          Memo1.Lines.Add('FAILURE ' + Value.Error);
      end);
  Memo1.Lines.Add('AFTER');
end;
```

Since the TValue future variant has separate OnSuccess and OnFailure handlers, they can be also used to simplify the success and failure logic. Using any of the completion handlers is purely optional:

```
    .Sync
    .OnSuccess(
      procedure(const Value: TValue)
      begin
        Memo1.Lines.Add('SUCCESS ' + IntToStr(Value.AsInteger))
      end)
    .OnFailure(
      procedure(const Error: string)
      begin
        Memo1.Lines.Add('FAILURE ' + Error);
      end)
```

In a little more than 200 lines of code, we have implemented a rather usable future framework. There is always room for improvement and adding more functionality, but the base code is rather simple. It is also flexible, because the future functionality is not tied to any particular threading or task framework, and any (even multiple ones simultaneously) of those can be used. You might like it, or you might not. It does not matter, though. The end goal is not the most important part here, it is the journey itself.

This is also a good example of how implementing various concepts does not need to be hard and complex. Most complexities arise from convenience features that are added on top of the original idea, and also from performance or memory optimizations.

When you land on fully-fledged framework code, it can be hard to find your way around it and even understanding which parts of the framework are crucial for particular functionality and which ones are not. If you are familiar with the concept the framework implements, understanding the code will be easier, but even that is not a guarantee. On the other hand, if you don't fully grasp the concepts, going through existing framework code is probably the least helpful thing you can do. Playing and writing your own code might give you a better understanding of the principles applied. And even if you abandon your exercise code and go back to using existing frameworks, at that point you will usually be in a better position to use it, or even improve it for your particular use cases.

And the last thing to keep in mind: Whatever kind of code you write, you will always need to make some compromises. For each particular part of code, you will have to think about the most important requirements and focus on those, because:

Fast code, thread-safe code, readable code. You can pick only one.

Part 5. Thread safety

Chapter 25

What is thread safety anyway?

Multithreading can be hard to do right. The most common point of failure is assuming some code is thread-safe when it actually is not. And then the whole multithreading castle crumbles into ruins.

So all you need to do is make your code thread-safe, and all will be good. And this is where the real trouble begins. In order to make something thread-safe, first you need to know what *thread-safe* actually means. It turns out that *thread-safe* is a pretty vague term.

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.

Not much better, is it?

What is proper behavior and unintended interaction? How to ensure both?

The main problem with the definition of *thread-safe code* is that it highly depends on the code itself and how it is used in a particular scenario. So, you literally need to re-evaluate thread safety for every piece of thread-safe code you write. After a while, this daunting task becomes easier as you start recognizing common patterns.

For instance, you can have a *thread-safe* list, and some code using that list will be thread-safe, while some will not.

The following code presents a simple, thread-safe list of integers. You can safely add an integer to the list, or check whether some value already exists in the list, from multiple threads. Access to the shared data, the `FItems` array, is protected by a lock—a critical section—and only one thread can execute the code within that lock at any given moment:

```
type
  TIntegerList = class
  protected
    FLock: TCriticalSection;
    FItems: array of Integer;
    ...
  public
    procedure Add(Value: Integer);
    function Exists(Value: Integer): Boolean;
    ...
  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;
```

But, that list does not have the built-in ability to add a unique integer value to the list. No problem, this is not so hard to do, and the list can be easily extended.

Thread-unsafe code

```
type
  TUniqueIntegerList = class(TIntegerList)
  ...
  public
    procedure AddUnique(Value: Integer);
  end;

procedure TUniqueIntegerList.AddUnique(Value: Integer);
begin
  if not Exists(Value) then
    Add(Value);
end;
```

Bang! It is broken. . .

While both `Add` and `Exists` are thread-safe operations on the list, executing them separately is not. It is perfectly possible that one or two separate threads will pass through the `not Exists` condition, and successfully add the same value to the list. This code will not cause memory corruption, nor will it cause crashes, but it will not always run correctly. We can get unintended interactions between threads. If the list is supposed to contain only unique integer values, the possibility that some values will not be unique can eventually cause the application to misbehave.

One of the possible sequences of execution on an empty list, or a list that does not contain a value of 10:

```
Thread 1 -> not Exists(10) -> true
Thread 2 -> not Exists(10) -> true
Thread 1 -> Add(10)
Thread 2 -> Add(10)
```

In order to make the `AddUnique` method thread-safe, we have to lock access to the shared `FItems` array for the whole method at once:

Thread-safe code

```
procedure TUniqueIntegerList.AddUnique(Value: Integer);
begin
  FLock.Enter;
  try
    if not Exists(Value) then
      Add(Value);
  finally
    FLock.Leave;
  end;
end;
```

In such code, we have to make sure the used lock is reentrant (because it will be called by the same thread more than once), or we have to expand the `Exists` and `Add` methods inside `AddUnique`. If code performance is critical, we might do that anyway to avoid unnecessary nested locking.

25.1 Wrong assumptions

As in the previous example, wrong assumptions are the main cause of issues in multithreaded code. You will either assume wrongly that some code is thread-safe, and it will break because of it. Or you will assume wrongly that some code is not thread-safe, and additional levels of protection can break it.

Knowing what kind of code is thread-safe, and under which conditions, is paramount.

When it comes to thread safety in Delphi (actually, this is not a Delphi-specific thing), there is very little built in by default. Basically, if you are writing some code that has to be thread-safe, you have to take care of the thread safety part all by yourself. And you have to be very careful with your assumptions, as you can easily come to the wrong conclusions.

To demonstrate how deeply unsafety goes, and how easy it is to make the wrong assumptions, I will use ARC as an example.

Of course, accessing the content of an object instance is not thread-safe in terms of having multiple threads reading and writing to that content. But what about references? If you don't have to change the content of the object instance across multiple threads, then surely you can safely handle its references and its lifetime across thread boundaries without additional safety mechanisms - without locks? After all, reference counting itself uses a locking mechanism to ensure a consistent reference count.

If you think ARC references are thread-safe, think again. They are not thread-safe at all. Not even close. Even something as trivial as the assignment of one reference to another can lead to a total disaster in a multithreaded scenario. To be fair, assignment of anything is not a thread-safe operation. For some basic simple types, the assignment will be atomic under some circumstances, but atomicity does not imply thread safety. Thread safety is context-dependent, and while occasionally atomic reads and writes may be enough for achieving thread saftety in the context of some code functionality, sometimes, in a very similar situation, atomicity will not be enough. You can find more about assignment thread safety in the *Assignment thread safety* chapter.

The only thread-safe part of the reference counting mechanism is keeping the reference count variable (number) in a consistent state. That variable, and that variable alone, is protected (locked) from being simultaneously accessed from multiple threads during reference count increments or decrements. And there is more code involved in assigning one reference to another than changing the reference count variable alone.

Assigning nil—clearing the reference—calls the `_IntfClear` helper function, while assigning one strong reference to another calls the `_IntfCopy` helper function:

```
function _IntfClear(var Dest: IInterface): Pointer;
var
  P: Pointer;
begin
  Result := @Dest;
  if Dest <> nil then
  begin
    P := Pointer(Dest);
    Pointer(Dest) := nil;
    IInterface(P)._Release;
  end;
end;

procedure _IntfCopy(var Dest: IInterface; const Source: IInterface);
var
  P: Pointer;
begin
  P := Pointer(Dest);
  if Source <> nil then
    Source._AddRef;
  Pointer(Dest) := Pointer(Source);
  if P <> nil then
    IInterface(P)._Release;
end;
```

If you have multiple threads accessing—reading and writing—the same reference, one thread can easily interfere with the other. Let's say we have a shared interface reference `Data`, pointing to a previously created object, one thread that sets that reference to nil, and another thread that tries to take another strong reference from that one with an assignment to another variable `Tmp`. The first thread will execute the `_IntfClear` function that will result in object destruction.

The second thread, trying to grab a strong reference preventing object destruction, will execute the `_IntfCopy` function:

Thread-unsafe code

```
var
  Data, Tmp: IInterface;

  // assuming construction was completed
  // before multiple threads started to write Data
  Data := TSomeData.Create;

  ...

  Data := nil; // Thread 1 -> _IntfClear

  Tmp := Data; // Thread 2 -> _IntfCopy
```

If the `Source._AddRef` line from the `_IntfCopy` function executes before the call to `IInterface(P)._Release` manages to decrease the reference count to zero and subsequently calls the object's destructor, all is good. The second thread will successfully capture another strong reference to the object instance. However, the first thread can interrupt the second thread at the wrong moment, just after the `Source <> nil` check was successfully passed, but before `Source._AddRef` had the chance to increment the object's reference count. In that case, the first thread will happily destroy the object instance while the second thread will happily grab a strong reference to an already-nuked object, and you can forget about *happily ever after*.

When it comes to weak interface references, they are not thread-safe, either. How could they be, when even taking a strong reference from a strong one is not thread-safe? Why assigning a weak reference to a strong reference is not thread-safe should be obvious from the previous example, because the same kind of code is involved. But because the *wrong assumptions* curse works in mysterious ways, it does not hurt to explicitly say that taking a strong reference from a weak one, in situations where original strong reference has been written to, is not thread-safe:

Thread-unsafe code

```
var
  Data, Tmp: IInterface;
  [weak] WData: IInterface;

  // assuming construction and strong to weak assignment was completed
  // before multiple threads started to write Data
  Data := TSomeData.Create;
  WData := Data;

  ...

  Data := nil; // Thread 1 -> _IntfClear

  Tmp := WData; // Thread 2 -> _IntfCopy
```

Note: Besides interface references to objects, the reference counting mechanism is also used for other types like strings and dynamic arrays. Assignment of those types is also not thread-safe. I know, I know... I already said that assignments in general are not thread-safe, but it never hurts to repeat that fact.

Rule of thumb. Is it thread-safe? Assume not.

25.2 Working example of thread-unsafe code

If you want to observe broken ARC in action, you can run the following code, and watch the invalid pointer operations dropping in. That code is equally broken on all Delphi compilers, classic or ARC (I am mentioning the deprecated ARC compiler here to show that even a full-blown ARC compiler is not thread-safe in such scenarios). Please note, when I say broken, I am not implying a bug in the compiler. It is merely an example of thread-unsafe code.

Thread-unsafe code

```
uses
  System.SysUtils,
  System.Classes;

var
  Data: IInterface;

procedure Test;
var
  Tmp: IInterface;
  i, j: Integer;
begin
  for i := 0 to 1000 do
    begin
      Data := TInterfacedObject.Create;

      TThread.CreateAnonymousThread(
          procedure
          var
            i: Integer;
          begin
            for i := 0 to 10 do Sleep(15);
            Data := nil;
          end).Start;

      for j := 0 to 1000000 do
        begin
          Tmp := Data;
          if not Assigned(Tmp) then break;
          Tmp := nil;
        end;
    end;
end;

begin
  try
    Test;
  except
    on E: Exception do Writeln(E.ClassName, ': ', E.Message);
  end;
  Writeln('Finished');
end.
```

Chapter 26

The importance of thread safety

Thread safety is an absolute category. Either some code is thread-safe or it is not. There is no middle ground. You cannot make it *almost* thread-safe and safely ignore the impact on the application's stability. What is worse, not only stability is at stake, but also the consistency of the user's data.

Issues caused by thread-unsafe code can manifest themselves in such a random manner that they can have an impact on literally any part of the application, causing crashes, deadlocks, data corruption… just about anything.

One of the problems is the *it works on my machine* syndrome. Threading issues are timing issues: Different computers with different specifications and other software running can cause threading issues to rarely appear on one computer and frequently on other. The user's workflow can also have a significant impact. For instance, clicking with the mouse too quickly can cause some issues to appear, while a slower action trigger can almost completely avoid the same problem.

If, by any chance, the developer's machine is not the one seriously affected, threading issues can be ignored for too long, and the more complex the application is, the harder it is to find all the places that need fixing. And only a single broken line can wreak havoc throughout the whole application.

Threading issues also tend to *move around.* Fixing an issue in one spot can move the problem down the line and make it appear in another unsafe piece of code. Sometimes, issues introduced by a partial fix can be even worse than the original issue. That does not necessarily mean that the fix for the original issue was a bad fix, just that in multithreading, a fix is not a fix until you fix all the affected parts. At times, making fixes to small pieces of the code will not cut it, and a complete redesign will be in order.

All of the above is not an argument against multithreading, but more of a warning to approach it seriously and avoid issues from the beginning, as they might be much harder to fix later on. Having said that, there is always the possibility that some code is not fully thread-safe. Even experienced developers make mistakes. But small localized mistakes can be corrected more easily than general design flaws that spread all over and may require big code refactoring.

You cannot approach writing multithreaded code from a "Let's make it work first, and let's add thread safety later on" standpoint. Thread safety is not a cosmetic, but rather a structural feature. It is not like putting a fresh coat of paint on a wall, it is about making strong walls that can support the whole house. If you fail to make the walls strong enough, no amount of paint will prevent the house from eventually collapsing when put under some stress, such as a small earthquake.

Chapter 27

Achieving thread safety

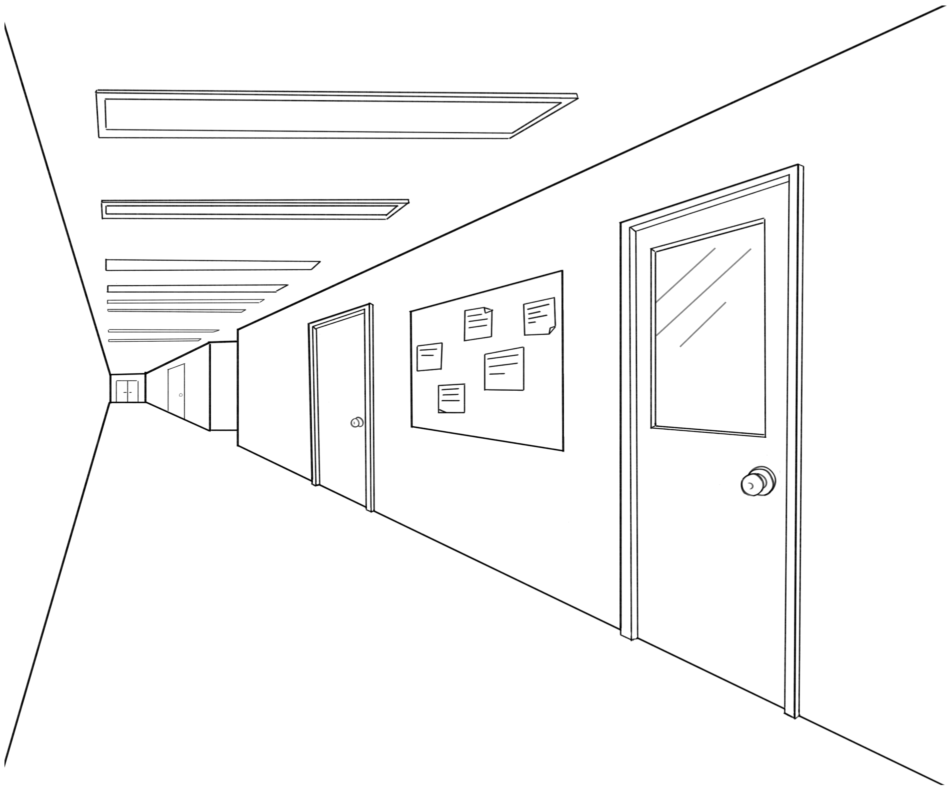
There are several ways of achieving thread safety. The first approach is to completely avoid race conditions and shared data, and the second is using various protection mechanisms that secure access to shared data and prevent unintended interactions.

- **Use immutable data**
- **Avoiding shared data**
  - use only local data
  - use thread local data
- **Protecting shared data**
  - thread synchronization and signalling
  - locks
  - atomic operations

Complexities in multithreading have two sources. The first is the ability to recognize whether some piece of code is thread-safe or not, and the second is deciding which of the numerous mechanisms to use in a particular situation.

Another problem arises while focusing on achieving thread safety in some code: It is very easy to get lost in all the necessary protectional ceremony and while you may get some part of local code right, it may have devastating effects on the bigger picture. In other words, you will not be able to see the forest for all the trees, and inevitably you will either get lost in it or will come out of it in the wrong place entirely.

While writing code is the end goal, it helps to visualize multithreading problems and solutions in a different, much simpler way.



To avoid getting lost in the woods, let's imagine our application is a building, our data is written on post-it notes pinned on the billboards inside, and each thread is a person that can move inside the building and read and write from those notes.

If only one person—a single thread—can ever enter the building, they can go around safely changing whatever is written on any of the notes.

But, if that board is standing in a hallway, and you let other people in, it can get messy if someone starts writing on the same note someone else is reading, or even worse if two people start writing simultanously on the same note.

It is almost immediately obvious that main problem in multithreading is not reading, but writing. If everyone would be just reading the notes, there would be no issues at all. You cannot corrupt data just by reading it. If anyone can read and write on the board at any time, this is where the trouble begins.

Another obvious thing is that not everyone needs access to everything. There is no need to pin your personal notes on the billboard. It is nobody else's business if you need to pick up the kids at noon. And they need the ability to change the time on that note even less.

The first step in multithreading is to compartmentalize the data and minimize the need for globally shared data. Leave the billboard as clean as possible, and let people carry their own notes for private use.

The next thing to do is figuring out which commonly shared data is only meant to be read. There is no problem if multiple people are reading the same note, as long as nobody is writing on it. Immutable data—data that can only be read and not written to—is always thread-safe.

Now you only need to handle shared data that can change during the application's lifetime while multiple threads have access. There are several ways to achieve this: one is thread synchronization, and another is to use locks and atomic operations.

Locks and atomic operations are a form of synchronization mechanism, controlling and *synchronizing* thread access to particular data and preventing simultaneous access, but I will use the term synchronization in a more narrow context, when I mean executing some piece of code in the context of a specific thread.

One can imagine synchronization like asking someone else to write a note and put it on some billboard. If you need to read that data again, you will need to ask that person to read it back to you. If someone else needs to read it, they will need to wait until the person in charge finishes his interaction with you, reading data to you or writing what you are telling them to write.

Beware of bad actors!

The problem with synchronization, is that the data is still written on the billboard, accessible to everyone. It is just a matter of courtesy to ask the appropriate person that is allowed to read and write on that billboard. If a rogue person comes in and starts reading, or even worse, writing on that billboard without permission, there is nothing anyone can do to prevent it.

This is a flaw of the synchronization approach, and of locking, as well. This is why multithreading can be so hard. If the data is on the billboard and not stashed in someone's pocket, even a single rogue actor can ruin everything.

Locking is in some ways similar to synchronization. But instead of having a billboard hanging in the hallway with a dedicated person reading and writing, it will be stored in a room behind locked doors.

To enter the room and read or write notes on the board, you need to take the key, unlock the doors, enter, lock the doors again, and then you are free to do whatever you need with the data. Other people must wait in front until you are done and out of the room. After you lock the doors behind you and put the key down, the next person in the line can pick it up and enter the room.

This is not very efficient, and in some cases, it can create long lines if too many people need to read and write on the board.

If, most of the time, people just need to read from the board and not write on it, you can have doors with a glass window. And you can have multiple people peeking through that glass and reading at the same time. But, if someone wants to change the data, they have to take the key, unlock the door, enter, and pull down the shades. While the shades are down, nobody outside can read the data. After the data on the board is successfully written, the person can lift the shades, get out of the room and leave the key. Everyone can read the new data on the board again.

If you think doors and locks are fully safe, compared to the synchronization metaphor, they are not. Namely, while those rooms do have doors with locks in one hallway, there is another hallway on the back with open portals, and no doors at all. Just like rogue actors can start writing on the billboard, they can also freely enter every room and make a mess if they feel like it.

Both synchronization and lock operations are slow and take some time in both code and our visualization model. They can create unnecessary lines when you just need to quickly swap one small note with another one and you want to make sure nobody else is reading and writing at that time. This is where atomic operations come in to speed things up.

This was a quick overview of how complex multithreading can get, and how easily things can go wrong. Discipline in protection is paramount, as well as keeping data private or immutable as much as possible, which can make a huge difference to the end result.

Chapter 28

General thread safety considerations

Thread safety is data safety. 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 in 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.

If multiple threads will only read data, regardless of the type, such access will always be thread-safe. If even one of the threads will write to either the variable (reference) or the associated memory, then unprotected access to such data is not thread-safe.

28.1 Memory allocation and deallocation

The default Delphi memory manager (FASTMM4) is thread-safe, and therefore allocating/deallocating memory is a thread-safe operation as long as you keep those operations balanced—in other words, if you allocate a memory location once, you are allowed to deallocate the same location only once. Since Delphi allows the use of custom memory managers, the thread safety of memory allocation depends on the memory manager. In general, memory managers are and should be thread-safe. This is merely a cautionary statement, because if, by any chance, you use a broken and thread-unsafe memory manager, then such a multithreaded application will be broken by design, and the only way to repair it would be replacing the defective memory manager.

28.2 Object construction and destruction

The compiler-generated code around calling a constructor and calling an object instance's destructor is thread-safe, but thread-safe destruction also depends on the destructor being called only once.

The thread safety of each particular class' construction and destruction additionally depends on the thread safety of all the other code (methods) involved in the process. If any of those methods contains thread-unsafe access to shared data, construction or destruction of such an object instance will not be a thread-safe operation, and such an object can only be safely constructed or destroyed from the context of the main thread.

Construction of an object involves the following stages:

1. memory allocation and initialization called in NewInstance
2. executing the appropriate chain of constructors
3. executing the AfterConstruction chain

Destruction of an object instance has the following stages:

1. executing the BeforeDestruction chain
2. executing the appropriate chain of destructors
3. cleanup of managed types and memory deallocation called in FreeInstance

The default initialization and cleanup code provided in the non-virtual `InitInstance` and `CleanupInstance` methods is thread-safe. The default implementation of `NewInstance` and `FreeInstance` in the base `TObject` class is also thread-safe, just like the default constructor, destructor, `AfterConstruction`, and `BeforeDestruction` methods. Constructing and destroying `TInterfacedObject` and `TThread` instances is also thread-safe. On the other hand, constructing and destroying `TComponent`-based classes is generally not.

For every other class, you will need to check whether the class overrides any of the methods involved in the construction/destruction process, and whether those overrides access shared data in an unsafe manner.

28.3 Assignment thread safety

As previously mentioned, assignment is generally not a thread-safe operation. However, assignment of naturally aligned integer types up to pointer size is an atomic operation. For larger sizes, assignment may be atomic, but there is no guarantee.

Naturally aligned type means that the address is a multiple of the type size. More specifically, byte-sized (8-bit) types are always aligned, 16-bit types are aligned if their address is a multiple of 2, 32-bit types are aligned if the address is a multiple of 4, and 64-bit types are aligned on 64-bit platforms if their address is a multiple of 8.

Alignment is always context-dependent. While the integer type will be naturally aligned on its own, if it is part of a complex data type, then alignment will depend on the alignment within that type and any other outer types.

For unaligned data types, assignment is atomic if data does not cross the cache line. If such unaligned data is part of a structure that fits into the cache line then assignment for its constituent elements will be atomic.

Note: Cache line sizes are CPU-dependent, so if you want to take advantage of atomic reads and writes within cache lines, you will need to consult the CPU manuals for specific targets.

28.3.1 What does atomic mean?

Atomic operations are operations that can fully complete, without any other thread interrupting and seeing a partially completed state.

For instance, you have two threads writing to an integer variable and reading it afterwards: one thread writing the value `$00000000`, and the other writing `$FFFFFFFF`. If the integer variable is aligned, then all reads and writes will be atomic, which means that no matter what value you write in that variable, it will always be fully written before other threads have a chance to read it. Also, reading will always be fully completed before other threads have a chance to start writing a new value. So, reading may give you either `$00000000` or `$FFFFFFFF` in both threads because the other thread could have overwritten the value, but you will never get some scrambled value like `$00FFFFFF`.

If the integer variable is not aligned and crosses the cache line, then you may experience *torn reads and writes*, and there is no guarantee what the actual value read or written may be. In other words, it is possible to get a mess like `$00FFFFFF`.

28.3.2 Atomic does not imply thread-safe

Even for simple types, atomic reads and writes don't automatically mean that a particular piece of code is thread-safe. Thread safety depends on specification, on what the correct functionality is in that particular context.

As an example, I will show exactly the same code for an imaginary game, where under one specification, atomic reads and writes will be enough to achieve thread safety, and under slightly different functionality requirements, the same code will not be thread-safe.

Let's say our game has 500 rooms stored in a global variable, where some parts of the room setup randomly change during play. Assuming that all individual data for the room is stored in byte-sized fields, all reads and writes of the individual fields will always be atomic. Populating random data takes time, so `PopulateRoom` will be called from a background thread. A different thread will run the `PresentRoom` procedure, called when the player enters a particular room and we need to present the current room state.

```
type
  TRoom = record
    ...
    Weather: byte;
    ...
    Character: byte;
  end;

  TRooms = array[0..499] of TRoom;

var
  Rooms: TRooms;

procedure PopulateRooms;
var
  i: Integer;
begin
  for i := 0 to High(Rooms) do
    begin
      ...
      Rooms[i].Weather := Random(255);
      Rooms[i].Character := Random(255);
    end;
end;

procedure PresentRoom(Index: Integer);
begin
  case Rooms[Index].Weather of
    0 : Sunshine;
    1 : Fog;
    2 : Rain;
    ...
  end;
  case Room[Index].Character of
    0 : Unicorn;
    1 : Bunny;
    2 : Sparrow;
    ...
  end;
end;
```

Under the first specification, all room fields are independent from each other. So it only matters that fields can be written and read in an atomic manner.

While presenting the room, we can have the situation that at the moment we enter the `PresentRoom` code, the room data contains *Rain* and *Bunny* values. But by the time we've finished presenting *Rain*, the background thread running the `PopulateRooms` code will change those values to *Sunshine* and *Unicorn*. Instead of presenting *Rain* and *Bunny*, we will show *Rain* and *Unicorn* to the player. But from the game's perspective, that is acceptable behavior because we are calling `PresentRoom` only once when the player enters the room, and we can show any possible combination of weather and characters.

If we change the game specification and say that *Unicorn* cannot appear when weather is *Fog* or *Rain*, and we adjust the code in `PopulateRooms` to reflect that, then the `PresentRoom` code can still run into a situation where we start with *Rain* and *Bunny* and end up presenting *Rain* and *Unicorn*, which, under the new specification, represents buggy behavior, and therefore our code is not thread-safe by the definition of thread safety, even though all reads and writes are atomic.

To make the game code thread-safe under the new specification, we would need to add some synchronization mechanism that would allow us to consistently read and write the whole room data, or pack dependent data in some way to have atomic reads and writes for dependent data only.

28.3.3 Reference assignments

For some reference types, assignment involves more code than replacing one pointer value with another. While an aligned reference assignment will be atomic, the side-effects involved will generally result in thread-unsafe code as shown in the *Wrong assumptions* chapter.

While there is plenty of code concerning reference types (including managed, reference-counted types), where allowing a reference to be written by multiple threads is not thread-safe without additional protection, there are a few scenarios where clearing a reference (assigning `nil`) is thread-safe.

Reference counting in managed types does have one protection. It protects the consistency of the reference count variable by using atomic increments and decrements. If you start with multiple threads already holding a strong reference, and they just need to be able to safely clear that reference and eventually cause the releasing of the object instance or clearing a string or dynamic array's memory, then such a scenario does not need any additional protection, and it will be thread-safe.

But it is imperative that taking a strong reference to an interface, string, or dynamic array is done before any of the threads starts clearing references, or at least that the original strong reference used for setting up each thread's reference copy will not be modified (cleared) before all threads have made one.

Dynamic arrays and strings have that protection built in, but for object instances, thread safety depends on the code in the `_Release` method. `TInterfacedObject` implements such thread-safe clearing and ensures that the destructor will be called only once, when a reference count decrement results in 0, which implies that there are no more strong references to that object instance:

```
function TInterfacedObject._Release: Integer;
begin
  Result := AtomicDecrement(FRefCount);
  if Result = 0 then
  begin
    // Mark the refcount field so that any refcounting during
    // destruction doesn’t infinitely recurse.
    __MarkDestroying(Self);
    Destroy;
  end;
end;
```

Chapter 29

Use immutable data

Read-only data is thread-safe. As long as shared data is immutable, it will not require any additional protection. However, immutability also requires that references to such data, variables, are also read-only. In other words, variable assignment is not a thread-safe operation, except for some basic simple types under certain conditions.

While some languages have the concept of `final` variables, that can be assigned only once, Delphi does not fully support this concept, as it only supports constant variable declaration for some data types.

So even when you can have some immutable data referenced by some variable, most of the time it will require discipline from the developer's side not to change the content of the reference itself.

```
const
  X = 42;
  Msg = 'Some message';
```

For some more complex data types, you can use typed constants to create truly immutable data.

```
  ZeroPoint: TPoint = (X: 0; Y: 0);
```

Delphi also supports writeable typed constants, ones that are compiled with the `{$J+}` or `{$WRITEABLECONST ON}` compiler directives. Writeable typed constants are variables in disguise—you can freely assign new values to them—and as such, are not thread-safe. Writeable typed constants are provided for backwards compatibility, and using them should be avoided.

Even if using constants or typed constants is not an option, you can still achieve immutability of inner data. That way, you will only need to pay attention to the variable itself, you don't need to worry about accidentally modifying the contents it holds. Strictly speaking, anything that is mutable in any way, even if it is just a variable, is no longer immutable and therefore loses its inherent thread safety, but every additional layer of protection can and will reduce potential issues.

Immutability also depends on the context. For instance, immutable data stored in a mutable variable can become fully immutable, if the variable is passed as a constant to some other method:

```
type
  TImmutableRecord = record
  strict private
    FIntValue: Integer;
  public
    constructor Create(AIntValue: Integer);
    property IntValue: Integer read FIntValue;
  end;

constructor TImmutableRecord.Create(AIntValue: Integer);
begin
  FIntValue := AIntValue;
end;

procedure Test;
var
  Data: TImmutableRecord;
  IntVal: Integer;
begin
  IntVal := 50;
  Data := TImmutableRecord.Create(IntVal);
  IntVal := 100;
  Data.IntValue := 200; // E2129 Cannot assign to a read-only property
end;
```

Unlike with value types, achieving mutability with reference types stored in a record is a bit trickier, and not all reference types behave the same. Strings in Delphi support COW (copy on write), so they behave differently and give a tad *more* immutability than other reference types, such as arrays:

```
type
  TImmutableRecord = record
  strict private
    FIntValue: Integer;
    FStrValue: string;
    FArrValue: TBytes;
  public
    constructor Create(AInt: Integer; const AStr: string; const AArr: TBytes);
    property IntValue: Integer read FIntValue;
    property StrValue: string read FStrValue;
  end;

constructor TImmutableRecord.Create(AInt: Integer; const AStr: string;
  const AArr: TBytes);
begin
  FIntValue := AInt;
  FStrValue := AStr;
  FArrValue := AArr;
end;

procedure Test;
var
  Data: TImmutableRecord;
  IntVal: Integer;
  StrVal: string;
  ArrVal: TBytes;
begin
  IntVal := 50;
  StrVal := 'abc';
  ArrVal := [1,2,3];

  Data := TImmutableRecord.Create(IntVal, StrVal, ArrVal);

  IntVal := 60;
  Delete(StrVal, 1, 1);
  Delete(ArrVal, 1, 1);
end;
```

```
Name            Value
FIntValue     50
FStrValue     'abc'
FArrValue     (1, 2, 3)

Name            Value
Data        (50, 'abc', (1, 3, 3))
    FIntValue   50
    FStrValue   'abc'
    FArrValue   (1, 3, 3)
```

And it is clear that this variant of an immutable record is not so immutable after all:

```
type
  TImmutableObject = class
  strict private
    FIntValue: Integer;
  public
    constructor Create(AIntValue: Integer);
    property IntValue: Integer read FIntValue;
  end;

constructor TImmutableObject.Create(AIntValue: Integer);
begin
  inherited Create;
  FIntValue := AIntValue;
end;
```

The class-based variant of immutable records is rather similar. The same rules for private fields apply to both classes and records. Immutability is not inherently given, but it can be achieved by limiting access to data to read-only and creating local copies of mutable data instead of storing just references. While there is nothing stopping you from storing object references in records, it is way more common for classes to have object fields. Limiting access to object properties, in combination with the property being an immutable class on its own, can give an additional level of safety, but achieving absolute immutability is not possible.

```
type
  TImmutableObject = class
  strict private
    FObjValue: TOtherImmutableObject;
  public
    ...
    property ObjValue: TOtherImmutableObject read FObjValue;
  end;

var
  Data: TImmutableObject;
  ...
  Data.ObjValue.Free;
```

Not only have you changed your immutable data, you have effectively killed it. Calling `Free` on any property is an extremely bad coding practice regardless of threading and mutability, but in the context of achieving true immutability, it is a loophole that prevents you from having really immutable object fields.

Using interface references as fields suffers from similar issues. You cannot call `Free` on an interface reference, but you can call the `_Release` method. Which, again, is not something that is a good practice or often used, but another loophole.

Some data you can make truly immutable. Some you simply cannot. You can add additional safety layers that will prevent most misuses, but you cannot achieve absolute protection. If you overdo your protection, you can easily end up with layers upon layers that will not only make reading and maintaining the code harder and introduce performance penalties, but will also not achieve your initial goal—truly immutable and therefore thread-safe data.

Sometimes, like in anything else, you just need to find good measure.

Chapter 30

Avoiding shared data

30.1 Use only local data

Multithreading issues arise around the usage of shared data—data accessed and handled by multiple threads. The best and most obvious way to avoid such issues is not to have any shared data at all.

If a background task has all the needed data to perform the task stored as a local copy, and returns a copy of the data that can then safely be applied in the context of the caller thread (usually the main thread), you will not have any problems with data protection, because there is nothing to protect.

Be careful when using reference types (objects, interfaces, dynamic arrays...), and make deep copies if needed, because copying the reference alone will not create a copy of the associated data, and other threads can still freely access that data and make modifications, unless the contents of such a reference type are immutable.

When creating deep copies or even just copying the reference itself, keep in mind that assignment and copying are not atomic operations, and if multiple threads have write access to the original reference variable or its content, the whole operation will not be thread-safe and will need additional protection (locking). However, if creating the copy is done in the context of the original thread, and the additional (background) thread does not have access to the data at the point of copying, then making a copy is thread-safe:

```
procedure DoFoo(AObjValue: TFoo);
var
  ObjValueCopy: TFoo;
begin
  ObjValueCopy := TFoo.Create;
  try
    ObjValueCopy.Assign(AObjValue);
    TTask.Run(
      procedure
      begin
        try
          // do something with ObjValueCopy
          ObjValueCopy.Modify;
          // depending on with what Apply actually does, Apply
          // might need to be synchronized with the main thread
          Apply(ObjValueCopy);
        finally
          ObjValueCopy.Free;
        end;
      end);
  except
    ObjValueCopy.Free;
    raise;
  end;
end;
```

There is no need to create a separate local copy of value types when they are passed as value or constant parameters.

```
// using AIntValue and ABoolValue within DoFoo is thread-safe
procedure DoFoo(AIntValue: Integer; const ABoolValue: Boolean);
...
```

30.2 Use thread-local data

Delphi supports thread-local variables using the `threadvar` keyword. They act like any other global variable, except that each thread will get its own local, zero-initialized copy.

```
threadvar X: Integer;
```

There are a few things to consider when using thread-local variables. While the variable itself is stored as a local copy, depending on the variable type and what you store inside and how you use it, they can still allow access to shared data in an unsafe manner. For instance, if you have a thread-local variable that is an object reference, and you store a reference to the same object instance across multiple threads, all threads will be able to access and change that instance's data.

When using managed types for thread-local variables, Delphi will not automatically manage their memory, and before the thread finishes you will need to manually release their memory. For instance, by assigning an empty string in the case of string variables, or setting interfaces or dynamic arrays to `nil`, or setting a Variant to unassigned.

Using such thread-local variables is not a recommended practice.

Since Delphi has the `TThread` class with encapsulated threading support, a much better option for storing thread local data is creating a custom class inherited from `TThread` with all the necessary private fields that will then *act* like thread-local data and can be safely accessed from the associated thread. That locality highly depends on their visibility specifiers. If you allow such a field to be publicly accessible, then it will not represent thread-local data in the full meaning of that term.

Just like with local data, or thread-local variables, fields in extended `TThread` classes are not inherently thread-safe—it all depends on their type and what and how you have stored inside:

```
type
  TFooThread = class(TThread)
  private
    FFoo: TFoo;
  public
    constructor Create(AFoo: TFoo);
    destructor Destroy; override;
    procedure Execute; override;
    property Foo: TFoo read FFoo;
  end;


constructor TFooThread.Create(AFoo: TFoo);
begin
  inherited Create;
  FFoo := TFoo.Create;
  FFoo.Assign(AFoo);
end;

destructor TFooThread.Destroy;
begin
  FFoo.Free;
  inherited;
end;
```

```
procedure TFooThread.Execute;
begin
  FFoo.Modify;
  ...
end;
```

The thread constructor runs in the context of the calling thread, and it is safe to do data copying inside. Regardless of whether the thread is being constructed as suspended or not, thread execution will not start until the constructor chain is executed, so you don't need to worry about code in the constructor interfering with code in the `Execute` method. The `Foo` property containing the final result (or any other thread field) must not be used from outside the `TFooThread` class, unless it is accessed from an `OnTerminate` event handler that runs in the context of the main thread.

Chapter 31

Initialization pattern

Read-only data is thread-safe, and as long as you can initialize such data before everyone starts reading, all is good, but sometimes a combination of lazy initialization patterns and multithreading can cause a situation where multiple threads might trigger data initialization. In such cases, initialization can cause threading problems and needs to be protected. After initialization is completed, such data will be safe to use in a read-only context when accessed from multiple threads, and no additional protection will be necessary.

You can always protect such data with locks, but locks will introduce an unnecessary performance penalty every time you access that data in the application. You can get a bit more speed by using a more appropriate lock type, like a multi-reader lock, but a lock is still a lock, and it will only take you so far.

Thread-unsafe lazy initialization

```
  TFooFactory = class
  private
    class var FInstance: TFoo;
    class destructor ClassDestroy;
    class function GetInstance: TFoo;
  public
    property Instance: TFoo read GetInstance;
  end;

class destructor TFooFactory.ClassDestroy;
begin
  FreeAndNil(FInstance);
end;
```

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

The problem with the regular lazy initialization pattern is in the `GetInstance` method. If multiple threads call it while the instance field is not yet initialized, both threads can separately create a new instance and assign it to the `FInstance` field. Not only you will leak one object instance in such a case, but much worse bugs may happen if one thread starts to use one instance, changing its state, and then the other thread subsequently replaces the singleton field with the other instance, causing all changes to be lost, sitting in a now-leaked object.

This problem can be solved in the following manner: If the instance is not initialized, multiple threads will be able to create separate singleton instances, but instead of directly assigning the newly created instance to the singleton field, it will be assigned to a local variable. Then with the help of the atomic compare and exchange (assign) methods, only one (the first) thread will be able to successfully set the `FInstance` field. Any thread that fails to assign the local variable to the singleton field can perform cleanup and release the unnecessary instance.

Since the singleton field is set only once, all threads will use the same object instance, and all changes will be accounted for:

Thread-safe lazy initialization

```
class function TFooFactory.GetInstance: TFoo;
var
  LInstance: TFoo;
begin
  if FInstance = nil then
  begin
    LInstance := TFoo.Create;
    if TInterlocked.CompareExchange<TFoo>(FInstance, LInstance, nil) <> nil then
      LInstance.Free;
  end;
  Result := FInstance;
end;
```

More information about atomic operations can be found in the *Atomic (interlocked) operations* chapter.

Chapter 32

Protecting shared data

32.1 Synchronization primitives (objects)

Synchronization primitives are a rather broad category of various synchronization mechanisms (objects) that are used to coordinate threads when accessing shared resources. Since thread and process synchronization are also highly coupled with the synchronization mechanisms provided on the operating system level, much of the commonly used terminology for particular synchronization objects will be heavily influenced by the operating system used, and its meaning may vary depending on the context.

Besides what is provided on the OS level, many languages add their own synchronization mechanisms, that are either part of the language itself, or within the core runtime libraries.

Some synchronization mechanisms belong to a higher level of synchronization mechanisms, and they combine various low level synchronization primitives to achieve more specific functionality.

Since the main purpose of this book is to provide an overview of the most important concepts used in common real-life code scenarios, and not to give you all the material you will ever need for winning a synchronization nitpicking contest, the terminology used will be the one commonly used in Delphi programming, with emphasis on Windows programming. While the general synchronization concepts are universal, there are various different implementations across platforms, and some basic building blocks may vary in some finer details or have slightly different names. Delphi started as a Windows tool, so Windows terminology prevails in the core Delphi frameworks even when they are used for cross-platform development.

For instance, *critical section* is a term used to define a section of code that will be protected from simultaneous access from multiple threads during thread synchronization, but that term is more often used to denote an exclusive lock object on Windows.

Similarily, *synchronization objects* in the Windows API describes a narrower set of objects: events, semaphores, mutexes, and waitable timers that can specifically be used in various `Wait` functions to coordinate execution of multiple threads and in multi-process communication.

Even if you focus on a single operating system and its synchronization mechanisms, you will find that that list is not exhaustive, and that with time, new mechanisms are added to provide very specific functionality that allows fine-tuning performance in specific scenarios.

A basic functional categorization of synchronization mechanisms:

- Locks
- Signalling—events, semaphores, condition variables, timers
- Interlocked (atomic) operations

Besides the synchronization functionality that is built into some core types and classes, the Delphi RTL also provides a collection of various synchronization mechanisms in the `System.SysUtils` and `System.SyncObjs` units. For other more specific, possibly OS-related functionality, you will either have to directly use the specific platform APIs, or rely on some 3rd-party library.

Since various synchronization mechanisms often rely on underlying OS mechanisms, the performance and behavior of the same object can greatly vary between platforms.

There are many synchronization object categories and each serves a particular purpose. Different mechanisms with slightly different behaviors give us the ability to fully optimize code for its specific needs.

Choosing the best synchronization object category and the best implementation for a particular scenario is not easy, and the differences can often seem almost non-existent to the untrained eye. Besides experience, the next most helpful things are reading the documentation and comparing the specifications. And last but not least, it is always prudent to run some tests before making the final choice.

When it comes to implementation details, synchronization objects can be implemented as classes (with or without reference counting) or records. Every synchronization object needs to be constructed before used (in the case of records, initialization and finalization may be automatic) and destroyed when no longer needed. If the synchronization object is implemented with a reference-counted class, make sure you use it only through interface references, to ensure proper reference counting and automatic destruction when the instance goes out of scope.

It is important to keep the same instance of the synchronization object during the whole lifespan of the protected resource. If you create/destroy the instance on every access, it will be useless.

For example, if you want to make a thread-safe list, you can use a lock instance that is declared as a field in the list class, and then that same instance is used in all methods, but you cannot use a locally-declared instance in each method.

Correct code

```
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;
    ...
  end;

constructor TIntegerList.Create;
begin
  inherited;
  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;
```

Incorrect code

```
type
  TIntegerList = class
  protected
    FItems: array of Integer;
  public
    procedure Add(Value: Integer);
    function Exists(Value: Integer): Boolean;
    ...
  end;

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

Coordinating threads and synchronizing access to data comes with its own set of problems, such as: *deadlocks* when one thread waits for a resource acquired by another thread, and that second thread needs to acquire a resource already acquired by the first one; *livelocks* where the threads involved are constantly changing their state in response to another thread's state, but they never make progress; *resource starvation* when some threads don't get chance to acquire shared resources, because other threads have constantly been given access to the resources first; *busy waiting* where a thread constantly polls whether it can access a resource and, in doing so, reduces the available time for other threads that can do more meaningful work.

The following example shows a deadlock scenario. Two threads need to acquire a shared resource, but they each only manage to acquire one, so they are stuck waiting for each other, and none of the threads can make any progress:

```
procedure TMainForm.DeadlockBtnClick(Sender: TObject);
var
  Lock1: TCriticalSection;
  Lock2: TCriticalSection;
begin
  Lock1 := TCriticalSection.Create;
  Lock2 := TCriticalSection.Create;

  TThread.CreateAnonymousThread(
    procedure
    begin
      Lock1.Enter;
      try
        OutputDebugString('THREAD1 LOCK1 ACQUIRED');
        Sleep(1000);
        Lock2.Enter;
        try
          OutputDebugString('THREAD1 LOCK2 ACQUIRED');
          OutputDebugString('THREAD1 WORKING');
        finally
          Lock2.Leave;
          OutputDebugString('THREAD1 LOCK2 RELEASED');
        end;
      finally
        Lock1.Leave;
        OutputDebugString('THREAD1 LOCK1 RELEASED');
      end;
    end).Start;

  TThread.CreateAnonymousThread(
    procedure
    begin
```

```
        Lock2.Enter;
        try
          OutputDebugString('THREAD2 LOCK2 ACQUIRED');
          Sleep(500);
          Lock1.Enter;
          try
            OutputDebugString('THREAD2 LOCK1 ACQUIRED');
            OutputDebugString('THREAD2 WORKING');
          finally
            Lock1.Leave;
            OutputDebugString('THREAD2 LOCK1 RELEASED');
          end;
        finally
          Lock2.Leave;
          OutputDebugString('THREAD2 LOCK2 RELEASED');
        end;
      end).Start;
    // this example leaks both locks, but this is not relevant
    // for demonstrating the deadlock problem
  end;
```

Running the above code will cause a deadlock and produce the following output, showing that the threads will never succesfully complete their work:

```
THREAD1 LOCK1 ACQUIRED
THREAD2 LOCK2 ACQUIRED
```

The following example shows a livelock scenario. Threads try to behave and prevent a deadlock situation. If the thread cannot acquire all needed resources it will release the already-acquired resource(s) to prevent a deadlock situation. While a deadlock indeed does not happen, threads are still blocked from doing any meaningful work, because none of the threads can successfully acquire all the needed resources and complete the task. Both threads will be stuck in the loop. Eventually, some timing difference might result in threads coming out of the livelock, but that might not happen in a reasonable amount of time.

```
procedure TMainForm.LivelockBtnClick(Sender: TObject);
var
  Lock1: TCriticalSection;
  Lock2: TCriticalSection;
begin
  Lock1 := TCriticalSection.Create;
  Lock2 := TCriticalSection.Create;

  TThread.CreateAnonymousThread(
    procedure
    begin
      while True do
        begin
          if Lock1.TryEnter then
            begin
              OutputDebugString('THREAD1 LOCK1 ACQUIRED');
              Sleep(1000);
              if Lock2.TryEnter then
                begin
                  OutputDebugString('THREAD1 LOCK2 ACQUIRED');
                  OutputDebugString('THREAD1 WORKING');
                  Break;
                end
              else
                begin
                  Lock1.Release;
                  OutputDebugString('THREAD1 LOCK1 RELEASED');
                end;
            end;
        end;
      Lock1.Leave;
      OutputDebugString('THREAD1 LOCK1 RELEASED');
      Lock2.Leave;
      OutputDebugString('THREAD1 LOCK2 RELEASED');
    end).Start;

  TThread.CreateAnonymousThread(
    procedure
    begin
      while True do
        begin
          if Lock2.TryEnter then
            begin
              OutputDebugString('THREAD2 LOCK2 ACQUIRED');
              Sleep(500);
              if Lock1.TryEnter then
                begin
                  OutputDebugString('THREAD2 LOCK1 ACQUIRED');
                  OutputDebugString('THREAD2 WORKING');
                  Break;
                end
              else
                begin
```

```
                    Lock2.Release;
                    OutputDebugString('THREAD2 LOCK2 RELEASED');
                  end;
              end;
          end;
        Lock1.Leave;
        OutputDebugString('THREAD2 LOCK1 RELEASED');
        Lock2.Leave;
        OutputDebugString('THREAD2 LOCK2 RELEASED');
      end).Start;

    // this example leaks both locks, but this is not relevant
    // for demonstrating the livelock problem
  end;
```

The output will be a neverending sequence of:

```
THREAD1 LOCK1 ACQUIRED
THREAD2 LOCK2 ACQUIRED
THREAD2 LOCK2 RELEASED
THREAD2 LOCK2 ACQUIRED
THREAD1 LOCK1 RELEASED
THREAD1 LOCK1 ACQUIRED
THREAD2 LOCK2 RELEASED
```

32.2 Thread synchronization

One way of preventing issues with simultaneous access to some data from multiple threads is to access that data solely from the context of one thread. This can be achieved by a mechanism called thread synchronization, where threads mutually coordinate the execution of a particular piece of code. Without cooperation, you cannot force any thread to execute anything in its context, so such a thread must have a mechanism in place that will allow running some piece of code that the other thread wants it to run. In other words, the first thread needs to periodically check whether there is some queued code waiting for execution.

In terms of Delphi programming, this terminology is more specifically used in relation with the main thread and running particular code in the context of the main thread, which has synchronization checks as part of handling main loop events.

There are several methods in the `TThread` class which you can use to synchronize (run) code in the main thread's context:

`Synchronize` methods, which are blocking calls—control to the calling thread will be returned after the synchronized code finishes running on the main thread—and non-blocking `Queue` methods, that put code in a synchronization queue and immediately return control to the caller.

If the `Synchronize` or `Queue` methods are called from within the main thread, they will just run the code immediately. If your code is running on the main thread, and you really want to force code to be queued rather than executed immediately, you can use the `ForceQueue` methods. `ForceQueue` also allows specifying a delay parameter, so that that code will run only after a certain number of milliseconds have elapsed.

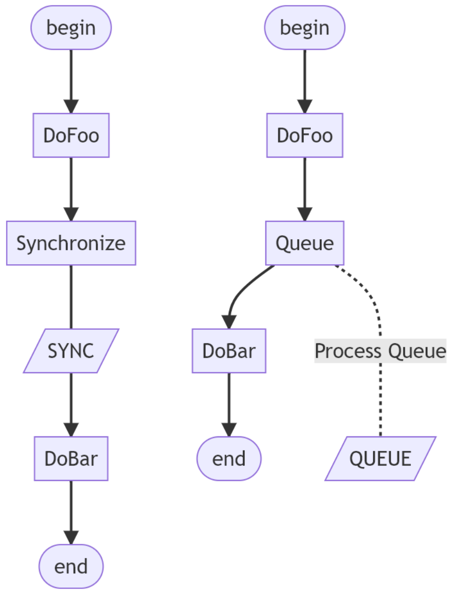
The `AThread` parameter is used to specify the calling thread if you need to access that information while running in the main thread, and if specified, it is also used in `RemoveQueuedEvents` for removing queued events if the specified thread has stopped running before the queued code had chance to run. If you don't need to know the calling thread, you can just pass `nil` as the `AThread` parameter.

```
procedure Synchronize(AMethod: TThreadMethod);
procedure Synchronize(AThreadProc: TThreadProcedure);

class procedure Synchronize(const AThread: TThread; AMethod: TThreadMethod);
class procedure Synchronize(const AThread: TThread;
  AThreadProc: TThreadProcedure);

class procedure Queue(const AThread: TThread; AMethod: TThreadMethod);
class procedure Queue(const AThread: TThread; AThreadProc: TThreadProcedure);
class procedure ForceQueue(const AThread: TThread;
  const AMethod: TThreadMethod; ADelay: Integer = 0);
class procedure ForceQueue(const AThread: TThread;
  const AThreadProc: TThreadProcedure; ADelay: Integer = 0);
```

Calling synchronization methods is pretty straightforward:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  TThread.CreateAnonymousThread(
    procedure
    begin
      DoFoo;
      TThread.Synchronize(nil,
        procedure
        begin
          Memo1.Lines.Add('SYNC');
        end);
      DoBar;
    end).Start;
end;
```

```
procedure TMainForm.ButtonClick(Sender: TObject);
var
  Thread: TThread;
begin
  Thread := TThread.CreateAnonymousThread(
    procedure
    begin
      DoFoo;
      TThread.Queue(nil,
        procedure
        begin
          Memo1.Lines.Add('QUEUE');
        end);
      DoBar;
    end);
  Thread.Start;
end;
```



Queue is not a blocking call, and the queued anonymous method will run at an unspecified time in the context of the main thread. At that moment, the anonymous thread might already have finished running, and what we passed as the AThread parameter will have consequences for any queued methods at that point.

In the previous example, we passed nil as the AThread parameter. That means the queued code is not associated with any thread, and it will always run when the main thread gets to process queued events. Regardless of whether the thread is already destroyed at that point or not, the word “QUEUE” will appear in the memo control.

```
      TThread.Queue(Thread,
```

If we pass `Thread` as a parameter to the `Queue` method, the queued code will be associated with that thread, and when the thread is destroyed, its pending associated event(s) will be removed from the queue. In the above example, whether a queued event will run depends on how fast the execution of the `DoBar` method called after `TThread.Queue` is.

If `DoBar` is an empty method, the thread will probably finish running and clean its events before the main thread will have a chance to observe the queue and run the code, so the code will never run and “QUEUE” will not appear in the memo.

If we add some longer-running code in `DoBar`, or simply call `Sleep`, we will artificially prolong the thread’s life, giving the main thread the chance to run queued events, and “QUEUE” will be shown in the memo.

This is extremely important from the perspective of memory management. An anonymous method will capture all accessed object references, but we need to be sure that anything we capture will live long enough, and will still represent valid object instances when the queued anonymous method actually runs.

Of course, synchronization also doesn’t protect us from accessing invalid data, but queuing is more error-prone. For instance, thread-local object instances will be available in synchronized code, but they may not be alive in queued code. If we are dealing with such objects, we must call the `Queue` method and pass the thread as a parameter instead of `nil`, to clear queued methods that can access invalid data. If we really need to run all queued methods at all costs, then we need to handle the lifetime of accessed object instances in a different way, or use different types that provide some form of automatic memory management, like records or reference-counted classes.

Incorrect code

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
  TThread.CreateAnonymousThread(
    procedure
    var
      Data: TDataObject;
    begin
      Data := TDataObject.Create;
      try
        ProcessData(Data);
        TThread.Queue(nil,
          procedure
          begin
            // Schrödinger’s data--maybe it is alive, maybe it is not
            // you won’t know until you run this code
            // and it can behave differently every time
            ShowData(Data);
          end);
        DoSomethingElse(Data);
      finally
        Data.Free;
      end;
    end).Start;
end;
```

32.3 Locks

Locks prevent simultaneous access to shared data (resources) from multiple threads. More specifically, they protect data by coordinating access to some piece of code (also called *critical sections*) that operates on the data. There is no limit in terms of how large a section of code you lock can be, but a general rule is that you should avoid locking too much code, because it can create thread contention, causing performance issues and deadlocks. On the other hand, if you don’t put a lock around enough code in a particular context, the lock will not achieve its purpose, and will fail to protect the integrity of the data that needs protection.

The common functionality that all locks share is two methods that must be called around the code we want to protect:

- acquire (enter)
- release (leave)

Entering and leaving locks should be protected with `try...finally` blocks, because any exception raised will otherwise lock the wrapped resources forever:

```
Lock.Enter;
try
  ...
finally
  Lock.Leave;
end;
```

32.3.1 No lock

No lock is a poor solution for any protection. The reality is, that shared data becomes shared data only when multiple threads can and will access that data simultaneously and some threads will modify the data. Usually, plenty of data in an application will be accessible to multiple threads, but only one thread will ever use it. Such data is at no risk, and does not represent shared data that needs protection. At least not explicit protection in code. The same is valid for read-only data that is not immutable.

No lock is mental protection. It means that if you have access to some variable in multithreaded code, you are not allowed to use it everywhere willy-nilly. That assumes having control over your code, knowing what purpose such *potentially* shared data serves, and where and when it is used and how.

Knowing the purpose implies having some kind of design specifications. They can be stored in your head, but that is a fairly bad place for any kind of documentation. If you don't have any specifications, and you are just making some proof-of-concept code, at least add some comments to such variables about acceptable use cases. Even if you think it is obvious, your future you will thank you, as will anyone that will ever have to maintain that code after you.

Proof-of-concept code very often outlives its original purpose, and ends up in production as-is. Multithreading issues can be very hard to track down and fix at that point. And even a single bad line of code can ruin everything.

Changing any variable (piece of data) from *no protection* to *needs protection* is a huge code refactoring, even though it may not seem like one. It requires going manually through every line of code where that data is used, and verifying that code is thread-safe, adding protection code as needed. This includes inspecting how some data is used when passed to various methods that can potentially mutate the data.

32.3.2 Recursive lock

A recursive, or reentrant lock is a lock variant where the thread that originally locked a resource is allowed to lock it again, and again, and again. These will typically keep track of the amount of times the resource has been acquired/released by the thread, only fully unlocking it when the number of releases is equal to the number of acquisitions. And, of course, if it's greater than the number of acquisitions... Something is terribly wrong.

Keep in mind that the underlying principle can be used with either exclusive or multi-reader locks, and does not in itself imply the use of one or the other.

Calling non-reentrant locks from the same thread that already acquired that lock will always cause a deadlock.

32.3.3 Exclusive lock

This kind of lock will allow access to the data for only one thread at a time, regardless of whether data will be read or written.

This is the least efficient kind of lock, and can create a lot of contention if you have too many threads waiting in line.

The most commonly used exclusive reentrant lock in Delphi is `TCriticalSection`.

Besides the standard `Enter` method, which unconditionally tries to acquire a lock, `TCriticalSection` also has a `TryEnter` method, which will try to acquire a lock, but if it is not successful, it will return `False` and will not block the calling thread while it waits for other threads to release the lock. If the lock is successfully acquired, `True` will be returned, and the lock will behave the same as if it had been acquired using the plain `Enter` method. You should call `Leave` only if the lock was successfully acquired.

`TryEnter` is usually combined with looping and trying to enter the lock multiple times. However, you should carefully write that logic, because instead of deadlocking the threads, you can end up livelocking them.

```
var
  Lock: TCriticalSection;

  if Lock.TryEnter then
    try
      ...
    finally
      Lock.Leave;
    end
  else
    begin
      // lock is not successfully acquired
    end;
```

32.3.4 Readers-writer or multi-reader lock

If data needs to be read very often, but seldom written, it is a candidate for being protected by this lock type, which allows simultaneous read (shared) access from multiple threads, but acts like an exclusive lock when a thread needs to write to it.

RW locks can come in many flavors. Each type of locking request—read or write—can be independently reentrant or not. Also some locks support upgrading read access to write access, while others do not. Again, like with any other lock, there might be differences in behavior between platforms for the same lock type.

Upgrading read to write access requires all readers (except the one trying to upgrade) to release read lock before write access is granted. If two threads try to do the upgrade simultaneously, they will deadlock because both will hold read access and will not be able to acquire write access.

Delphi has a few implementations of multi reader locks: `TMultiReadExclusiveWriteSynchronizer`, and its more lightweight variant, `TLightweightMREW`, introduced in Delphi 10.4.1 Sydney.

There is also something called `TSimpleRWSync`, but it is basically a wrapper around an object that is used as a lock via the `TMonitor` mechanism. It behaves like an exclusive lock, and does not offer any specific functionality regarding allowing multiple readers.

RW locks commonly have a `BeginRead`-`EndRead` method pair for requesting read access, and a `BeginWrite`-`EndWrite` pair for requesting write access.

```
Lock.BeginRead;
try
  ...
finally
  Lock.EndRead;
end;
```

```
Lock.BeginWrite;
try
  ...
finally
  Lock.EndWrite;
end;
```

`TLightweightMREW` also implements the `TryBeginRead` and `TryBeginWrite` methods, which will try to acquire read or write access, and will return a Boolean indicating whether the attempt was successful or not. If the access was granted, then the appropriate `Endxxx` must be called to balance the `TryBeginxxx` call.

Various RW lock implementations in the Delphi RTL across different platforms exhibit the full palette of different behaviors and approaches. This is a perfect example why reading the documentation—and even the implementation—is a must.

`TMultiReadExclusiveWriteSynchronizer` implements RW functionality only on the Windows platform, while on other platforms it uses `TSimpleRWSync`, which is basically a wrapped object used as a monitor synchronization object and always acts like an exclusive lock, even when asked only for read access. It is reentrant for both reading and writing and allows converting read access to write access.

When upgrading read access to write access, there is a possibility that another thread that requested write access had access granted, and that the protected resource was modified during that time. `BeginWrite` is a function that returns a Boolean, and the result of the call will be

True if another thread hadn't been granted write access in the meantime. If another thread had been granted write access, you need to treat the protected resource as modified, and re-read it again before making any changes. For instance, if you were in the middle of iterating through a list, the list's contents might have changed in the meantime, and you will need to restart the iteration. Depending on the code, it might be simpler and faster to just request write access from the beginning.

Upgrading read access to write access:

```
var
  Lock: TMultiReadExclusiveWriteSynchronizer;

  Lock.BeginRead;
  try
    ...
    NotModified := Lock.BeginWrite;
    try
      ...
    finally
      Lock.EndWrite;
    end;
  finally
    Lock.EndRead;
  end;
```

TLightweightMREW is a wrapper around the OS-provided RW locking mechanisms on all platforms, and is a faster replacement for TMultiReadExclusiveWriteSynchronizer. It is implemented as a managed record, which makes it require less memory and avoid additional heap allocation, unlike TMultiReadExclusiveWriteSynchronizer, TSimpleRWSync, and TCriticalSection, which are implemented as classes.

TLightweightMREW supports reentrant read access, but not write access. Reentering write access on Windows will just deadlock, while on POSIX platforms, it will raise an exception. Also you cannot upgrade read access to write access with a TLightweightMREW lock.

32.3.5 Spinlocks

A spinlock is a lock variant that causes a thread to spin in a loop until the associated resource is unlocked, repeatedly querying if it can acquire the resource for itself. Busy waiting allows faster throughput than blocking the thread, or making it wait and waking it up when the resource becomes available. This works fairly well with a handful of threads each quickly poking the resource and getting out—especially if the hardware can run multiple queues concurrently—but the longer they use the resource, the more CPU time is wasted on looping through each thread to check the lock's status.

As with recursive locks, spinlocks are a modification of a different lock type, rather than a fully independent lock type in their own right.

Delphi implements a non-reentrant spinlock in the `TSpinLock` record.

32.3.6 Monitor

A monitor is a synchronization object that allows both locking and waiting for a certain condition, including a way to signal other threads when the condition is met. In Delphi, any object instance can be used as a monitor via the `System.TMonitor` class methods. It is a reentrant type of lock that can also be used as a lightweight alternative to critical sections, and it performs a configurable spin wait, so its performance can be tweaked if needed.

```
var
  Obj: TObject;
...
  TMonitor.Enter(Obj);
  try
  finally
    TMonitor.Exit(Obj);
  end;
```

32.4 Events

Besides locks, other commonly used synchronization objects are events, which enable signaling other threads that something has happened.

The `TEvent` class implements such an event synchronization object.

Using events is rather straightforward. The `SetEvent` method is used to signal the event—turn it on—and the `ResetEvent` method is used to clear the event—turn it off— while `WaitFor` is used to wait for a specified amount of time (or infinite) for the event to be signalled.

`WaitFor` returns a result as a `TWaitResult` enumeration:

```
  TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError, wrIOCompletion);
```

In the following example, we are simultaneously starting two threads. But after performing some task, the second thread can only continue its work after the first thread is finished. We can use an event to signal the second thread when it is safe to continue the work:

```
procedure TMainForm.EventBtnClick(Sender: TObject);
var
  Event: TEvent;
begin
  Event := TEvent.Create;

  TThread.CreateAnonymousThread(
    procedure
    begin
      OutputDebugString('THREAD1 WORKING');
      Sleep(1000);
      OutputDebugString('THREAD1 FINISHED');
      Event.SetEvent;
    end).Start;

  TThread.CreateAnonymousThread(
    procedure
    begin
      OutputDebugString('THREAD2 WORKING');
      Sleep(300);
      OutputDebugString('WAITING FOR THREAD1');
      if Event.WaitFor(INFINITE) = wrSignaled then
        OutputDebugString('THREAD2 CONTINUE WORK AFTER THREAD1 IS FINISHED')
      else
        OutputDebugString('THREAD2 UNSPECIFIED EVENT ERROR');
      // after this point it is safe to release the event object
      Event.Free;
    end).Start;
end;
```

Running the above example will produce the following output:

```
THREAD1 WORKING
THREAD2 WORKING
WAITING FOR THREAD1
THREAD1 FINISHED
THREAD2 CONTINUE WORK AFTER THREAD1 IS FINISHED
```

32.5 Atomic (interlocked) operations

If you think of locks as taking a sheet of paper and telling people that no one else is allowed to use it until you're done writing a novel on it, atomic operations are more like protecting the part of the sheet while you are writing a single word on it.

If this sounds an awful lot like locking... It *is*. But there is no additional *lock* instance involved; the locking takes place behind the scenes—often at the hardware level—lasts only as long as is needed to complete a simple operation, and, on its own, cannot cause thread deadlocks or livelocks.

The Interlocked API provides a set of functionality that enables a simple mechanism for performing simple operations on variables atomically.

Simple reads and writes on the naturally aligned, (up to) pointer-sized variables, are atomic. You will not read or write partial data when you do such an operation without any protection mechanism.

The problem arises when you need to read such a value, and then write another one related to the already-retrieved value, without possible interference from another thread.

For instance, if you want to increment the value of an integer, it is possible that one thread reads the integer value, and the other one writes some other value before the original thread has had the chance to write down the incremented value.

In other words, the following code is not thread-safe. If accessed from multiple threads, the value in `x` will not be consistently incremented:

Thread-unsafe increment

```
var
  x: Integer;
...
begin
  x := x + 1;
end;
```

Thread-safe increment

```
var
  x: Integer;
...
begin
  TInterlocked.Increment(x);
end;
```

Atomic operations are implemented as functions that return a value significant to the performed operation. For instance, in case of `Increment`, the returned value will be the actual variable value after the increment was performed, before another thread got the chance to access and change its value.

Incorrect code

```
var
  x: Integer;
  ...

procedure IncrementAndPrintX;
begin
  TInterlocked.Increment(x);
  // at this point, another thread might have changed the value in x
  Writeln(x);
end;
```

Correct code

```
var
  x: Integer;
  ...

procedure IncrementAndPrintX;
var
  Value: Integer;
begin
  Value := TInterlocked.Increment(x);
  Writeln(Value);
end;
```

In Delphi, the `TInterlocked` class currently provides the following static functions:

```
class function Increment(var Target: Integer): Integer;
class function Increment(var Target: Int64): Int64;
class function Decrement(var Target: Integer): Integer;
class function Decrement(var Target: Int64): Int64;
```

`Increment` does what its name suggests, atomically incrementing an integer, while `Decrement` atomically decrements the integer.

```
class function Add(var Target: Integer; Increment: Integer): Integer;
class function Add(var Target: Int64; Increment: Int64): Int64;
```

Add increases the value in Target by the value in the Increment parameter. Function returns the value in the Target variable after the operation is completed.

```
class function BitTestAndSet(var Target: Integer; BitOffset: TBitOffset): Boolean;
class function BitTestAndClear(var Target: Integer;
  BitOffset: TBitOffset): Boolean;
```

These two methods return whether, in Target, the bit at BitOffset is set to 1, then set its value to 1 (BitTestAndSet) or 0 (BitTestAndClear), respectively. This can be useful with bitmasks or flags, as it allows you to atomically change a field while retrieving its data at the same time. The function returns the bit's value from before the new value had been written.

```
class function Exchange(var Target: Pointer; Value: Pointer): Pointer;
class function Exchange(var Target: Integer; Value: Integer): Integer;
class function Exchange(var Target: Int64; Value: Int64): Int64;
class function Exchange(var Target: TObject; Value: TObject): TObject;
class function Exchange(var Target: Double; Value: Double): Double;
class function Exchange(var Target: Single; Value: Single): Single;
class function Exchange<T: class>(var Target: T; Value: T): T;
```

The name Exchange is only a loosely accurate description of what this method actually does. A better name for this method family would be Replace or Assign, since it atomically replaces the value in Target variable with the value in Value, and returns the old value from the Target.

While the naming is confusing, at least it is consistent with the Interlocked API used by Microsoft.

```
class function CompareExchange(var Target: Pointer; Value: Pointer;
  Comparand: Pointer): Pointer;
class function CompareExchange(var Target: Integer; Value: Integer;
  Comparand: Integer): Integer;
class function CompareExchange(var Target: Integer; Value: Integer;
  Comparand: Integer; out Succeeded: Boolean): Integer;
class function CompareExchange(var Target: Int64; Value: Int64;
  Comparand: Int64): Int64;
class function CompareExchange(var Target: TObject; Value: TObject;
  Comparand: TObject): TObject;
class function CompareExchange(var Target: Double; Value: Double;
  Comparand: Double): Double;
class function CompareExchange(var Target: Single; Value: Single;
  Comparand: Single): Single;
```

```
class function CompareExchange<T: class>(var Target: T; Value: T;
  Comparand: T): T;
```

The `CompareExchange` family suffers from the same naming issue. It does the same thing as `Exchange`, but only if `Target` and `Comparand` are equal. If they aren't, nothing happens. The returned value is the value originally passed `Target` variable, regardless of whether it had been overwritten by a new value.

```
  Result := TInterlocked.CompareExchange(Target, Value, 0);
```

In the above example, `Target` will be replaced with `Value` only if the current value of `Target` is 0. If the replacement occurred, the value of `Result` will also be 0. If the replacement didn't occur, then the value of `Result` will be some integer value other than 0. (In either case, `Result` will be equal to what `Target` used to be.)

```
class function Read(var Target: Int64): Int64;
```

There is no absolute guarantee that a naturally aligned 64-bit integer can be atomically read on all 32-bit platforms, so you can use the `Read` function's result to atomically retrieve 64-bit integers on those platforms.

Part 6. GUI and Multithreading

Chapter 33

Main (GUI) thread

Every application has (runs on) at least one thread. The thread that runs the *application event loop* is commonly called the *main thread*, *GUI thread*, or *UI thread*.

While inter-thread communication in general requires careful design in order to prevent various threading issues, the GUI thread requires special care. This is the thread most threads will try to communicate with, and OS and user interactions can cause additional problems.

The main thread is like a highway passing through a big city, with high traffic coming in, and too many side roads. Other threads are more commonly either slow country roads, or have to deal with far less busy intersections. While you can certainly crash your car anywhere, less traffic reduces the chance that something will go wrong.

There are three important rules regarding the GUI and multithreading:

- Rule number one: Never, ever touch the UI from a background thread.
- Rule number two: Never, ever touch the UI from a background thread.
- And last, but not least, rule number three: Never, ever touch the UI from a background thread.

That means **NOTHING**. No reading, no writing, no "just need to quickly peek at this one thing", no creating UI components in the background, and no destroying in the background, either.

There are many reasons why, generally, GUI frameworks across various operating systems are not thread-safe—not meant to be accessed and used from background threads.

GUIs are extremely complex frameworks on their own. Making them thread-safe would not only be hard to achieve in terms of the coding required for each and every control to make it thread-safe, but also, all the locking and synchronization would inherently make them significantly slower, thus defeating the original purpose of using threads to speed up things.

Besides being complex on their own, GUIs need to interact with the OS and hardware— both input and output—and the synchronization required to coordinate all that would inevitably cause additional slowdowns, and would be extremely prone to deadlocks.

There are classes in some parts of various UI frameworks that provide thread safety in a very limited context and can be used to speed up some more intensive operations, such as bitmap manipulation. But when using any of those, you need to be careful and use it only as intended, because anything beyond that might not work properly from the start without you even realizing it, and it can also very easily get broken with the next framework update.

An example of the complexities involved, and of how hard can it be to make a multithreaded UI framework, is the FMX framework on Android. And while not thread-safe as such, in the beginning, it was running in its own UI thread, separated from the Android UI thread. That made it run faster, but often in complex interactions with the Android UI thread, applications could deadlock. Making each FMX control thread-safe, would suffer just the same. Because of those issues, the dual-UI-thread model eventually had to be abandoned, and since Delphi 10.3 Rio, the FMX UI thread is unified with the Android UI thread.

So whether you like it or not, you will just have to keep background threads away from your UI if you want to have a well-behaved application.

Chapter 34

Communicating with the main thread

Long-running tasks in background threads often need to communicate with the main GUI thread, if for nothing else than to show their progress to the user. There are several ways to implement such communication, but the most common approaches are using messaging systems—either OS-provided ones or from custom frameworks—or to use thread synchronization to execute blocks of GUI-related code in the context of the main thread.

No matter which communication approach you use and what kind of information you need to communicate and when during running background tasks, you will face two main problems: avoiding deadlocks, and using references to controls and data that are already destroyed. The following chapters will mainly focus on various communication approaches and implementations, and the common problems that can arise when using any of them will be covered separately in the *Communication and GUI issues* chapter.

Using messaging systems will generally be a better and faster option, while synchronization is slower and is more susceptible to the previously mentioned issues, but is also usually simpler to implement. If sending updates to the UI does not happen very frequently, then the synchronization approach might be a good way to start, especially for beginners working on very simple applications.

If the background task is properly implemented, and does not suffer from other threading issues, switching to messaging systems from the synchronization approach should not pose a significant problem. If you opt into the synchronization approach, keep the code running in the context of the main thread as small and fast as possible.

34.1 Synchronization with the main thread

34.1.1 Updating UI after the task is completed

This is the simplest scenario. You can simply call `Synchronize` or `Queue` at the end of the task, or use the thread's `OnTerminate` event handler.

Using `Queue` is generally a better choice, because it is not a blocking call, and the thread will be able to complete the task faster. Also, when using reusable threads through tasks in the PPL library or any other similar framework, and when synchronization needs to be done upon task completion, `Queue` will release the thread for reuse sooner. On the other hand, if you need to perform some task cleanup that would complicate your logic if `Queue` is used, then using `Synchronize` might be a better option.

If you will at any point wait for the thread or task in the context of the main thread, then you must not use `Synchronize`, because the application will deadlock. You also need to be careful with references used in `Queue` calls if you do wait, because they may be gone by the time the called code is executed. That includes references to the GUI controls you are updating.

If you don't wait for the thread or task to finish, then you need to pay attention to references used in both `Synchronize` and `Queue` calls, because they can be long gone at that point.

Choosing between `Synchronize` and `Queue` should be made on a case-by-case basis. Sometimes one will be better than the other, sometimes both options will be equally bad (or good).

All of the above is also valid for all other communication scenarios and approaches.

34.1.2 Updating progress counter or description

Showing task progress is not much more complicated than updating the UI when the task is completed. The following example focuses only on the code required for showing progress, not solving reentrancy issues, deadlocks, or vanishing GUI controls. Applied as-is, it will most certainly suffer if the user closes the form and it gets destroyed while the thread is running, because the `ProgressBar` and `ProgressLabel` controls will also get destroyed.

At this point, it may seem like the synchronization approach is a total dead end, but there are some ways to work around those issues, as shown in the *Cleanup on GUI destruction* chapter.

```
TTaskForm = class(TForm)
  ProgressBar: TProgressBar;
  ProgressLabel: TLabel;
  ...
private
  procedure StartProgress;
  procedure StopProgress;
  procedure Progress(APercent: Integer; const ADescription: string);
end;
```

```
procedure TTaskForm.StartProgress;
begin
  ProgressBar.Position := 0;
  ProgressBar.Visible := True;
  ProgressLabel.Caption := '';
  ProgressLabel.Visible := True;
end;

procedure TTaskForm.StopProgress;
begin
  ProgressBar.Visible := False;
  ProgressLabel.Visible := False;
end;

procedure TTaskForm.Progress(APercent: Integer; const ADescription: string);
begin
  TThread.Queue(nil,
    procedure
    begin
      ProgressBar.Position := APercent;
      ProgressLabel.Caption := ADescription;
    end);
end;

procedure TTaskForm.TaskBtnClick(Sender: TObject);
begin
  StartProgress;
  TTask.Run(
    procedure
    var
      I: Integer;
    begin
      for I := 1 to 100 do
        begin
          Sleep(100);
          Progress(I, 'Step ' + I.ToString);
        end;
      TThread.Queue(nil,
        procedure
        begin
          StopProgress;
        end);
    end);
end;
```

34.1.3 Iteratively creating and populating GUI controls while task is running

In general, there is very little difference between writing code that updates progress, and code that iteratively creates and populates GUI controls. Both things will need to call either `Synchronize` or `Queue` around the code that handles the GUI. The main difference comes from the time needed to run a synchronized piece of code. Updating progress usually requires a very small amout of time to run, whereas creating and populating controls can require a lot more time, especially for complex ones. The number of created controls or synchronization calls can also have a significant impact on performance. There is a huge difference between creating a few controls, and creating a thousand of the same.

```
  TTaskForm = class(TForm)
    ...
    Memo: TMemo;
    ListView: TListView;
    ProgressBar: TProgressBar;
    ProgressLabel: TLabel;
  private
    procedure GUIBeginUpdate;
    procedure GUIEndUpdate;
    procedure GUIAddItem(const AData: string; APercent: Integer;
      const ADescription: string);
  end;

procedure TTaskForm.GUIBeginUpdate;
begin
  Memo.Lines.BeginUpdate;
  ListView.Items.BeginUpdate;
  ProgressBar.Position := 0;
  ProgressBar.Visible := True;
  ProgressLabel.Caption := '';
  ProgressLabel.Visible := True;
end;

procedure TTaskForm.GUIEndUpdate;
begin
  Memo.Lines.EndUpdate;
  ListView.Items.EndUpdate;
  ProgressBar.Visible := False;
  ProgressLabel.Visible := False;
  FTask := nil;
end;
```

```
procedure TTaskForm.GUIAddItem(const AData: string; APercent: Integer;
  const ADescription: string);
begin
  TThread.Queue(nil,
    procedure
    var
      Item: TListItem;
    begin
      Memo.Lines.Add(AData);
      Item := ListView.Items.Add;
      Item.Caption := AData;
      ProgressBar.Position := APercent;
      ProgressLabel.Caption := ADescription;
    end);
end;

procedure TTaskForm.GUICreationBtnClick(Sender: TObject);
begin
  GUIBeginUpdate;
  try
    TTask.Run(
      procedure
      var I, Count: Integer;
      begin
        try
          Count := 100;
          for I := 1 to Count do
            begin
              Sleep(100);
              GUIAddItem('Item ' + I.ToString, Round((I / Count) * 100),
                'Step ' + I.ToString);
            end;
        finally
          TThread.Queue(nil,
            procedure
            begin
              GUIEndUpdate;
            end);
        end;
      end);
  except
    GUIEndUpdate;
  end;
end;
```

34.2 Speeding up GUI controls

The BeginUpdate and EndUpdate methods called around the GUI control calls speed up adding data, because the control will be repainted once all the work is done, not for every added item. If you have to add only a few items, you might like the effect of iteratively painting new data, but this is really not a viable approach if you have many items.

The previous example populates rather simple controls with very simple data, but even with those, you can see a visible difference in speed if you comment out the Sleep call that simulates some processing work, and try changing the Count value. Adding 100 items will run fast, adding 1000 is visibly slower but is still manageable... but try adding 10000.

In the case of TMemo and similar controls, where data is contained in a simple container class like TStrings, a faster approach would be to create a local TStrings instance, add all data to that local instance in the context of the background thread, and then just assign that instance to the TMemo control when all data is processed. In such cases, you don't even need to call the BeginUpdate and EndUpdate methods:

```
TTask.Run(
  procedure
  var
    I: Integer;
    Count: Integer;
    Items: TStringList;
  begin
    Items := TStringList.Create;
    try
      Count := 10000;
      for I := 1 to Count do
        Items.Add('Item ' + I.ToString);
    finally
      TThread.Queue(nil,
        procedure
        begin
          Memo.Lines := Items;
          Items.Free;
        end);
    end;
  end);
```

But there are a few tiny issues with the above approach. Namely, TStrings is an extremely inefficient class, and assigning one list to another literally iterates through the original list, adding the strings one by one. The only speed gain is in not calling TThread.Queue a zillion times.

If you want to show your progress, then you will be calling TThread.Queue anyway. However, even if you do need to show progress, there is a difference between calling TThread.Queue 100 times and 10000 times. So if you change your progress logic to remember the old progress value and do a progress update only when that percentage changes, you can also save some time.

The main performance issue—populating the control—still remains. The solution is using controls that offer more separation between data and the control itself, or at least controls that have better algorithms and options when replacing bulk data.

Luckily for us, TStrings is an abstract class, and controls use more appropriate descendant variants. TMemo uses its own TMemoStrings class, and while assigning TStrings uses a dumb algorithm, assigning its Text property is another story:

```
  TTask.Run(
    procedure
    var
      I: Integer;
      Count: Integer;
      Items: TStringList;
      ItemsStr: string;
    begin
      Items := TStringList.Create;
      try
        Count := 10000;
        for I := 1 to Count do
          Items.Add('Item ' + I.ToString);
        ItemsStr := Items.Text;
      finally
        Items.Free;
        TThread.Queue(nil,
          procedure
          begin
            Memo.Lines.Text := ItemsStr;
          end);
      end;
    end);
```

The above code could avoid the additional string variable, and directly write

```
Memo.Lines.Text := Items.Text;
```

but the constructing content of the Text property takes time, and using an intermediate string variable ensures that this work is done in the context of the background thread instead of the context of the main thread. Whenever it is possible to perform some work in the background, it is prudent to do so.

But the main speed gain here is in the fact that TMemo is a thin wrapper around an OS control that allows setting complete text in one go, without having to fiddle with separate lines.

Not all controls do that kind of processing. `TComboBox`, for instance, does additional parsing of the `Text` property, and then still adds items one by one. Assigning `TStrings` directly is a better option there. In order to have performant code, sometimes you need to know more details about the inner workings of the controls and classes you use.

An even faster approach would be having a control where the data container is not owned by the control, and could be easily swapped. Imagine `TComboBox` controls where assigning `Items` property would not create a deep copy - and would just replace one reference to data with another, followed by repainting the control. Such controls would have slightly more complicated memory management of data containers, but that problem can be easily solved by using reference-counted classes as data containers.

If populating complex GUI controls is the real bottleneck for the performance, then the only real solution is using controls where the visual representation is separated from the actual data.

For scrollable controls that can contain large amounts of data, and where creating an actual control for each item would take too much time, the solution is using views with recyclable items, that don't require creating visual representation controls for each item in the list, but create controls for only the few items that can be seen on the screen at the same time. When the user scrolls through the list, existing controls will just be repopulated (recycled) with new data.

34.3 Using messaging systems

Using messaging systems is generally the best option for any thread-to-GUI communication. The main problem is that Delphi does not provide such a system out of the box. The messaging system in the `System.Messaging` unit is not thread-safe, and cannot be used for messaging beyond thread boundaries.

On Windows, you can always use the Windows messaging system, but sending any data that does not fit into two integer parameters means you will need to send pointers, and get yourself involved in a form of manual memory management that can easily turn into a temporal warzone.

If you need a cross-platform solution, then obviously the Windows messaging system is the wrong approach altogether. Be careful if the messaging system implementation has synchronization code calling `TThread.Synchronize` or `TThread.Queue`, because it can cause unintentional deadlocks or lifetime issues.

34.3.1 Using System.Messaging

We already established that `System.Messaging` is not a thread-safe solution, but if you don't have any better solution at hand, you can still use `System.Messaging` if you call the library's code from the context of the main thread. This approach can work with any other messaging library that is not thread-safe.

Declaring message classes and subscription variables, or subscribing and unsubscribing to and from messages, is very much like in non-threaded communication:

```
type
  TProgressStartMessage = class(TMessage);
  TProgressStopMessage = class(TMessage);
  TProgressMessage = class(TMessage)
  public
    Percent: Integer;
    Description: string;
    constructor Create(APercent: Integer; const ADescription: string);
  end;

  TTaskForm = class(TForm)
    ...
    ProgressBar: TProgressBar;
    ProgressLabel: TLabel;
  private
    FTask: ITask;
    FProgressStartID: Integer;
    FProgressStopID: Integer;
    FProgressID: Integer;
    procedure SendStartProgress;
    procedure SendStopProgress;
    procedure SendProgress(APercent: Integer; const ADescription: string);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  end;

constructor TProgressMessage.Create(APercent: Integer;
  const ADescription: string);
begin
  inherited Create;
  Percent := APercent;
  Description := ADescription;
end;

constructor TTaskForm.Create(AOwner: TComponent);
begin
  inherited;
  FProgressStartID := TMessageManager.DefaultManager.SubscribeToMessage(
    TProgressStartMessage,
    procedure(const Sender: TObject; const M: TMessage)
```

```
      begin
        ProgressBar.Position := 0;
        ProgressBar.Visible := True;
        ProgressLabel.Caption := '';
        ProgressLabel.Visible := True;
      end);

    FProgressStopID := TMessageManager.DefaultManager.SubscribeToMessage(
      TProgressStopMessage,
      procedure(const Sender: TObject; const M: TMessage)
      begin
        ProgressBar.Visible := False;
        ProgressLabel.Visible := False;
      end);

    FProgressID := TMessageManager.DefaultManager.SubscribeToMessage(
      TProgressMessage,
      procedure(const Sender: TObject; const M: TMessage)
      begin
        with TProgressMessage(M) do
          begin
            ProgressBar.Position := Percent;
            ProgressLabel.Caption := Description;
          end;
      end);
  end;

  destructor TTaskForm.Destroy;
  begin
    TMessageManager.DefaultManager
      .Unsubscribe(TProgressStartMessage, FProgressStartID);
    TMessageManager.DefaultManager
      .Unsubscribe(TProgressStopMessage, FProgressStopID);
    TMessageManager.DefaultManager
      .Unsubscribe(TProgressMessage, FProgressID);
    if Assigned(FTask) then
      FTask.Wait(INFINITE);
    inherited;
  end;

  procedure TTaskForm.SendStartProgress;
  begin
    TMessageManager.DefaultManager.SendMessage(nil, TProgressStartMessage.Create);
  end;
```

```
procedure TTaskForm.SendStopProgress;
begin
  TMessageManager.DefaultManager.SendMessage(nil, TProgressStopMessage.Create);
  FTask := nil;
end;

procedure TTaskForm.SendProgress(APercent: Integer; const ADescription: string);
begin
  TThread.Queue(nil,
    procedure
    begin
      TMessageManager.DefaultManager.SendMessage(nil,
        TProgressMessage.Create(APercent, ADescription));
    end);
end;
```

The main difference (besides waiting for the task in the destructor) is in the code that sends progress messages. In singlethreaded code that uses `Application.ProcessMessages` to show GUI updates, our task code might look like the following:

```
procedure TTaskForm.TaskBtnClick(Sender: TObject);
var
  I: Integer;
begin
  SendStartProgress;
  Application.ProcessMessages;
  for I := 1 to 100 do
    begin
      Sleep(100);
      TMessageManager.DefaultManager.SendMessage(nil,
        TProgressMessage.Create(I, 'Step ' + I.ToString));
      Application.ProcessMessages;
    end;
  SendStopProgress;
end;
```

But if we add threading to the above code and want to use a thread-unsafe messaging library, we need to run the messaging code in context of the main thread:

```
procedure TTaskForm.TaskBtnClick(Sender: TObject);
begin
  // prevent reentrancy
  if Assigned(FTask) then
    Exit;

  SendStartProgress;
  FTask := TTask.Run(
    procedure
    var
      I: Integer;
    begin
      for I := 1 to 100 do
        begin
          Sleep(100);
          SendProgress(I, 'Step ' + I.ToString); // SendProgress calls Queue
        end;
      TThread.Queue(nil,
        procedure
        begin
          SendStopProgress;
        end);
    end);
end;
```

This kind of code still has a problem if waiting for the task will last too long, but unsubscribing from receiving messages at least guarantees that any destroyed GUI controls will not be used during that time. The queued code will run and send messages, but no one will receive them. If there is some code that absolutely must run, then it should be called outside the `Queue` method, provided that it does not need to be executed in the context of the main thread.

If you need to run some finalization code in the context of the main thread, you can find solutions for such scenarios in the *Cleanup on GUI destruction* chapter.

Using a messaging system to send data updates and create or populate GUI controls is not too different from using the synchronization approach. The speed issues with populating controls are the same; the difference is that you need to declare and register an additional message type that will pass on the appropriate data.

34.3.2 Using a thread-safe messaging system

If you use a thread-safe messaging system, then you can safely omit calls to `TThread.Queue`, because adding messages to the messaging queue is a thread-safe operation. Since sending messages is not done from the context of the main thread in this case, you will need to make sure that any code that runs when the message is received runs in the context of the main thread.

In the following example, the `System.Messaging` API is used for simplicity, and we are pretending that `TThreadSafeMessageManager` is a thread-safe implementation of that API.

Besides being thread-safe, it is of the utmost importance that such a messaging system supports posting messages in non-blocking calls. If you use a blocking `SendMessage` call, you will have trouble with the `Synchronize` call when the message is received, and if you use `Queue` instead of `Synchronize` at that point, then you have opened the possibility of accessing the GUI controls after the GUI has been destroyed, or even worse, the message object itself may be destroyed at that point, depending on how the message lifetime is handled in that particular messaging system. If the messaging system is thread-safe, but does not have the ability to post messages, then you will just have to use the same code logic used for the thread-unsafe `System.Messaging`.

```
...
  FProgressID := TThreadSafeMessageManager.DefaultManager.SubscribeToMessage(
    TProgressMessage,
    procedure(const Sender: TObject; const M: TMessage)
    begin
      TThread.Synchronize(nil,
        procedure
        begin
          with TProgressMessage(M) do
            begin
              ProgressBar.Position := Percent;
              ProgressLabel.Caption := Description;
          end;
        end);
    end);
...

procedure TTaskForm.PostStartProgress;
begin
  TThreadSafeMessageManager.DefaultManager.PostMessage(nil,
    TProgressStartMessage.Create);
end;

procedure TTaskForm.PostStopProgress;
begin
  TThreadSafeMessageManager.DefaultManager.PostMessage(nil,
    TProgressStopMessage.Create);
  FTask := nil;
end;

procedure TTaskForm.PostProgress(APercent: Integer; const ADescription: string);
```

```
begin
  TThreadSafeMessageManager.DefaultManager.PostMessage(nil,
    TProgressMessage.Create(APercent, ADescription));
end;

procedure TTaskForm.TaskBtnClick(Sender: TObject);
begin
  // prevent reentrancy
  if Assigned(FTask) then
    Exit;

  PostStartProgress;
  FTask := TTask.Run(
    procedure
    var
      I: Integer;
    begin
      for I := 1 to 100 do
        begin
          Sleep(100);
          PostProgress(I, 'Step ' + I.ToString);
        end;
      PostStopProgress;
    end);
end;
```

Chapter 35

Communication and GUI issues

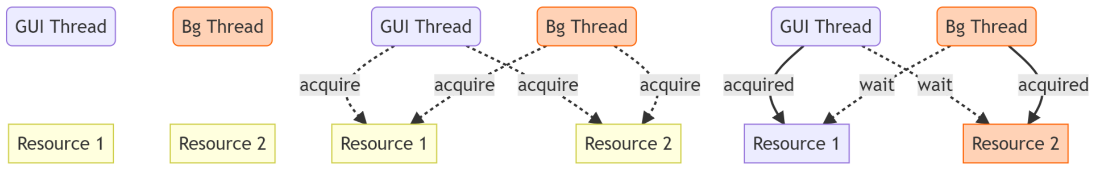
35.1 Deadlocking the main thread

Deadlocking threads is one of the perils of multithreading, and it is the one that can most easily appear when the main thread is involved.

It is not because the main thread is somehow more susceptible to deadlocking than other threads, it is just that most of the multithreading code in the application (especially client apps) will involve synchronization and accessing shared data in the context of the main thread. The same issues can easily arise if two secondary threads use such code.

35.1.1 Deadlock with two shared resources

In the following image, we can see the timeline of the deadlock. We start with two threads: a GUI thread and a background thread (this kind of scenario can happen with any two threads), that both need to acquire two resources for their work. They both issue acquire requests for both resources. If any of the threads can successfully acquire both resources, there is no deadlock, and the other thread will just have to wait a bit for the first thread to finish. However, if one thread manages to acquire one resource, and the other thread manages to acquire the second resource, both threads will not be able to finish their work, and will be locked waiting for the needed resource forever:



The following deadlock example is similar to the one in the *Synchronization primitives* chapter, but it involves a GUI thread and a background thread instead of two background threads:

```
procedure TMainForm.DeadlockBtnClick(Sender: TObject);
var
  Lock1: TCriticalSection;
  Lock2: TCriticalSection;
begin
  Lock1 := TCriticalSection.Create;
  Lock2 := TCriticalSection.Create;

  TThread.CreateAnonymousThread(
    procedure
    begin
      Lock2.Enter;
      try
        OutputDebugString('THREAD2 LOCK2 ACQUIRED');
        Sleep(500);
        Lock1.Enter;
        try
          OutputDebugString('THREAD2 LOCK1 ACQUIRED');
          OutputDebugString('THREAD2 WORKING');
        finally
          Lock1.Leave;
          OutputDebugString('THREAD2 LOCK1 RELEASED');
        end;
      finally
        Lock2.Leave;
        OutputDebugString('THREAD2 LOCK2 RELEASED');
      end;
    end).Start;

  Lock1.Enter;
  try
    OutputDebugString('GUI THREAD LOCK1 ACQUIRED');
    Sleep(1000);
    Lock2.Enter;
    try
      OutputDebugString('GUI THREAD LOCK2 ACQUIRED');
      OutputDebugString('GUI THREAD WORKING');
    finally
      Lock2.Leave;
      OutputDebugString('GUI THREAD LOCK2 RELEASED');
    end;
  finally
    Lock1.Leave;
    OutputDebugString('GUI THREAD LOCK1 RELEASED');
  end;
  // this example leaks both locks, but this is not relevant
  // for demonstrating the deadlock problem
end;
```

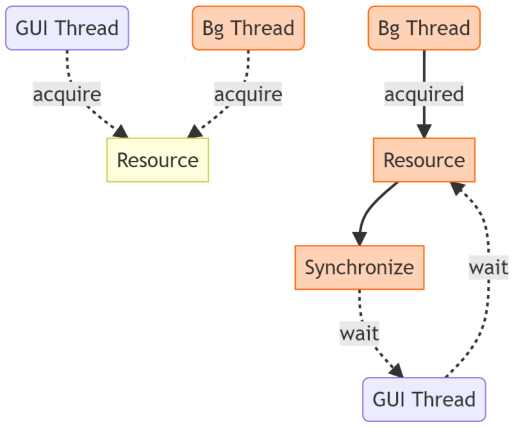
Running the above code produces the following output, showing that none of the threads were capable of completing their task:

```
GUI THREAD LOCK1 ACQUIRED
THREAD2 LOCK2 ACQUIRED
```

35.1.2 Deadlock with shared resource and Synchronize

To deadlock a GUI thread, you don't even need two resources in play. All you need is a single shared resource, and a background thread trying to do some work in the context of the GUI thread.

Both a GUI and a background thread try to acquire a shared resource. If the GUI thread acquires it first, then all is good, and the background thread will just have to wait a bit. But if the background thread acquires the resource first, and then calls `Synchronize` while the GUI thread is already waiting for the shared resource, then both threads will be locked, waiting forever:



```
procedure TMainForm.SyncLockBtnClick(Sender: TObject);
var
  Lock: TCriticalSection;
begin
  Lock := TCriticalSection.Create;
  TThread.CreateAnonymousThread(
    procedure
    begin
      Lock.Enter;
      OutputDebugString('THREAD LOCK ACQUIRED');
      try
        Sleep(1000);
        TThread.Synchronize(nil,
          procedure
          begin
            OutputDebugString('SYNCHRONIZED');
          end);
      finally
        Lock.Leave;
        OutputDebugString('THREAD LOCK RELEASED');
      end;
    end).Start;

  Sleep(200);

  Lock.Enter;
  try
    OutputDebugString('GUI THREAD LOCK ACQUIRED');
  finally
    Lock.Leave;
    OutputDebugString('GUI THREAD LOCK RELEASED');
  end;
  // this example leaks the lock, but this is not relevant
  // for demonstrating the deadlock problem
end;
```

The above code will cause a synchronization deadlock and the following output, showing that none of the threads were able to complete their work:

```
THREAD LOCK ACQUIRED
```

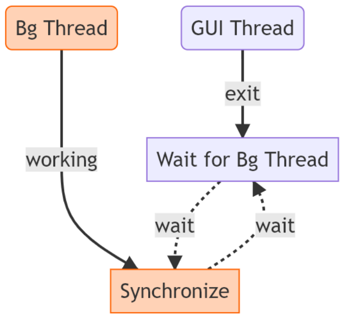
One of the workarounds for a synchronization deadlock is calling the non-blocking `Queue` method instead of the blocking `Synchronize`. That way, the background thread will continue its work, and will eventually be able to finish and release the acquired resource. However, in such a scenario, all queued code will not run before the background thread is done and the GUI thread has finished what it is currently working on.

However, this workaround is often not a proper solution for the problem, but can merely be used as a temporary hack to prevent application deadlock. While the GUI thread is waiting for the background thread to finish its job, it will be completely blocked and unable to process OS messages—which is exactly the situation the background thread was supposed to solve in the first place.

A synchronization deadlock in combination with a bad workaround exposes deeper problems. Any situation where a background thread can lock a shared resource needed by the GUI thread can result in an unresponsive GUI. Only in situations where locks will be held for an extremely short time, and when there is no danger of a synchronization deadlock, can you safely share (locked) resources between GUI and background threads.

35.1.3 Deadlock with waiting for a thread and Synchronize

One of the common problems when communicating with the GUI from a background thread arises when the GUI needs to wait for a thread or task to finish. If the task runs to completion before the user closes the form (or another part of the GUI), triggering its destruction, such code will work well. However, if the task is still running, attempts to communicate with non-existent GUI controls or using other related data will cause crashes. One of the solutions is to wait for background task completion. However, that waiting will be done in the context of the main thread, and if the background thread calls `Synchronize` at any point while the main thread is waiting, that will cause a deadlock:



```
destructor TTaskForm.Destroy;
begin
  if Assigned(FTask) then
    FTask.Wait(INFINITE);
  inherited;
end;
```

```
procedure TTaskForm.TaskBtnClick(Sender: TObject);
begin
  if Assigned(FTask) then
    Exit;

  FTask := TTask.Run(
    procedure
    begin
      OutputDebugString('TASK RUNNING');
      Sleep(10000);
      TThread.Synchronize(nil,
        procedure
        begin
          OutputDebugString('TASK FINISHED');
          FTask := nil;
        end);
    end);
end;
```

If we close the form—triggering waiting for the task running in the background thread— before the task is finished, the application will deadlock, showing that the task was never completed:

```
TASK RUNNING
```

A workaround for the above kind of deadlock is using `Queue` instead of `Synchronize`, but just like using `Queue` was a poor workaround in the previous example with `Synchronize` and shared resources, in this particular situation it is even worse. Again, waiting for a thread or task will cause an unresponsive GUI, but that is the least of your problems. The queued code will run after the relevant GUI elements have already been destroyed, and any attempt to use any of the destroyed GUI controls or data will crash.

If you are using custom threads and create/destroy a thread inside a GUI control, you can pass the thread variable to the `Queue` method to clean up queued code when the thread is destroyed. But, that approach is only viable if you use non-reusable threads, and you wait for the thread upon GUI destruction. If you use `TTask` or any similar framework, cleaning the queued code will not work.

Using a messaging system that will just skip sending queued messages after the recipient is gone is a step in the right direction, but it will not solve potential GUI unresponsiveness if waiting for the task takes too long. However, if you are communicating with messages, you most likely don't need to wait for tasks at all.

35.2 Cleanup on GUI destruction

There is one particularly hard problem that's rather specific to Delphi, because of its lack of automatic memory management for GUI controls: doing proper cleanup when background tasks are running and the user closes the related part of the GUI and triggers GUI destruction. There are several aspects of this problem, but the main issue arises from the fact that the GUI controls your thread might use for showing results during synchronization might have been destroyed by that point.

35.2.1 Don't destroy the GUI

The easiest way to avoid issues with accessing destroyed GUI controls is preventing GUI destruction while the thread or task is running. You can also show some message to the user to let them know that the task is in progress:

```
procedure TTaskForm.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  CanClose := not Assigned(FTask);
end;
```

35.2.2 Use guardian interface to prevent access to destroyed GUI

Tasks that run inside anonymous methods can use their variable capturing mechanism to capture some guardian interface reference, which will know whether the GUI is still available or not. It is important to only use such a guardian variable in code running in the context of the main thread, otherwise its Boolean flag might get invalidated in the middle of some operation.

A guardian interface can be used to protect GUI access when a messaging system is not used for thread-to-GUI communication. Since the guardian is implemented as a reference-counted class, its memory will be automatically released when its last strong reference goes out of scope:

```
type
  IGuardian = interface
    function GetIsDismantled: Boolean;
    procedure Dismantle;
    property IsDismantled: Boolean read GetIsDismantled;
  end;

  TGuardian = class(TInterfacedObject, IGuardian)
  private
    FIsDismantled: Boolean;
    function GetIsDismantled: Boolean;
  public
    procedure Dismantle;
    property IsDismantled: Boolean read GetIsDismantled;
  end;

  TTaskForm = class(TForm)
    ...
  private
    FGuardian: IGuardian;
    ...
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  end;

function TGuardian.GetIsDismantled: Boolean;
begin
  Result := FIsDismantled;
end;

procedure TGuardian.Dismantle;
begin
  FIsDismantled := True;
end;

constructor TTaskForm.Create(AOwner: TComponent);
begin
  inherited;
  FGuardian := TGuardian.Create;
  ...
end;

destructor TTaskForm.Destroy;
begin
  FGuardian.Dismantle;
  ...
  inherited;
end;

procedure TTaskForm.GUIAddItem(const AGuardian: IGuardian; const AData: string;
  APercent: Integer; const ADescription: string);
begin
  TThread.Queue(nil,
    procedure
    var
```

```
        Item: TListItem;
      begin
        if AGuardian.IsDismantled then
          Exit;

        Memo.Lines.Add(AData);
        Item := ListView.Items.Add;
        Item.Caption := AData;
        ProgressBar.Position := APercent;
        ProgressLabel.Caption := ADescription;
      end);
  end;

  procedure TTaskForm.GUICreationBtnClick(Sender: TObject);
  var
    LGuardian: IGuardian;
  begin
    if Assigned(FTask) then
      Exit;

    LGuardian := FGuardian;
    GUIBeginUpdate;
    try
      FTask := TTask.Run(
        procedure
        var
          I: Integer;
          Count: Integer;
        begin
          try
            Count := 100;
            for I := 1 to Count do
              begin
                Sleep(100);
                GUIAddItem(LGuardian, 'Item ' + I.ToString,
                  Round((I / Count) * 100), 'Step ' + I.ToString);
              end;
          finally
            TThread.Queue(nil,
              procedure
              begin
                if not LGuardian.IsDismantled then
                  GUIEndUpdate;
              end);
          end;
        end);
    except
      GUIEndUpdate;
    end;
  end;
```

A guardian interface is extremely simple, as is its implementation. It only needs to hold a single `Boolean` flag that will tell us whether *protected* resources—GUI or any other data—have been dismantled or not. The guardian field should be initialized in the constructor, before any other code would attempt to use it, and the `Dismantle` method should be called as soon as we know the GUI will be destroyed. This can be in the destructor, but also in any other method that is called before the destruction process begins, like the `OnCloseQuery`, `OnClose`, and `OnDestroy` form event handlers, or the `BeforeDestruction` method.

It is safe to call the `Dismantle` method multiple times if you have a guardian field implemented in some base GUI class, and the descendant classes need to protect access to the GUI sooner than the `Dismantle` method is to be called in the ancestor class.

Anonymous method variable capture will capture our guardian instance and extend its lifetime beyond the lifetime of the associated GUI controls, and it will be valid for the whole lifetime of the anonymous method that captures it. However, we must assign the guardian interface to the local variable and capture the location of that variable, not the field itself. If we used the field directly, the captured location in that case would be the field, and it would be cleared when the GUI control is destroyed. At that point, our captured guardian reference would contain `nil`:

Incorrect code

```
procedure TTaskForm.GUICreationBtnClick(Sender: TObject);
begin
   ...
          TThread.Queue(nil,
            procedure
            begin
              if not FGuardian.IsDismantled then
                GUIEndUpdate;
            end);
   ...
end;
```

35.2.3 Using messaging systems

Using messaging systems is often the best approach to prevent accessing GUI controls after they have been destroyed. Make sure that you unsubscribe from receiving messages before any part of the GUI or the other used data is no longer valid. It is also important to have a messaging system where, after unsubscribing, all already-queued messages for that recipient will be discarded.

35.2.4 Waiting for a thread

Waiting for a thread blocks the GUI, so it should be avoided for all but the most simple scenarios, where the completion of a particular thread or task will only take half a second or preferably less, even in a worst-case scenario. If you are using any of the previous mentioned approaches: not destroying the GUI while the task is running, or using a guardian interface or messaging systems, then you don't need to wait for a thread or task at all when closing secondary forms or parts of GUI elements.

Be careful with calling `Free` on GUI-owned threads, because the thread destructor will call the `WaitFor` method as part of the cleanup process.

For any task and thread that needs more than a half-second to complete (in its worst-case scenario), you should implement task cancellation or checking the thread's `Terminated` flag. In such cases, you should call the task's `Cancel` method or the thread's `Terminate` method upon GUI destruction. Also, such threads should not be owned by the GUI, because waiting for a thread in the GUI destructor can prevent smooth shutdown of the GUI elements.

The following example shows the complete lifecycle of the task, that prevents reentrancy, and implements cancellation:

```
  TTaskForm = class(TForm)
  ...
  private
    FTask: ITask;
  public
    destructor Destroy; override;
  end;

destructor TTaskForm.Destroy;
begin
  if Assigned(FTask) then
    FTask.Cancel;
  inherited;
end;

procedure TMainForm.TaskCancelBtnClick(Sender: TObject);
```

```
begin
  if Assigned(FTask) then
    begin
      FTask.Cancel;
      FTask := nil;
    end;
end;

procedure TTaskForm.TaskBtnClick(Sender: TObject);
var
  LTask: ITask;
begin
  // prevent reentrancy
  if Assigned(FTask) then
    Exit;

  LTask := TTask.Create(
    procedure
    begin
      try
        ...
        LTask.CheckCanceled;
      finally
        TThread.Queue(nil,
          procedure
          begin
            LTask := nil;
            FTask := nil;
          end);
      end;
    end);
  FTask := LTask;
  FTask.Start;
end;
```

While waiting for threads and tasks is not the best approach while closing secondary forms and parts of the GUI, when the whole application shuts down, any running threads will need to be waited for, otherwise bad things can happen. Even using a messaging system will not help in such cases, because by the time the thread finishes, the messaging system might already have been dismantled. If you are using tasks from the PPL library, then proper cleanup will be done as part of the PPL library's shutdown process.

If you have to handle many custom threads, it would be prudent to have them stored in some kind of repository—like a non-reusable pool—to be able to perform proper cleanup upon application shutdown. Keep in mind that the user can decide to close the application at any moment, and secondary forms and other GUI logic have to anticipate such events.

If you think I am giving contradictory suggestions—first don't wait for threads, then wait for threads—you are right. Whether you will be waiting or not depends on the context, and you have to be prepared to handle both scenarios. This is where using the PPL library has an advantage over using standalone threads through custom `TThread` extensions. With the PPL, you can avoid waiting for a task, and still have proper cleanup without having to worry too much about it.

35.3 Final notes

Avoiding issues in multithreading can be extremely hard. All the little pieces of code and examples usually represent only a small piece of the puzzle. When you are building the whole application, you have to make sure that all the pieces fit well together, because even small modifications in the logic can break things.

For instance, if you don't have task cancellation, you can safely use `TTask.Run`. The moment you implement cancellation, you will have to take into the account that `TTask.Run` will automatically start the thread, and that your task variable may not be initialized at the time you are checking for cancellation.

Many examples have been written without fully handling task exceptions, but in your code, you will want to have all that properly handled. For some small, short task, you may be in position to simply ignore exceptions raised in the task, as they will not be automatically propagated to the application. Then you realize that you need to wait for that task upon GUI destruction (it is a short task, so waiting is appropriate), and all of a sudden, that simple `Wait` propagates that silent exception to the main thread's level.

Every single line has the potential to have an undesireable impact on previously working code logic, and this is what ultimately makes asynchronous programming and multithreading hard to do right.

It may be overwhelming if you are just starting to learn it, but eventually, you will pick up common patterns, and you will not have to contemplate about every single line of code. And then... just when you start feeling comfortable, you will make some rookie mistake. Don't feel too bad about it—we all do that, no matter how experienced we are.

Again, pay special attention to:

- properly protecting shared data on every access point; locks only work if all code accessing the protected resource uses that lock
- locking multiple resources
- locking in general - even without deadlocks, sharing resources with the main thread can cause an unresponsive GUI
- calls to `Synchronize` in combination with waiting on a thread, task, or locking
- calls to `Synchronize` in general

- the memory management aspects of calls to `Queue`—make sure that all used references will still be valid when the queued code runs
- general lifetime of the used references in background threads; `Queue` is the most error-prone, but `Synchronize` is far from being safe either
- avoid waiting for long lasting threads or tasks in secondary forms—even when it will not cause a deadlock, it can make your application unresponsive

And last but not least, don't be afraid to abandon bad code and bad designs. Hanging onto your mistakes just because you have spent a lot of time making them will not magically fix them. Learn from your mistake and start over, it will take less time than the neverending process of fixing bad code that really cannot ever be properly fixed.

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

- Delphi Exceptions

http://docwiki.embarcadero.com/RADStudio/en/Exceptions_(Delphi)

- Prevent Uncaught Hardware Exceptions

http://docwiki.embarcadero.com/RADStudio/en/Migrating_Delphi_Code_to_Mobile_from_Desktop#Use_a_Function_Call_in_a_try-except_Block_to_Prevent_Uncaught_Hardware_Exceptions

- What do the letters W and L stand for in WPARAM and LPARAM?

https://devblogs.microsoft.com/oldnewthing/20031125-00/?p=41713

- Windows Data Types

https://docs.microsoft.com/en-us/windows/win32/winprog/windows-data-types

- Windows Messages and Message Queues

https://docs.microsoft.com/en-us/windows/win32/winmsg/messages-and-message-queues

- Windows System-Defined Messages

https://docs.microsoft.com/en-us/windows/win32/winmsg/about-messages-and-message-queues#message-types

- TerminateThread function

https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminatethread

- Catch Me If You Can

https://dalijap.blogspot.com/2018/10/catch-me-if-you-can.html

- TApplication.ProcessMessages

http://docwiki.embarcadero.com/Libraries/en/FMX.Forms.TApplication.ProcessMessages

- Delphi type sizes and alignment

http://docwiki.embarcadero.com/RADStudio/en/Internal_Data_Formats_(Delphi)

- Align fields directive

http://docwiki.embarcadero.com/RADStudio/en/Align_fields_(Delphi)

- 64-bit Windows Data Types Compared to 32-bit Windows Data Types

http://docwiki.embarcadero.com/RADStudio/en/64-bit_Windows_Data_Types_Compared_to_32-bit_Windows_Data_Types

- Atomicity on x86

https://stackoverflow.com/q/38447226/4267244

- Why is integer assignment on a naturally aligned variable atomic on x86?

https://stackoverflow.com/q/36624881/4267244

https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

- ARM Advantages of natural data alignment

https://developer.arm.com/documentation/dui0472/c/compiler-coding-practices/advantages-of-natural-data-a

- MemAtomic—checks whether memory access is atomic

https://github.com/gabr42/GpDelphiCode/tree/master/MemAtomic

Quality Portal Reports

- Generic Class Helper support for Delphi

https://quality.embarcadero.com/browse/RSP-10336

- Helpers for Interface types

https://quality.embarcadero.com/browse/RSP-16763

- Expanded helpers

https://quality.embarcadero.com/browse/RSP-13340

- PPL Task Continuation and Cancellation Token support

https://quality.embarcadero.com/browse/RSP-13286

- Compiler turns weak reference captured by anonymous method into strong one

https://quality.embarcadero.com/browse/RSP-19204