Coding in Delphi

1. 写在前面的话

本次翻译纯属爱好，目的是提高对英文文档的理解和阅读能力，本文档大部分采用直译的方式，而且保留了原来的英文。目的只是辅助大家理解，不喜勿喷。翻译的认为不正确的，强烈的欢迎大家入群讨论(DIOCP 群: 320641073)一起翻译，衷心希望有英文功底加入进行指导。

常驻翻译：Ryan, D10.天地弦

客串翻译：小生(骗术师)....和 DIOCP 群内的小伙伴。

友情审核：bp，逆水寒，爱国贼。

文档整理：D10.天地弦

此次翻译利用中午休息的时间进行的，大家每天在一起翻译一小段，然后讨论的方式，力争每个词都理解到位。把他整理成 PDF，希望能够帮到大家。

2. 异常和异常处理

2.1. 介绍

Added into Delphi back at the very beginning with Delphi 1, exception handling was a fundamental change to the way we thought about and wrote our code.

自 delphi1 开始加入的异常处理，从根本上改变了我们的思考和代码编写的方式。

Unfortunately, despite almost twenty years of use, there are still many misconceptions and misunderstandings about how exceptions work and especially about how they should be used.

不幸的是 尽管使用了近 20 年，对异常的原理特别怎么样使用方面还存在着很多的误解。

In this chapter, I'll look at how exception handling should be done.

在这一章，我会着眼于应该怎样进行异常处理。

I'll start out with examples of the wrong way to use and handle exceptions.

我会用错误的方式使用和处理异常的例子开始。

Using that as a base, I'll then discuss proper ways for using exception handling.

依次为基础,我会讨论正确使用异常处理的方式。

Used improperly, exception handling can actually cause more problems and errors than it prevents. Used correctly, they can aid you in writing clean, well-designed code that is easy to maintain. I'll assume that you are familiar with exception handling syntax, and the basics of how exceptions work.

使用不当，异常处理实际上会导致更多的问题和错误比预防。使用得当，他可以辅助你编写干净的、设计巧妙且易于维护的代码。本章假定你已经熟悉的异常处理的语法和了解异常的基本工作原理。

2.2. 结构化异常处理(SEH)

An exception is a language feature that allows a programmer to stop execution of a process or thread immediately, but intercept that “stop” at any point in the call stack when necessary. Structured exception handling is a combination of language features and good design that makes use of exceptions in order to allow programmers to make useful assumptions when writing code and, most critically, respond correctly when those assumptions turn out not to be true.

异常是一种语言特性(它允许程序员在一个进程或者线程中立即停止执行), 但是当有需要的时候拦截那个"停止"在调用堆栈的任何一个位置。
结构化异常处理是语言特征和良好设计的组合,利用异常是为了让程序员在编写代码的时候做有用的假设，最重要的是当这些假设不成立时做出正确的回应。

These assumptions are often called preconditions – they are things which must be true for a method to succeed. For example, a method that deletes a record from a database might have the precondition that the user is logged into the database before it is run. Because the program’s user interface is structured to only run the method after logging in, a programmer might assume that
this precondition will always be true. But what happens when the database server crashes?

这些假设通常被成为一个方法执行成功的前提条件。比如一个方法：从数据库里面删除一条记录，这个方法的前提条件是用户已经登陆到数据库。用户这个程序的用户接口是一个只有用户登陆后才能运行的方法, 程序员假定这个前提条件总是成立，但是如果数据库服务器奔溃了呢?

The programmer could address this by checking to ensure that the server is connected at the start of every method. But this is by no means the only precondition, and may lead to duplication of code. Or the programmer might forget to include all of the relevant checks when writing a new method. Structured exception handling provides an elegant solution to all of these issues. It ensures that a method will fail if the preconditions are not fulfilled, and furthermore that it fails in such a way that the developer can recognize and respond to the failure.

当然程序员可以通过在每个函数的开始地方检测是否已经登陆到数据库服务器来解决这个问题，但是这可不是唯一的前提，而且这种解决方法会导致很多重复的代码，在写新的方法的时候也有可能程序员忘记了加入这些检测。结构化异常处理提供了这些问题完美的解决方案。以确保 在前提条件不满足的情况下 方法不会执行成功！而且这种失败可以提供给开发者能做出正确的反应。

2.3. 怎么不使用异常(How Not to use Exceptions)

Much of the work I do involves working on existing projects that have been struggling – most often
because of poor design – and rewriting code that wasn’t well done in the first place.
我涉及的大量工作是在现有的项目中苦苦挣扎 -- 通常是因为糟糕的设计和一开始没有做好而导致的代码重写。

One of the most common coding errors I see is the misuse of exceptions – sometimes really ugly misuse. To start off, I'll go through a few "Don'ts" for exception handling use, discussing why each is not a good technique.

重写代码在项目的前期都是不好的。在我参与的项目中，我看到的最常见的编码错误就是异常的使用不当，有时候是相当的不当。我通过一些[注意事项]来说明异常处理的使用，讨论每一个不恰当的使用。

Don' t Eat Exceptions

不要吃异常

Probably the most common — and really egregious – misuse of exceptions is the "eating" of exceptions. Very often I'll see code like this;

异常使用不当中最常见的就是：粗暴的吃掉异常。我经常看到一些这样的代码：

```
try
   SomeRoutineThatSometimesCausesAHardToFindAccessViolation
except
end;
```

For the Love of Baby Elvis, please do not do this. As you can see, this code will "eat" any exception that gets raised in the called routine. Very often, the code in the try block raises an error that isn't easily found, and rather than do the hard work of actually tracking down the error, the programmer will take the easy way out and just eat the exception. Sometimes, the reason for eating the exception is nothing more than the desire to never let the end user see any error messages. If that is your goal, however, you should do so in such a way that you don't conceal the errors from the rest of your code, as well.

正如你所见，这段代码会吃掉所有调用过程中抛出的异常，最常见的是我们很难发现在 try 块中的代码引发的错误,程序员宁可选择逃避吃掉异常,也不愿意认真的去追踪错误.往往吃掉异常的原因只是不希望最终的用户看到错误信息,如果这是你所期望的，那也不应该去隐藏你其他代码中的错误.

Rest assured, the user will see no error messages as a result of this code. Every single exception that could possibly arise from this code will be suppressed – database exceptions, out of memory exceptions, hardware failures, anything. This means that your program may return incorrect results while appearing to succeed. It is better to clearly indicate failure than to silently make an error that could result in an incorrect paycheck or worse!

的确，用户看不多任何的异常信息。任何一个异常信息都被这写代码屏蔽了，数据库异常，内存溢出，硬件错误，等等。这意味着你的程序总是会显示执行成功，即使刚刚说的这一切都发生了。清楚的标明故障比让默默的犯错而导致付错薪金或造成更大错误来说是更明智的选择。

The only time I can think of when simply eating an exception is acceptable is when you need to

prevent an exception from propagating across module boundaries.
这样简单的吃掉异常，我唯一能想到的是在跨模块之间阻止异常传播。

If you are doing inter-module programming, for instance, code that will run in a DLL, you shouldn't let any exceptions escape from the current module.
如果你跨模块编程，例如，代码运行在一个 DLL 里面，你不应该让任何异常逃离当前这个模块(DLL).

In this case, using an empty exception handler on the outer boundary of a DLL call do that for you.
在这种情况下，你可以在 DLL 调用边界中使用一个空异常来处理.

But unless such is the case, eating exception handlers like this should be considered a gross error and a coding horror.
但是除了这种情况，吃掉异常会被认为是一种严重的过失和的非常不合适的编码。

Even in this situation, you should somehow log the exception, or acknowledge it in some way. Eating exceptions means that the information about the error – which could make fixing it easy – is gone for good.
即使是在这种情况，你应该以某种形式记录这些异常或者以某种方式承认他。吃掉异常将使得原本可以很容易修复问题的信息再也不会出现了.

Your customer may never realize there is a problem, and even if they do you may not be able to figure out why it happened and how to fix it.
你的客户可能不会意识到这是一个问题，即使他们意识到这是个问题，你也不见得可以找出这种问题是怎么发生的，也不知道怎么去解决它。

Bottom line: just don't eat exceptions
底线:不要吃异常

Don't Generically Trap Exceptions

不要一般的抓捕异常

Sometimes I see code that looks like this:
有时我看到一些这样的代码:

```
try

  SomeCodeThatMightCauseAProblem
```

```
except
  on E: Exception do
  begin
    MessageDlg(E.Message, mtWarning, [mbOK], 0);
  end;
end;
```

And I think "That's sort of like drinking Caffeine-Free Diet Coke" - in other words, why bother?
我认为"就像喝无咖啡因的健怡可乐",换句话说，何苦呢？

This code doesn't do anything other than report an exception that will likely be reported anyway. Actually, it does one other thing, and that is to stop the exception in its tracks, The exception will be handled locally, and will never be allowed to escape the current scope, In addition, it will trap all exceptions, including ones that you may very well not want trapped.
这些代码除了看上去总会报告一个异常之外没有做任何事情。实际上
它还做了另外一件事，就是阻止了异常的传播方向(轨迹)，异常会被就地处理，永远都无法脱离当前的范围，并且这也会使得所有的异常包括那些你可能非常不想捕捉的异常被捕捉到。

The only time that you might even consider using this construct - which is only slightly better than eating the exception altogether - is when you known that the calling routine doesn't want to handle any exceptions or when the calling routine expect to handle the specific exception.
唯一的情况你可以考虑使用这种结构(稍微比吃掉所有异常好一点点)是当你知道调用他的过程不想处理任何异常的时候或者期望处理特定异常的时候。

For instance, the TClientDataSet has an OnReconcileError event that actually passes an exception into it If were doing some batch processing with a Clientdataet, then allowing this exception to bubble up the stack will stop the loop. In this case, you might want to generically trap all the exceptions that are passed into the event handler.
例如，TClientDataSet 有一个 OnReconcileError 事件，实际上传递一个异常进去给他,如果用一个 ClientDataSet 做一些批量处理，然后让这个异常冒出来停止堆栈的循环。在这种情况你可能想一般的捕获传递到事件处理程序的所有异常。

Don't Go Looking For Exceptions

不要去寻找异常

Exceptions are fairly expensive in terms of processing power to create and handle, and so you shouldn’t be creating them as a matter of course.
异常的创建和处理对处理能力的开销是十分大的，所以你不应该理所当然地去创建它他。

In addition, you shouldn't be creating them on purpose or using them for the purpose of error checking, per se.
另外，你本身也不应该以检查错误为目的去创建和使用他们。

For instance, you might be tempting to do something like this admittedly contrived example:
例如，你可能尝试着去编写下面这种做作的例子:

```
function StringIsInteger(aStr: string): Boolean;
var
   Temp: integer;
begin
   Result := True;
   try
     Temp := StrToInt(aStr);
   except
     Result := False;
   end;
end;
```

This code will do what you want it to do – determine whether a string holds a valid integer, but it will also probably raise a lot of exceptions that can hurt performance, especially if the number of times that it is expected to return false is high.
这代码将实现你想做的事情(判断一个字符串是否是一个有效的整数)，但是也有可能引发很多影响性能的异常，尤其是期望返回 false 很频繁的时候。
Performance issues aside, this is an incorrect use of exception handling because it is not a violation of a precondition for the method.
撇开性能问题不说，这不是异常处理的正确用法，因为这不是用来判定一个方法是否违规的先决条件.
The method is clearly designed to accept non-integer arguments, hence it should not use the exception method, even internally, to handle such a case. A better
implementation of this method would use the TryStrToInt function in SysUtils.
这个函数很清晰的被设计成接受非整型数字参数，因此即使在内部也不应该去使用异常方法处理这种情况，这个方法更好的实现是使用 Sysutils 里的 TryStrToInt 函数.

Don't Use the Exception Handling System as a Generic Signaling System

不要将异常当作信号传递系统来使用

```
type
  TNormalBehaviorException = class(Exception);
...
```

```
begin
    SomeCodeThatDoesNormalThingsAndDoesntHaveAnyErrors;
    raise TNormalBehaviorException.Create('Something perfectly normal' +
    'and expected happened');
end;
```

You might be tempted to write the above code as a means of signaling the calling code about some type of information, especially if your custom exception handler has additional information in it that can be 'signaled' back to the calling routine.

你可能为了传递给调用者某些类型的信息就随手写出上面的代码，特别是你自己定义的异常处理过程有一些额外的信息可能会被传回到调用者。

Remember that exceptions are a flow control mechanism as well as a tool for conveying information. Raising exceptions when you simply want to send a message can have unexpected consequences for program flow.

你要记住的是异常不但是一种信息传递工具同时也是一种流程控制机制,为了简单发送一个信息而引发异常可能会导致程序流程出现非预期的后果 (意思是会打乱原来程序预期的执行顺序)。

Raising unneeded exceptions can be very irritating, even if your application always captures and handles the exception in question, or your custom exception descends from EAbort and won't result in an error message that they user will see.

引发不必要的异常是很令人讨厌的。即使你的程序总是会捕捉和处理这些异常问题，或者你自定义的异常源于 EAbort,不会导致用户看到错误信息。

For one thing, this is a pretty processor-intense way of signaling and passing information.

首先 这是发信号和传递信息非常不错额外处理方式.

Secondly, the error will appear at design-time, driving you and other developers crazy, as well as at run-time, annoying users. (A very popular third-party library does this sort of thing, and it drives me to distraction.)

其次这些错误会在设计期间出现，把你和其他开发者会逼疯。而且在运行时会骚扰用户(一个非常流行的三方库也做类似的事情，我快要发疯了.)

As a general rule, if you are raising an exception that pretty much requires your users to add that exception to the list of exceptions to be ignored by the IDE, then you should think twice about raising that exception at all.

一般情况下，如果你引发一个你的用户在 IDE 中非常需要添加到异常忽略列表中的异常，那你就需要三思一下是不是完全没必要这么做!

2.4.怎么样正确的使用异常(How to Use Exceptions Properly)

Now that you've seen some of the ways not to use exceptions, here are some tips for properly using Delphi's exception handling system.
现在你已经看到了一些不使用异常的方式，这里是一些正确使用 Delphi 异常处理系统的小巧门。

Use Exceptions to Allow Your Code to Flow without the Interruption of Error Handling Code

使用异常使你的代码在没有错误处理代码中断的情况下运行流畅。

One of the main purposes of exception handling is to allow you to remove error-checking code altogether and to separate error handling code from the main logic of your application.
异常处理最主要的一个目的是允许你从应用程序的主要逻辑中移除掉错误检测代码和分离错误处理代码。
With exception handling, you can write your code as if nothing ever goes wrong, and then wrap that code up with try...except blocks to deal with any of the errors and problems that may occur. This enables your code to run more efficiently, as it isn't constantly checking parameters and other data to make sure that it is in the proper form before doing anything with it.
使用异常处理，你可以在没有任何错误发生前提下编写你的代码，然后用 try..except 块包住你的代码来处理可能发生的任何错误和问题。这样使你的代码运行更高效， 因为不用经常去检测在使用参数和其他数据之前确保它是否符合格式。

One way to separate code and exceptions is to handle exceptions centrally. TApplication has an event that allows you to do just that – the OnException event. You can use this event to deal with all exceptions of any type that aren't otherwise handled by your application. You can use this event to log your exceptions, or provide specific handling for specific types of exceptions.
一种分离代码和异常处理的方式是集中处理异常。TApplication 有一个事件(OnExeption)允许你做这件事情。您可以使用这个事件来处理其他没有（经你的应用程序）处理的任何异常。你可以使用这个事件去记录你的异常,或者为特定类型的异常提供特殊处理。

Application Writers Should Trap Exceptions

应用程序编写者应该抓捕异常

As will be discussed below, components and library code should be the main source of exceptions;

正如下面将要讨论的，组件和库代码应该是异常的主要来源;
that is, components and library code should be the place where most exceptions are created and raised.
也就是说,组件和库代码应该是大多异常创建和引发的地方。
When writing applications, there is little need for you to create and raise exceptions.
当编写应用程序时几乎没有必要创建和抛出异常。
Application writers should mainly be about the business of handling exceptions that are raised by components and library code.
应用程序编写者应主要针对组件和库代码抛出的异常进行业务处理。

Trap only specific exceptions

只捕获特定异常

As noted above, you should never eat exceptions. What you should do instead is to trap only specific exceptions that might reasonably be expected to occur in your code.
如上所说，你永不该吃掉异常，取而代之你应该做的是捕捉你代码中可能发生的特定异常。

If you are doing a lot of math, you might want to trap for EMathError exceptions. If you are doing a lot of conversions, you might want to trap for EConvertError.
如果你正在做大量算法运算,你可能想捕获 EMathError 的异常.如果你正在做大量转换,你可能只想捕获 EConvertError。
Likewise, when doing database work, you might want to look out for EDatabaseError exceptions.
同样地，当你做数据库工作的时候，你可能只想关注 EDatabaseError 异常。

But even those errors might be a bit general. For instance, within database code, there may be specific descendant classes of EDatabaseError that occur when specific database actions are taken.
但即使是这些错误也可能有一些笼统了．举个例子，在数据库代码中，当特定的数据库操作被执行时可能会发生派生自 EdatabaseError 的特定异常。
So if you are opening a query, perhaps you should trap for exceptions that occur only on the opening of datasets, rather than the more generic EDatabaseError.
所以当你正在打开一个查询时，也许你应该捕捉那些只会出现在打开数据库操作时的异常特例,而不是更笼统的 EdatabaseError。

As I mentioned above, I see code that eats exceptions added because the developer (or manager, or someone not thinking very clearly) never wants the user to see any errors. The way to deal with that is to trap the specific exception that the user is seeing. For instance:
如上我所提到的，我明白了添加那吃掉异常的代码是因为开发者(或经理、或某些没有经过深思熟虑的人）不希望用户看到任何错误。处理那种问题的方式是捕获用户看见的特定异常,

例如

```
try
  SomeCodeThatRaisesAnEConvertError;
except
  on E: EConvertError do
  begin
    // Deal with this specific exception here
  end;
end;
```

This code is better than simply eating all exceptions, no matter what you do with the exception, because at the very least, it will only trap that one exception and not every exception that comes along.
无论你对这个异常做何处理，这样的代码比简单的吃掉所有的异常(的代码)要好， 因为至少，他只会捕获那一个异常而不是每个出现的异常。

Furthermore, database exceptions (and some others, like COM errors) generally include an error code, and you may wish to trap only errors with a certain error code and allow others to surface. You can do this as follows:
此外,数据库异常(和一些其他的，像 COM 错误)一般包含一个错误代码，你可能希望至捕获某一些错误代码的错误然后让其他一下错误浮现。你可以像下面一样做：

```
try
   SomeCodeThatRaisesAnEConvertError;
except
   on E: EIBError do
   begin
      if E.ErrorCode = iSomeCodeIWantToCatch then
      begin
         // Deal with this specific exception here
      end else
      begin
         raise; // re-raise the exception if it's not the one I handle
      end;
   end;
end;
```

Another reason to trap exceptions as far down the hierarchy chain as possible is that there may

be future exceptions created that descend from the more generic Exception class. For instance, if you have this code:
另外一个尽可能捕获层次链下层的异常的原因是未来可能会创建继承自更笼统异常类的异常，例如，你有如下的代码:

```
try
   SomeDataset.Open
except
   on E: EDatabaseError do
   begin
   // Handle exception
   end;
end;
```

And then I come along and declare:
然后我这样申明了

```
type
  ENxStrangeDatabaseError = class(EDatabaseError)
```

My new and strange exception will be trapped by your code, and perhaps that isn't what you want to have happen. Obviously you can't prevent this from ever happening, as a developer can descend from any existing exception, but you can make it happen less by trapping exceptions at the bottom of the class diagram.
你的这段代码将捕获我这个新陌生异常，也许这不是你所想要的，明显你永远不能阻止这样的事，因为开发者可以从任何已存在的异常中继承，但是你可以通过捕获类图中最底端的异常来减少这样的情况.

Bottom line: Trap exceptions as far down the class hierarchy as you can and only trap those exceptions that you are planning on handling.
底线是：尽可能的捕获那些类层次中最下层的异常，并且只捕获你打算处理的异常。

Component Writers and Library Code Writers Raise Exceptions

组件和库代码编写者引发异常。

Exceptions do not mysteriously appear. The vast majority of them are created and raised within framework code. (Some can actually occur outside the purview of Delphi code.) And it is perfectly acceptable for you to raise your own exceptions as well.

异常不会无缘无故的出现。他们绝大多少在框架代码里面被创建和引发。(事实上有一些发生在 Delphi 代码范围以为.) 引发你自己的异常对于你也是完全可以接受的

As a general rule, you should raise specific exceptions in your library code and in your components. That way, application writers can trap those specific exceptions in their code as discussed above.
通常情况下，你要在你的代码库和组件中引发一个特定的异常。那样的话，应用程序编写者可以按如上讨论的方式捕捉到代码中所有的具体异常

You should write library code and component methods in such a way that they do one of two things:
你应该按这种方式写库或组件方法代码，使其只能有两种可能性:
they either execute successfully and return, or they raise an exception. Application writers should assume the same thing – that a routine that is called will either return successfully (with a valid result if the routine is a function) or that it will raise an exception.
要么执行成功和返回，要么引发一个异常。作为应用程序开发者应该做同样的假设(一个过程被调用要么返回成功(如果是 function 要有一个有效返回值)，要么引发一个异常)

Raise Your Own Custom Exceptions

引发你自己自定义异常

When you do raise exceptions, always raise your own custom exceptions. For instance, I have a library of code that I use in a file called NixUtils.pas. In that file I declare
当你引发一个异常的时候，总是去引发你自己自定义的异常。例如，我有一个库代码引用了 NixUtils.pas，在那个文件里面我定义了

```
NixUtilsException = class(Exception);
```

And any exceptions that I raise in the routines found there are either of that type or are descendants of NixUtilsException.
然后我在过程中引发的任何异常(发现)要么是 NixUtilsException 要么是 NixUtilsException 的派生类型。

Doing this allows users of your library code and components to do what I've exhorted you to do above: trap only specific exceptions.
这样一来，允许你库代码和组件的使用者去像我上面劝你去做的那样:只捕捉特定的异常。

Don't be afraid to declare your own exception classes and then descend from them, even to the point of having specific exceptions for specific routines.
不要害怕去定义你自己的异常类和它的派生类，甚至有时候需要为特定的过程定义特定的异常。

This allows your users to trap exceptions with whatever level of granularity they require.
这样允许你的用户以他们所需的任何级别的颗粒度去捕捉异常.

Let Your Users See Exception Messages

让你的用户看到错误信息。

If you are tempted to hide all errors from the tender eyes of your users, ask yourself this question:
如果你想从你用户温柔的眼睛中隐藏错误信息(不想让用户看到错误信息),问一下你自己这问题:

Which is worse, having your users see error messages, or having the application roll along as if nothing has gone wrong, possibly leaving a trail of bad calculations and corrupt data?
让你的用户看到错误信息，或者让应用程序在好像什么错误都没有发生的情况下继续运行，可能留下一个个错误的计算和不正确的数据，哪个更糟糕？
Sadly, I've seen a lot of code that answers that question with the latter option rather than the former.
不幸的是, 我看到很多代码回答了这个问题(更愿意选择后者).
That's how you end up with code that has empty exception handlers and eats any and all exceptions.
那就是你最终的代码:一个空异常来处理和吃掉任何所有的异常。

Exceptions occur because something is wrong. Ignoring them can have unexpected results. Eating an EOutOfMemory exception can have disastrous results, because your application – and your users – will continue on as if nothing bad has happened, when in fact bad things have happened.
异常的发生是因为有些东西错误。忽略他们可能会得到意想不到的结果。吃掉一个 EOutOfMemory 异常会得到一个灾难性的结果，因为你的应用程序和你的用户会继续好像没有好像没发生什么糟糕的事情。
Users are fearful of dialog boxes, as many user interface experts have noted, but if your dialog box actually gives them something to do, then the dialog boxes may be more useful than they normally are.
许多用户界面专家已经指出, 用户害怕对话框. 但是如果你的对话框实际上给他们要做的事情，那样的话这对话框可能是非常有用的比起平常的对话框。

Feel Free To Provide Good Exception Messages

随意的提供好的异常信息

You don’t have to be terse and uninformative with your errors. When you raise an exception, feel free to make the message passed up the stack as informative as you like:
你错误不一定简洁和无信息的。 当你引发一个异常的时候，随意的编写这个信息沿着堆栈中向上传递,这个信息你喜欢多详细都可以:

No need to do this:
不需要这样做:

```
type
  ESomeException = class(Exception);


procedure CauseAnException;
begin
  raise ESomeException.Create('Boring message');
end;
```

when you can do something like this:
需要的时候你可以下面这样做：

```
type
  ESomeException = class(Exception);


procedure CauseAnException;
begin
  raise ESomeException.Create('An exception occurred, and here is exactly what happene
d....');
end;
```

Write full, descriptive errormessages. You can even include the procedure name, the TObject.ClassName, or whatever you like in the message.
写满描述性错误信息.甚至你可以在信息里面包含过程名，类名，或者你喜欢的一些信息。
You can, in fact, enhance an existing exception’s error message, set the new exception as the “outer” exception, and raise your new exception with the original exception as the inner exception:
事实上，你可以加强一个现有的错误的信息，作为这"外部的"异常来设定这新的异常，然后用原有的异常作为内部异常来引发这个新的异常。

```
type
  EMyException = class(Exception);
  EMyInnerException = class(Exception);


procedure RaiseInnerException;
begin
    try
       raise EMyException.Create('This is the message from EMyException');
    except
      on E: EMyException do
      begin
        E.RaiseOuterException(EMyInnerException.Create('This is the message from FMyIn
nerException'));
      end;
    end;
end;
```

Since the exception handler includes the call to raise an outer exception, the exception isn't being eaten here, but the code provides information about where the error occurred, etc. You include information about where users can go for help, what they should do if the error persists, etc.

自从这个异常处理包括调用引发一个外包异常，这个异常没有在这里被吃掉。但是这代码提供了关于在哪里产生错误的信息，等. 你可以包含关于用户在哪里可以得到帮助，如果这个错误持续存在他们应该怎么做，等。

Provide Two Versions Of A Library Routine

提供两个版本的库函数

Sometimes people don't like routines that return exceptions. Well, okay, why not accommodate them? The RTL does this – it sometimes provides two functions that do the same thing, with one raising an exception on failure and the other returning nil or some error value, depending on the result.

有些时候人们不喜欢函数抛出异常，嗯，那好，何不满足他们呢? RTL 就是这么干的，某些时候它会提供2个函数做同样的事情，一个函数是错误的时候抛出异常，一个则是返回 nil 或者一个错误码，这取决于结果.

The FindClass/GetClass pair comes to mind. FindClass locates and returns a class type by name, and if it can't find it, it raises an exception. GetClass, on the other hand, will simply return nil if the class cannot be found.

让我想到了 FindClass/GetClass 这对函数。FindClass 根据名称查找和返回类，然后如果找不

到这个类，这个函数就会引发一个异常。GetClass 在另外一个方面，当这个类没有被找到的时候将简单的返回 nil。

2.5. 结论 Conclusion

It's quite easy to fall into the trap of using exception handling improperly. The ability to make errors and problems 'disappear' is quite tempting. However, misunderstanding and misusing exceptions in your code can lead to some real problems, including untraceable crashes and data loss. The proper use of exceptions can make your code easier to read an maintain. Use exceptions wisely, and you'll be able to product robust, clean code.

错误的使用异常处理很容易掉入陷阱中. 异常让问题和错误"消失"的能力是很诱人的`， 不管怎么样, 误解和错误的在代码中使用异常都会导致真正的问题, 包括不可追踪的程序崩溃和数据丢失, 异常正确的使用可以使得你的代码容易阅读和维护. 明智的使用异常能产生强健,干净的代码.

3. 使用接口(Using Interfaces)

本章常驻翻译：Ryan， D10.天地弦，小生(骗术师) ， QDAC- swish
友情翻译: 五毒公主ら
友情坐镇：bp，逆水寒，爱国贼。
文档整理：D10.天地弦

3.1. 介绍(Introduction)

Just about everything that I write about in this book will be predicated on the use of interfaces. If you aren't using interfaces for just about everything you do, you need to start. I once tweeted

我在这本书里写的几乎所有的东西都是基于接口的使用之上, 如果你对所做的任何东西没有使用接口的话, 那么你需要开始用了. 我曾经在推特上写过：

"If I could teach a new programmer one thing it would be this: Program to an interface, not an implementation.1"

如果我能教一个新的程序员一件东西，那将是: 针对接口编程, 而非针对实现编程

Program to an interface, not an implementation
对接口编程,而不是对实现

When you use interfaces, you can decouple yourself from any particular implementation. When one module of code isn't directly connected to another module of code, that code is said to be "loosely coupled". And as I will probably repeat countless times throughout this book, loosely coupled code is a very good thing.
当你使用接口编程的时候，你就可以分离任何一个独立的实现 做到去耦，当一个模块的代码不是直接关联到另外一个模块中的代码的时候，我们称之为”松散耦合的”，正如贯穿本书里我可能无数次强调的，松散耦合的代码是非常棒的.

3.2. 解耦(Decoupling)

All through this book, I'll talk a lot about decoupling your code and why that is really good. But a definition here is probably a good idea.
贯穿本书，我将讨论了许多有关解耦代码和这样做会带有什么样的好处的话题。不过在这里定义也许是一个好主意。
Code can be said to be decoupled when your classes are designed in such away that they don't depend on the concrete implementations of other classes. Two classes are loosely coupled when they know as little about each other as possible, the dependencies between them are as thin as possible, and the communication lines between them are as simple as possible.
当你的类如果是这样设计的: 它们不依赖于其他类的具体实现，这样的代码可以说是松散耦合的. 两个类当相互之间了解的尽可能的少，相互依赖尽可能的弱，彼此之间通讯尽可能的简单，这样的两个类是松散耦合的.

In other words, decoupling is the process of limiting your dependencies to abstractions as much as possible. If you want to write good, clean code, you will try to couple only to abstractions; interfaces are abstractions. Much of what will be discussed in this book will be about using interfaces to code against abstractions. As I tweeted above, this is critical to writing good code that is easy to maintain.
换句话说，解耦就是限制你尽可能多的依赖抽象的过程，如果你想写出漂亮干净的代码，你该试试只耦合抽象，接口就是抽象的，这本书里很多将要讨论的就是使用接口写出针对抽象的代码，如我在上面推特里写到的，这是写出容易维护的好代码的关键.

In Delphi, decoupling starts by limiting what goes into the uses clause of any given unit. Every time you use code from another unit, you are creating a connection or a coupling of code. You should try as much as possible to expose units that declare only Delphi interfaces to other units. You should endeavor to put as little as possible in the interface section of your units. This will allow you to limit the coupling of your code – if you don't put it in the interface section, it can't be coupled to.

Delphi 中,解耦从限制任何进入 uses 的单元开始. 每当你使用另外一个单元的代码时，你是在创建一个关联或者说是耦合代码，你应该尽可能的尝试去暴露那些只申明了接口的单元给其他单元。你应该努力做到尽可能少放代码在你单元的 interface 部分. 这将限制你代码的耦合 - 如果你不不放在 interface 节中,那它不会被耦合.

Once you have limited the connections between your units, you can then start limiting the connections between your classes. Classes have to connect to each other somehow – otherwise you can't build a system. But if you are going to connect things together, you want those connections to be as thin as possible. That's where interfaces come in.

一旦你已经限制了单元之间关联，你就可以限制类之间的关联. 类不得不以某种方法连接在一起，否则你就不能构建出一个系统，但是如果你将要把一些东西关联在一起，你就要让这些关联尽可能的弱，那就是接口的用武之地.

Of course to program against an interface, you have to first know what an interface is.

当然针对接口编程，你首先必须知道什么是接口

3.3. 什么接口(What are Interfaces?)

Interfaces are kind of hard to describe, but I'll have to try anyway, eh?

接口是很难描述的，但是我无论如何必须的试试，额？

The dictionary describes an interface as:

Interface – n. : A point where two systems, subjects, organizations, etc., meet and interact.

字典中是这样描述一个接口：

接口 - 名词. : 在两个系统，主题，组织等等之间接触和互动的一个点.

In the general case of code, an interface is the means by which code modules – usually classes – interact with each other to perform specified actions. Your code needs to do something, and interfaces are the definitions that allow code modules to work together.

代码中一般情况下，接口是指代码模块-通常是指类-彼此交互执行特定的动作。你的代码要完成一些功能，而接口就是模块之间协同工作的约定

That's the general definition – but Delphi has a reserved word interface which has a specific syntactical meaning and which provides a specific functionality. So what are interfaces in Delphi? Interfaces are an abstract definition of functionality.
那是一般的定义，但是 delphi 有一个保留的 interface 关键字， 这个关键字有一个特殊的句法含义，还提供了特殊的功能。在 Delphi 中接口又是什么意思呢？接口是功能的一个抽象定义。

Okay, that sounds all intellectual, doesn't it, but it's true.
OK,这听起来太学术了是吧，但确实如此。

An interface declaration defines a set of properties and methods that you can use for a specific purpose. Interfaces declare and require no specific implementation for that purpose. They define what a class can do and limit the exposure of a class to only those things defined by the interface.
接口申明定义了一系列的属性和方法，你可以用他们达到特定的目的。为了那个目标接口不需要特定的实现那些申明。他们定了一个类能做写什么和限定一个类只能暴露接口定义的这些东西。

Of course, an interface must be implemented to be useful. And when you use an interface, you ultimately don't care about how it is implemented; you just know that you can call the methods and properties on the interface and the functionality you need will be provided. Implementation hiding (i.e., abstraction) is one of the main purposes of using interfaces.
当然，一个接口必须被实现才是有用的。当你使用一个接口时，你根本不需要关心他是怎么实现的; 你只需要知道你可以调用这个接口的方法和属性，而且你需要的功能将会被提供。隐藏实现(即，抽象)是使用接口的一个主要目的。

3.4. 接口无处不在(Interfaces Everywhere)

We see interfaces in our lives every day. Electric plugs are an interface. In the United States, our electric plugs have two vertical, rectangular slots for the positive and negative nodes, and one round hole below for the grounding node. What is that but a definition of functionality? In order to implement the interface, you merely provide a plug that matches the interface, and your device will work.
在日常生活中我们每天都能看到接口，电源插座就是一种接口，在美国，我们的电源插座有 2 个垂直的矩形插槽分别接火线和零线，还有一个圆形的孔是接地线，这不是接口的功能定义是什么呢？为了实现该接口，你只需要提供一个符合该接口的插头就能让你的设备工作了.

A plug is just an interface
(一个插座就是接口)

You can plug in hair dryers and computer monitors – the interface doesn't care. These items could be said to be "powerable" as a result.
你可以接上吹风机或者电脑显示器，接口不关心这些，这些东西最终都可以被看作是电源驱动的.,

Delphi developers are already doing this with interfaces, perhaps without even realizing it. Well, a form of interfaces, anyway. Every Delphi unit has a section in it called the interface section, right? This section declares the functionality that the unit provides. Only those things declared in the interface section of a unit may be used by other units. Classes and methods in the interface section are implemented in the implementation section of the unit, and that implementation is hidden from
users of the unit. You can't put implementing code in the interface section. Code declared in a unit's implementation section is not usable outside the unit. In this way, a unit is a simple example of what interfaces are for: declaring functionality and hiding (but still providing) an implementation.

delphi 的开发者们或许都还没有意识到接口就已经在用它做开发了， 嗯，接口无非只是一种形式，在 delphi 的单元里都有一个小节叫做 inteface 段，对吧？这个节描述了该单元提供的功能，也只有这些在单元 interface 节中声明的东西才能被其它单元使用，interface 节中的类和方法是在单元的 implementation 节中实现的，这些实现对该单元的用户是隐藏的，你不能把实现代码放在 interface 节中，在实现节中声明的代码无法在单元外使用．单元其实就是一个简单的例子，它描绘出了接口是什么：声明功能和隐藏(但提供了)实现．

The second main reason for using interfaces is the one discussed above – decoupling. If you program against interfaces, you can write your code in such a way that it is never coupled – or "connected" – to anything but that interface. The more loosely coupled your code is, the less likely it is that a change in one place will affect code in another. In addition, by coding against an interface, you can replace the implementation with a better one without breaking anything. (Once again, I'll be harping on loosely coupled code throughout the book, and particularly in the discussion about Dependency Injection.) For now, the point is that interfaces are critical in decoupling your code, and that decoupled code is good.

在上面已经说过了使用接口的第二个主要原因 - 解耦.如果你要针对接口编程,你可以用这种除了接口外从不耦合或关联的方式.你的代码耦合度越低,你的改动对其他代码的影响就越小.此外,对接口编程,你就可以用更好的代码修改实现而不会产生任何破坏.(不止一次,我可能书中一直唠叨松耦合,尤其当讨论依赖注入时.)现在,接口是解耦代码的关键临界点,而且解耦的代码是很好的.

The code for an interface is actually quite simple: It is a declaration of methods and properties without an implementation. However, that simplicity can make them hard to understand. Interfaces as a language feature were first introduced in Delphi 3, and I remember very clearly thinking "Huh? Why in the world would you want to use interfaces?" Little did I know then, however, that interfaces are probably the single most effective coding tool in your arsenal. Once you truly understand what interfaces are for and what they can do, it will make it so much easier to write clean, uncoupled, testable code. (We'll be talking about the meaning of "testable code" in the chapter on Unit Testing.)

一个接口的代码实际上是相当简单的: 就是一些方法和属性的申明而且不带实现。不过那简单使他们难以理解(理解接口并不容易)。接口作为一个语言特征第一次引入是在 Delphi 3，而且我还记得非常清楚的思考过"呃，究竟是为什么想使用接口?" 那时候我根本不知道，不过接口可能是最有效代码工具。一旦你真正明白了接口是什么和接口可以做什么，他可以使你轻易的写出整洁，非耦合，可测试的代码。(我们会在单元测试(Unit Testing)章节讨论"可测试的代码")

Interfaces are similar to abstract classes in that they can define a set of methods that must be implemented. They differ in that they have no implementation themselves, whereas an abstract class might have implementation of non-abstract methods. The most analogous thing to an interface would be an abstract class that has all abstract methods and thus no implementation at all.

接口与一个抽象类相似之处他们可以定义一系列的必须被实现的方法. (接口和派生类)他们的不同之处在于:接口(They 是指 Interfaces)没有自己的实现，反之一个抽象类抽象类可能有

非抽象方法的实现。 和接口最相似的就是全部都是抽象方法而且都没实现的抽象类。

抽象类(Abstract Classes)	接口(Interfaces)
…can contain abstract methods which are normally overridden by descendants 包含通常被派生类重写的方法	…contain a list of methods that must be implemented by the implementing classes 包含一系列必须被实现类实现的方法。
…can be but normally are not Instantiated 通常不会实例化	… can't be instantiated because they are merely a definition of functionality 不能被实例化，因为他们仅仅是功能的定义
…can contain concrete functionality which will thus be provided to all descendants 可以包含具体的功能提供给所有派生类	…have no functionality at all attached to them. They depend entirely on their implementing class for functionality. 根本就没有附加任何的功能。他们的功能完全依赖于实现类
…are not reference counted and must be manually freed. 没有引用计算需要手工释放	…are normally reference counted and their implementing instance will by default be freed automatically. 通常引用计数，他们的实现的实例默认是自动释放的
… can only have single-inheritance descendants 只能有单继承派生类	… can be combined together in an implementing class that implements multiple interfaces 实现类可以实现多个接口

Interfaces cannot themselves be instantiated, and are only useful when implemented. They are pure references. An interface is implemented by a class that declares in its type definition that it will implement a given interface. The implementing class must then provide an implementation for the exact set of methods and properties as declared in the interface. A failure to do so will result in a compiler error. Any class can implement any interface – that is one of the strengths of the feature.

接口本身是不能被实例化的，只有当他们被实现了才有用，他们是纯引用， 一个接口被一个类实现，这个类在他的类型定义中声明他将会实现给定的接口，实现类必须之后为由接口中定义的一系列方法和属性提供实现，如果没有这样做将会导致编译错误，任何的类可以实现任意接口，这是接口特性的优点之一

A class may also implement any number of interfaces. In this way, Delphi can provide a functionality similar to multiple inheritance without all the trappings and difficulties of that feature. (I don't know if he originated the statement, but Zack Urlocker, Delphi's original Product Manager, has been quoted as saying "Multiple inheritance is the GOTO of the nineties." I always loved that, even though the nineties were a while ago.) Interfaces can also inherit from, and thus enhance, another
single interface.

一个类也可以实现任意多个接口，这样的话, delphi 就可以提供类似多重继承的功能，又可以摆脱掉多重继承中所有的陷阱和难点这样的特性(我不知道是不是他创造的语句，但是 delphi 先前的产品经理 Zack Urlocker 说过这样一句话"多重继承是 90 年代的 GOTO"， 尽管

90 年代刚刚过去，但我一直深爱着这句话 ），接口也是可以继承于另外一个单独的接口来得以加强.

3.5. 一个简单例子(A Simple Example)

For example, here is a declaration of a very simple interface:
例如， 这里有一个非常简单的接口申明:

```
type
  IName = interface
    ['{671FDA43-BD31-417C-9F9D-83BA5156D5AD}']
    function FirstName: string;
    function LastName: string;
  end;
```

Interfaces consist of a name (in this case, IName) and a declaration of methods and properties. By convention, Delphi interfaces start with the letter “I,” but that is not enforced. You can create interfaces with any name you want, but using the “I” makes it easier to identify an interface in your code. An interface includes no “real” code for implementing functionality. An interface cannot declare fields, variables or constants. Nor can they define scope such as private, protected, etc. It is purely a declaration of capability. Thus, every member of an interface is essentially public.
接口由一个名字(这里是 IName)和方法和属性的申明组成. 按照惯例， Delphi 接口一字母"I"开头，但那不是强制的。你可以用任何你想要的名字建立一个接口，但是使用“I"开头可以在你的代码中轻易的标明出一个接口。接口不包含实现功能的"真正"代码。接口不能申明字段(成员变量)。变量或者常量. 他们也不能定义 private, protected，等这样的范围。接口只是单纯的功能声明。所以每个接口的成员本质上是说是公有的(public)

In the case above, the interface says “Hey, when I am implemented, I’ll give you information about someone’s name.” It tells you what functionality is available. It does not tell you how the functionality will be implemented. In fact, the interface doesn’t care, and the user of the interface shouldn’t care either. The implementation of the interface might do any number of things to get the name information – randomly pick from a list, grab it from a file, or pull it from a database or some other data store. The interface itself doesn’t care and cannot dictate where the name comes from. All the interface knows is that it’s implementer will return a string – that’s it. It’s not even guaranteed that the string will be a person’s name, though that is obviously the intent.
在上面这个例子，接口说“海,当我被实现的时候，我会给你某个人的名字."，他告诉你功能是有效的。不会告诉你功能是怎么样被实现的。实际上，接口不会在意这些，接口的用户同样也不会在意。接口的实现可能为获取名称信息做很多事情-从列表中随机抽取，从文件中抓取，或者从数据库或者他数据存储中拉取一个。接口本身并不在乎也不能决定这名称的由

来. 所有的接口知道的是实现类将会返回一个字符串-就这样。他甚至不能担保哪个字符串是一个人的名字, 尽管那是有意为之。

Note that the declaration of the interface has a Globally Unique Identifier (GUID) right after the initial declaration.
注意, 接口的声明中有一个全局唯一标识(GUID) 直接跟在初始的声明之后
This GUID is used by the compiler to identify this interface.
GUID 是用于编译器来识别这个接口的
You can use an interface without the GUID, but much of the RTL and most frameworks that take advantage of interfaces will require a GUID be present. (You can generate a GUID any time you want in the IDE by typing CTRL+SHIFT+G)
你可以使用一个没有 GUID 的接口, 但是多数 RTL 和大多数的框架需要 GUID 的存在才能利用接口的优势,(你可以在在 IDE 中任何想要的时候按下 CRTL+SHIFT+G 来产生一个 GUID)

Of course, the purpose here is to create an implementing class that has some meaning when it goes into “Get me a person’s name” mode. For instance, you might have a form that gathers the name from the user. You might be iterating over records in a database and use the interface to grab each name as they are iterated. The point is that it doesn’t matter what the implementing objects are or what they do – they just produce a name when treated as an IName interface.
当然, 目的是创建一个具有获取一个人名功能的实现类(很拗口直译:这里的目的是创建一个实现类, 当这个实现类进入到"获取一个人名字"的模式时，这个类具有一定的意义)。例如, 你可能有一个窗体可以从用户那收集名字. 你可能正在遍历数据库中整个的结构并且当它们被遍历到的时候使用一个接口来抓取每个名字。重点是它不关心实现对象是什么或者它们做了什么, -当它们被当作 IName 接口时只是产生一个名字而已。

3.6. 实现接口(Implementing an Interface)

But of course, as you’ve guessed, an interface can’t do anything without an implementing class. Fortunately, Delphi makes it really easy to implement interfaces. To do so, you need to declare a class as implementing an interface, and then make sure that class implements all the methods in the interface.
当然，就像你想的那样，接口如果没有实现类不能做任何事情。所幸的是，Delphi 使得实现接口相当容易。这样做：你需要申明一个类来实现接口，然后确保这个类实现了接口中的所有方法。

In order to implement IName, you might declare a class as follows:
实现 IName， 你可以定义如下一个类

```
type

  TPerson = class(TInterfacedObject, IName)

  protected
```

```
    function FirstName: string;
    function LastName: string;
  end;

  function TPerson.FirstName: string;
  begin
    // Could get this from a database or anywhere, but for demo purposes,
    // we'll hard-code it
  Result := 'Fred';
  end;


  function TPerson.LastName: string;
  begin
     // Could get this from a database or anywhere, but for demo purposes,
     // we'll hard-code it
     Result := 'Flintstone';
  end;
```

Here are some things to note:
注意事项:

- .The class declares IName after the base class. (The base class is TInterfacedObject – we'll discuss this special class below.) It declares and implements the two functions required by IName.
- 类申明 IName 在基类后面.(这基类是 TInterfacedObject - 我们会在下面讨论这特殊的类.) 这个类申明和实现 IName 必须的两个函数。

- .The declaration of the two methods is defined as protected. The interface doesn't care what the visibility is – it will allow access to any implementing method regardless of the visibility – but by declaring the methods as a visibility not available to users of the class, you can ensure that the only way to talk to the class is via the interface.
- 这两个方法的申明被定义成保护的。接口不关心可见性是什么-它允许访问任何实现的方法而不管可见性-但是定义的方法作为类的用户不可访问的话，你唯一的途径和这个类对话是通过接口。

```
type
  IAged = interface
    ['{3D7E1BBE-6273-47A9-9CB7-CB31FDF6AB69}']
    function GetAge: integer;
```

```
    procedure SetAge(aValue: integer);
    property Age: integer read GetAge write SetAge;
  end;
```

Note that this means that the getters and setters can be called directly from the interface - even if they are private in the implementing classes. Of course, you should leave those getters and setters totally alone and only access them via the property.

请注意这意味着 getters 和 setters 从接口可以直接调用 - 即使他们在实现类的私有部分。当然你可以完全不理会 getters 和 setters 而只通过属性去访问他们。

Thus, you can declare an implementing object like so:

因此，你可以像下面一样申明实现对象：

```
TAgedThing = class(TInterfacedObject, IAged)
private
  FAge: integer;
  function GetAge: integer;
  procedure SetAge(aValue: integer);
public
  property Age: integer read GetAge write SetAge;
end;
```

A class can implement any number of interfaces, so you can have a class declaration like so:

一个类可以实现任何数量的接口，所以你可以有一个像这样的类申明：

```
TAgedPerson = class(TInterfacedObject, IName, IAged)
private
  FAge: integer;
  function GetAge: integer;
  procedure SetAge(aValue: integer);
public
  function FirstName: string;
  function LastName: string;
  property Age: integer read GetAge write SetAge;
end;
```

wherein the TAgedPerson class provides an implementation for all the interfaces included in its declaration.

TAgedPerson 类中提供所有申明在类中接口的实现

3.7. 进一步需要注意的东西 (Some Further Things to Note):

- An implementing class can have any number of other fields and methods that it needs or requires, as long as it has the methods defined by the interface. If you fail to provide all the necessary methods, the compiler will give you an error until you do. But the class can be as complex as you need it to be (though generally you should frown on complex classes, right?)
- 一个实现类可以有任意个需要的或需求的其他字段和方法，只要有接口中定义的方法就行.如果你没有提供需要的方法，编译器会提示错误，直到你提供了为止.但是这个类你想要多复杂就有多复杂. (尽管你一般可能看不惯复杂的类，对不?)

- The base class can be any base class as long as it implements the necessary methods for Delphi's interface reference counting. (TInterfacedObject does this automatically for you – more on TInterfacedObject and what reference counting is will be discussed below.) But I want to stress again – the base class can be anything. It could be a class you created. It could be a VCL class. It could be TButton or TClientDataset or anything. It doesn't matter, and the interface doesn't care, as long as you provide implementations for all the necessary methods.
- 基类可以是任何只要实现了 delphi 接口引用计数需要的方法的基础类,(TInterfacedObjecy 自动的为你做了这些., 更多关于 TInterfacedObject 和引用计数是什么将会在后面讨论.)但是 我想再次强调, - 基类可以是任何类，有可能是你创建的类. 可能是一个 VCL 类，可能是一个 TButton 或者 TClientDataset 或者其他的.这都无关紧要并且接口不关心这个，只要你提供了需要的方法的实现.

3.8. 接口继承(Interface Inheritance)

Interfaces can inherit from other interfaces, so you can declare an interface like so:
接口可以继承自其他接口， 所以你可以像这样申明一个接口

```
IFullName = interface(IName)
  ['{07E8CFE2-4C2B-41F4-8934-D9D3B5BE39BC}']
  function FullName: string;
end;
```

In this way, the child interface will require an implementation for all its declared methods as well

as those of its parent. Any class that implements IFullName will have to provide implementations for all of the methods in IFullName as well as IName.
这样，子接口需要它申明以及那些它父类的方法的实现. 任何实现了 IFullName 的类必须提供 IFullName 以及 IName 的所有方法的实现

Note that an implementation of the child interface is not an explicit implementation of the parent. An implementation must explicitly declare the interface that it wants to implement, even if it declares all the required members. That is, unless a class explicitly lists a given interface as part of its declaration, it cannot be used to implement that interface. Thus, you need to list both parent and child interfaces in an implementing declaration if you want use it as both the parent and child interface.
请注意子接口的实现并不是父接口的显式实现.需要实现的接口的必须被显式的声明在实现类中,即使该类声明了所有需要的成员.也就是说,除非一个类在声明中显式的列出了给出的接口,否则它不能用来实现那个接口.所以,如果你想要一起实现父接口与子接口,那么你需要在类声明中一起列出父接口与子接口.

Actually, the use of the phrase 'interface inheritance' is a bit misleading – it's not really true inheritance in that you can't override a method from a parent interface, and there is no polymorphic behavior that results. 'Interface augmentation' might be a better way to phrase it.
事实上,使用的短语"接口继承"是有误导性的 - 它并不是真正的继承,你无法覆盖来自父接口中的方法, 并且这导致了没有多态行为."接口增益"可以能更好的形容它.

3.9. 思考的其他事情(Other Things to Think About)

Here are a few things to think about when creating and dealing with interfaces:
当创建使用接口处理的时候这里有一些事情需要考虑

- As a general rule, you should declare interfaces in their own unit, preferably one interface per unit. Interfaces should be defined separately from any implementation or any other code. It's very tempting to declare an interface and a class that implements it in the same unit, but you should resist this temptation. Keep interfaces separate and completely decoupled from any particular implementation.
- 通常情况下，你可以自己的单元中申明接口，最好的做法是一个单元一个接口。接口定义应该跟实现和其他代码分开。在同一个单元非常容易申明一个接口和一个实现类，但是你应该抵制这种诱惑。保持接口独立的并且从任何特定实现完全解耦出来。

- It's a good idea to have interfaces be specific and to the point. If you have a large interface – one with many methods, you might consider breaking it down. This is known as the Interface Segregation Principle (http://en.wikipedia.org/wiki/Interface_segregation_principle) – the idea that interfaces

shouldn't have methods that the implementer doesn't need. There are times when it's okay to have a larger interface, but don't be afraid to have interfaces with just a handful of – or even one – methods, and don't be shy about having a class implement more than one interface.

- 让接口细化并且简明扼要是个好办法.如果你有一个包含很多方法的大接口，你可能需要考虑分解它了.这是大家所知的接口的分离原则 (http://en.wikipedia.org/wiki/Interface_segregation_principle). 思想是接口不应该包含实现者不需要的方法.有些时候有一个大接口没什么问题，但是不要害怕让接口只有少量甚至只有一个方法，也不用胆怯让类实现不止一个接口.

- Interfaces can be used as abstractions for your code, but not all interfaces are necessarily abstractions. If you have a "leaky" abstraction, then your interface isn't truly an abstraction. A leaky abstraction is one that allows implementation details to sneak through, thus dictating a certain kind of implementation. Say you had an interface called IDataSource and that interface had a method called GetConnectionString. Having that method appears to strongly imply the notion of a database that requires a connection string. An implementation of IDataSource might use a collection or a list, and so the notion of a connection string shouldn't be part of the implementation. In this case, the implementation detail of having a connection string has "leaked through" your attempt at abstracting the notion of a data source. Along those lines, if your interfaces are merely a reproduction of the public class you use to implement it, then it is very likely that you have a leaky abstraction. In this case, you should go back to the drawing board and refactor your code accordingly.
- 接口可以用于抽象你的代码，但是不是所有的接口都需要抽象.如果你有一个"泄漏"的抽象，那么你的接口就不是真正的抽象.一个抽象泄漏是指允许绕过实现细节，而指定某一种实现.假设你有一个叫做 IDataSource 的接口，那个接口有一个方法叫做 GetConnectionString.有那样的方法就很强烈的暗示了需要连接字符串的数据库概念.IDataSource 的实现可能使用一个集合或者列表，因此连接字符串的概念就不是实现的一部分.这样含有连接字符串的实现细节就泄漏了你企图抽象数据源的概念. 根据这些线索，如果你的接口只是你用于实现它的公共类的一个翻版,，那么这就很有可能你有了一个抽象泄漏. 这样的话，你就该回到画图板上相应的重构你的代码了

3.10. 关于 TInterfacedObject (About TInterfacedObject)

Okay, so there's a bit more to it than what I've discussed so far. In Delphi, interface references are reference counted. That is, the compiler keeps track of each reference to the implementing object, increasing the count when a reference is added and decreasing the count when a reference goes out of scope. The compiler automatically generates calls (AddRef and Release) that can be used to keep track of the number of times that an interface has been referenced. As we'll see, the "normal" way for interfaces to work is to use those compiler-generated calls to keep a count of those references. When the interface's reference count goes down to zero, then the compiler automatically frees the instance that is implementing that interface.

好了， 关于 Tinterfacedobject，除我迄今为止讨论过的，还有一些东西.在 delphi 里，接口引

用是引用计数的.也就是说，编译器会跟踪每一个对实现对象的引用，在引用增加时增加计数，离开作用域时减少计数.编译器自动产生调用(AddRef 和 Release)，这些调用可以用于追踪一个接口被引用的次数.如我们知道的，让接口合理的工作的方式是使用这些编译器产生的调用来保持这些引用的计数.当接口的引用计数减少到 0 的时候，编译器会自动销毁掉实现该接口的(对象)实例.

In this way, Delphi can do automatic memory management for interfaces. This means that once you create a class that is referenced by an interface, you can't call Free on the instance – unless you happen to have a member called Free on the interface itself. Instead, the compiler will write all the code to track the references and free the object when it is no longer referenced.
这样的话,Delphi 就可以为接口做内存自动管理. 这意味着一旦你创建了一个被接口引用的类，你可以不用调用该实例的 free，除非很偶然接口本身有一个叫 Free 的成员。反倒是编译器会产生所有的代码来追踪引用，并在它不再被引用的时候释放掉该对象.

Thus, the compiler needs a way to do all that reference tracking. It also needs a way to figure out which interface is what (remember the GUID from above?). In order to do that, all classes that implement an interface need to declare and implement three methods:
因此，编译器需要一种途径来实现所有引用的跟踪。也需要一种途径去找出哪个接口是什么(记得之前的 GUID 吗?). 为了这些，所有那些实现一个接口的类需要申明和实现下面三个方法：

```
function _AddRef: Integer; stdcall;

function _Release: Integer; stdcall;

function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
```

These three methods are part of the root interface for all interfaces, IInterface. The first two methods are called automatically by the compiler each time it sees that a reference is used (_AddRef) and goes out of scope (_Release). The QueryInterface method is used to determine if the class implements a given interface. All three of these methods are required by the base interface IInterface and thus are required by all interfaces.
这三个方法是所有接口的根接口(IInterface)中的一部分,头两个方法是每当遇到引用(_Addref)和离开作用域时被编译器自动调用的。QueryInterface 是用来判断类是否实现了给定的接口. 所有这 3 个方法都是基础接口 IInterface 所必须的，因此所有的接口也都需要它们.在这里我不打算继续深入了，但到现在你至少可以理解 TinterfacedObject 的声明(和实现)了.

```
TInterfacedObject = class(TObject, IInterface)
protected
  FRefCount: Integer;
  function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
public
```

```
  procedure AfterConstruction; override;
  procedure BeforeDestruction; override;
  class function NewInstance: TObject; override;
  property RefCount: Integer read FRefCount;
end;

....

function TInterfacedObject._AddRef: Integer;
begin
  Result := InterlockedIncrement(FRefCount);
end;

function TInterfacedObject._Release: Integer;
begin
  Result := InterlockedDecrement(FRefCount);
  if Result = 0 then
    Destroy;
end;
```

You can see the entire declaration of TInterfacedObject in the System.pas unit. It basically does all the housekeeping of managing the reference count of the implementing object, and destroying it when the count gets to zero. It should be noted, too, that referencing an interface's reference count is thread-safe because the _AddRef and _Release methods use Interlocked operations. This means that you can safely access interface implementations across multiple threads. Note that this doesn't mean that all your code is thread-safe – just that the referencing of interfaces between threads is thread-safe.

你可以在 System.pas 单元看到 TInterfacedObject 的整个声明它基本上做了所有管理实现对象引用计数的家务活，并且在计数变成 0 的时候销毁掉该对象.这里也应该注意的是，涉及一个接口的引用计数是线程安全的，因为_AddRef 和_Release 方法使用了 Interlocked 系列操作.这意味着你可以很安全跨越线程来访问接口的实现.注意，这不意味着你所有的代码是线程安全的，只是线程间接口的引用是线程安全的。

Some further things to note about using TInterfacedObject:

更多关于使用 TInterfacedObject 的东西：

- If you want reference counting for your interfaces, you need to either descend your implementing classes from TInterfacedObject, or declare similar functionality in your class.The majority of your interface implementing classes will probably descend directly

from TInterfacedObject.

- 如果你想为你的接口实现引用计数，你需要从 TInterfacedObject 继承你的实现类或者在你的 class 中声明相似的功能 大多数接口的实现类可能从 TInterfacedObject 直接继承。

- The RTL also provides two additional, but less commonly used, classes that implement the three “magic” interface functions. TAggregatedObject and TContainedObject are designed to be aggregated or contained by an outer, controlling class. If you have an implementing class that will create and control internal interfaces, you can use these classes as base classes and they will be reference counted and managed by their controlling, external classes.
- RTL 也提供了 2 个额外的但不是很常用的类，它们实现了这 3 个"神奇的"接口方法，TAggregatedObject 和 TContainedObject 是为了被外部控制类聚合和包含而设计的，如果你有一个实现类创建和控制内部的接口，你就可以使用这些类来作为基类 ，它们是有引用计数的，并且由外部控制类来管理的。
- You should never have a need to call the three “magic” methods of TInterfacedObject. If you feel the need to, you almost certainly are doing something you shouldn’t be. Pay no attention to what the IDE’s code completion tells you – don’t call them or mess with them in any way.
- 你根本就没有必要去调用 TInterfacedObject 里的 3 个"神奇的"方法，如果你觉得有必要去调用的话，那你无疑是在做一些本不该做的事，不用在意 IDE 的代码自动完成给你的提示， 无论如何也不要调用它们或者用它们把事情弄乱了。
- With regard to the previous note, it is interesting that you can call the methods on TInterfacedObject in your code despite them being protected or even private in scope. Remember, they are part of the interface, and the interface doesn’t have the notion of private, protected, or public.
- 关于前面的提醒，有意思的是你可以在代码中调用 TInterfacedObject 上的方法，尽管它们是被保护甚至是在私有的作用域中。 记住了，它们是 Interface 的一部分，并且接口没有私有,保护和公开的概念。

Reference counting of interfaces also means that you should never, ever mix interface references with “real” object references to an implementing class. Never. Just don’t do it – it can lead to the reference counting system being by-passed, leaving you with stranded references to objects out on the heap.

接口的引用计数也意味着永远不要混用接口引用和实现类的"实际"对象引用. 永远不要，就不要这样干，这样可能会导致引用计数系统被绕过，留下的是堆上被搁置的对象引用.(内存泄漏)。

For instance, the following code mixes object and interface references:

举个例子，下面的代码混用了对象和接口引用：

```
procedure DoNotMixInterfaceAndObjectReferences;
var
  ObjectReference: TInterfacedObject;
  InterfaceReference: IInterface;
begin
```

```
  ObjectReference := TInterfacedObject.Create();
  // RefCount is 0
  InterfaceReference := ObjectReference;
  // RefCount is 1
  InterfaceReference := nil;
  // this causes RefCount = 0, thereby destroying the underlying object
  // but ObjectReference is still in scope and might be used.
end;
```

Again, don't do this as it is very easy to create an access violation. Keep your interface references and your object references completely away from each other and don't mix them.
再说一次，不要这样做，因为这样很容易导致内存违规访问，保持你的接口引用和你的对象引用彼此分离，而不要混在一起。

3.11. 如何使用接口(How to Actually Use Interfaces)

Okay, so now you know about declaring and implementing interfaces. But how do you actually use them? It's quite straight forward. You simply declare a variable of the interface type you want to create, and then assign to it an instance of an object that implements that interface. A simple example would look like this:
好了，现在你已经了解了接口的声明和实现．但是你该如何使用他们呢？ 这很直接，你只需声明一个你想要创建的接口类型变量，然后将它赋值为实现该接口的对象实例。一个简单的例子可能是这样的：

```
var
  NamedPerson: IName;
begin
  NamedPerson:= TPerson.Create;
  WriteLn(NamedPerson.FirstName, ' ', NamedPerson.LastName, ' is a person.
');
end;
```

Here are some things to note about the simple example above:
关于上面简单例子的一些说明

First, TPerson descends from TInterfacedObject, and therefore it meets the minimum requirements for implementing an interface – that is, _AddRef, _Release, and QueryInterface.
首先，TPerson 是从 TInterfacedObject 继承的，所以它满足了实现接口的最小需求,也就是 _AddRef, _Release, and QueryInterface。

NamedPerson is defined as IName. This means that you can only call the methods of IName on NamedPerson even though TPerson may have more methods than IName does.
NamedPerson 是被定义为 IName 类型，也就意味着即使 TPerson 中的方法比 IName 有的还多,你也只能在 NamedPerson 上调用 IName 中的方法。

Because interfaces are reference counted, there is no need for a try...finally block with a call to Free. Sure, you create an object, but the compiler will keep track of all references to the instance and free it when there are no references left. The construct may look a little strange to you at first – I know that I was very used to seeing the try...finally block – but the resulting code is simpler and easier to manage.
由于接口是引用计数的，所以没有必要使用 Try..finally 块来调用 free，的确，你创建了一个对象，但是编译器会跟踪该实例的所有引用并且在没有引用存在的时候释放它， 我习惯看到 try..finally 块，我知道这个概念对于你来说一开始可能看上有点奇怪,但是这样的写出的代码更简单也更容易维护。

Interfaces can be used anywhere that regular variables are used. You can pass them as parameters and declare them as fields, properties and local variables. Indeed, you should take advantage of interfaces everywhere you can.
任何常规变量使用的地方，接口都可以使用，你可以把他当作参数传递，也可以把他们声明为字段，属性和局部变量. 的确，你应该在任何能用的地方利用好接口。

针对抽象编程(Coding Against Abstractions)

Interfaces allow you do something that is critical to writing good code: They allow you to program to an abstraction. A pure abstraction, in fact. Why do you want to program to abstractions and not implementations? The simple answer is this: an interface is the smallest, thinnest, and least complicated thing you can couple your code to. As discussed above, loosely coupled code is good.
接口是编写好代码的关键：允许你面向抽象编程. 实际上是一个纯抽象。为什么你想面对抽象而不是实现呢？简单的说：接口是最小的，最薄的，最少的复杂东西你可以耦合到你的代码. 就像上面讨论的那样，低耦合的代码是不错的。

Now, if your code is completely decoupled, then it can’t do anything at all. Code has to be coupled to something in order to provide any useful functionality, so “perfectly decoupled” code is going a touch too far – what you really want is loosely coupled code. Very loosely coupled code. You want the coupling of your code to be as loose as humanly possible. If you never couple to anything but interfaces, then that is as loosely coupled as you can get.
现在，如果你的代码是完全解耦的，那么根本不能做任何事情。为了提供任何有用的功能，代码必须被耦合，所以“完全解耦"的代码是有点过头 - 你真正想要的是松散耦合的代码。非常松散耦合的代码。 你想尽可能的松散耦合你的代码。如果你除了耦合接口不耦合其他任何东西， 那么这是你能得到的松散耦合。

Ultimately, this is the bottom line reason why you should use interfaces in Delphi: They provide a

very thin – but very powerful – abstraction to your code. Much of the rest of this book is really an expansion on that one simple but fundamental idea. Think of your code as Lego blocks, with pins on one side and holes on the other side. With this loosely coupled interface you can join any block to every other block and create anything you want. It would be very limiting if the red blocks had a different hole size than the green blocks' pins, no?

最终，这是为什么你应该在 Delphi 中使用接口的基本理由：他们能在你的代码中提供一个非常简短但是非常强大的抽象。本书中剩下的大多数东西是基于简单但又基础的概念的延伸.把你的代码想象成是乐高积木，一边是插脚另外一边是插孔，用这种松散耦合的接口你就可以把任意的一块积木和任何其他的积木连接在一起来创造你想要的东西。它将要成为如果红色块有一个与绿色块插脚相比较的不同孔径的恰当限制。不是吗？

As I've said before, and I'll say again: A good developer codes against abstractions, and not implementations. Interfaces are a great way to create abstractions. If you want a thorough discussion on why this is a good idea, I suggest reading Erich Gamma on the topic2 – but I'll talk a bit about it here.

正如我以前说的,并且我要再说一次:好的程序员针对抽象而不是针对实现来编程.接口是一个伟大的方法去创造抽象.如果你想要彻底的讨论为什么这是一个好主意,我建议你去读 Erich Gamma 的话题(http://www.artima.com/lejava/articles/designprinciples.html) - 但我会在这里讲解一部分。

可拔插的实现(Pluggable Implementations)

If you program against abstractions, you can't couple yourself to a specific implementation. Interfaces allow you to make the coupling between your classes very loose. Classes should be developed and tested in isolation with few or no external dependencies. But they almost certainly have to depend on something. And certainly once you have a well-designed class library created, you need to piece it together to create the system you need to build – just like a child can build almost anything he or she wants out of Lego blocks. In the end, an interface is the lightest and thinnest thing that a class can depend on. (I've likened coupling to interfaces as "grasping at smoke".) So, if you program primarily with interfaces, you can't help but create very loosely coupled code. And we all know that loosely coupled code is good. So interfaces help produce good code.

如果你要针对抽象编程,你就不能耦合特定的实现.接口能让你的类之间的耦合非常松散.类能被独立的开发与测试而少依赖甚至不依赖外部.但是它们几乎肯定是依赖于一些东西.并且一旦你创建了一个优良设计的类库,你就需要把组合起来去创造你需要构建的系统-就像一个孩子用乐高积木几乎能创建任何他想要的东西.最后,接口是类能依赖的最轻最瘦的东西.(我把接口耦合比做"抓住烟".)所以如果你主要用接口来编程,你会自然而然的做到松耦合.并且我们知道松耦合的代码是很好的.所以接口帮助我们创作出好代码.

But there's more – interfaces also let you alter implementations, even at runtime. Because you are dealing with an interface, and not an implementation, you can very easily pick and choose what implementation you want when you want. For instance, you can write code like this:

这里还有一些 - 接口也让你改变实现，甚至在运行时也可以。因为你是用接口来处理的，而不是一个实现，你可以非常轻易的选择你想要的实现。例如，你可以想这样的写代码：

```
procedure EncryptSomething(aSuperSecretStuff:TBuffer; const aIWantToBeSafe:
 Boolean);
var
  Encryptor: IEncrypt;
begin
  if aIWantToBeSafe then
  begin
    Encryptor := TSuperDuperPowerfulEncryption.Create;
  end else
  begin
    Encryptor := TWhoCaresHowSafe.Create;
  end;
  Encryptor.Encrypt(aSuperSecretStuff);
end;
```

This is a silly example, but you can see how this can be very useful – you can have a single interface and select the proper implementation at runtime as needed. In fact, I'd say that if you can't choose your implementation at runtime, then your code is too tightly coupled.

这是一个糊涂的的例子，但是你可以看到这个是非常有用的 - 你可以你可以有一个单一个的接口并且在运行的时候根据需要选择一个合适的实现。实际上，我想说的是:如果你不能在运行时选择实现，那么你的代码太紧密耦合了。

A more practical example might be an ICreditCardProcessor interface where you instantiate different credit card implementations based on the choice made by your customer. Because you aren't tied to a specific implementation, you can use a single ICreditCard interface while making it easy to alter which implementation gets used. If you want to add a new credit card processor,it's as easy as implementing a new class and adding it to the means you use to choose the credit card processor. No changing the main code that handles credit cards – it doesn't matter what implementation you use.

一个更实用性的例子可能是一个 ICreditCardProcessor 接口,你用它来实现基于你客户选择的不同的信用卡的实例.因为你没有被栓在一个特定的实现上,你可以使用单一的 ICreditCard 接口以使你在使用时容易的更换实现.如果你想要增加一个新的信用卡处理机,你只要简单的实现一个新的类并且增加到信用卡处理机的选择上.用来处理信用卡的主代码并没有改变-它不在意你使用了什么实现。

And of course, you might think that your implementation of your interface is great, but it's entirely possible that a better one will come along, and you want to be able to easily plug that new implementation in without having to radically change your code. Interfaces make that quite possible and easy. A good rule of thumb is to "*Always develop against interfaces and code as if a radically better implementation is just around the corner.*" (That sounds too good to be my original quote. I can't find it's origin, and I did try. If the quote is yours, please let me know and I'll be happy to provide proper attribution.)

当然,你可能认为你接口的实现是很棒的,但完全有可能会有更好的改进,而你想要很容易的插入新的实现并且无需不得不从根本上修改你的代码.接口让这成为了可能并且非常容易.一个好的经验准则是"*经常针对接口编程,则在根本上更好的实现就指日可待了.(https://twitter.com/NickHodges/status/146018445002145792)*"(这听上去好像与我的原始引用不尽相同.我没能找到原始出处.如果这话是引用你的,请告知我,我很乐于给它加上正确的归属.)

模块之间通信(Intermodule Communications)

But wait, there is more! Interfaces are good for inter-module communications. Say you have a large system with different teams working on different major modules. Those teams are responsible for providing functionality in their own modules, and thus they are responsible for the integrity and quality of the code in their modules. Let's say you are on the Widget team, and you need the Sprocket team to do something for you. You go to the Sprocket guys and say "Hey, I need to add a few things in the Sprocket code so that we can do some Sprocket-type stuff in our Widget." The Sprocket guys are going to laugh at you – like they are going to let you poke around in their carefully crafted system!

不要着急，我还要说：接口对于模块间通讯是个好主意。假设你有一个由不同团队不同模块构成的大型系统，它可以提供在模块间很好的功能响应及处理模块代码的完整性和质量。假设你在一个部件团队，你需要齿轮团队来为你做一些事情。你找到齿轮团队的同伴说："嗨，我需要在齿轮代码中添加一些东西，以便于我们可以在我们部件中处理一些齿轮类型的东西。"齿轮团队将笑话你-就像他们允许你在他们精心设计的系统中瞎折腾。

No, instead, they will likely ask you what you need, build the functionality, and hand you some code with an interface and a factory for instantiating an implementation of that interface. They aren't going to let you anywhere near their code – but they are happy to let you have an interface into that code. You get what you want – some Sprocket functionality – and they don't have to expose anything more than an interface to you.

不,相反的,他们可能会问你需要什么,并构建功能,然后传给你一些代码,包含接口和能实例化接口实现的工厂.他们不会让你从任何地方接近他们的代码-但是他们很乐于让你拥有一个连接那段代码的接口.你得到了你想要的东西-一些扣链齿轮的功能-并且他们除了接口没有暴露任何给你。

And later, if they completely re-architect their code, what do you care? Your interface still works,even if they completely change the internal implementation. That's a sound way to develop, all made possible because of interfaces。

将来,如果他们完全重构他们的代码,你需要关心什么?你的接口依然工作,即使他们完全改变了内部实现. 那是一个好的开发方式, 因为接口一切变得可能。

可测试的代码(Testable Code)

It doesn't end there – interfaces make your code testable. As noted above, because you are using interfaces you can readily substitute any implementation you want.What if you are testing and

you don’t want to connect to the production database? You can simply provide a fake implementation for your database connection interface – one that only pretends to be the database and that returns canned data – and now you can test your code in isolation without actually connecting to a database. I’ll discuss this more in the chapter on unit testing.

还没结束-接口接口使你的代码可测试。就像上面提到的，因为你是使用接口，你可以轻易的替换你想要的任何实现。如果你测试的不想连接产品数据库那怎么办呢？你可以简单的为你的数据库连接提供一个虚拟的实现 - 一个只是假装的数据库并且返回存储数据 - 然后你可以在没有真实连接数据库的环境下测试你的代码。我们将在单元测试章节进行更多的讨论。

模式(Patterns)

Finally, interfaces make it easy to implement design patterns and do things like Dependency Injection. Most of the new patterns and practices – including Dependency Injection frameworks – are enabled because of the power and flexibility of interfaces. Development patterns and architectures such as Model-View-Controller (MVC) and Model-View-ViewModel (MVVM) are much easier to implement and use when designed with interfaces.

最后，接口便于实现设计模式,并像依赖注入一样完成它.大多数新的模式和实践-包括依赖注入框架-因为接口的功能和灵活性才得以使用.接口令开发模式和架构的比如 MVC(Model-View-Controller: 模型-视图-控制器)和 MVVM(Model-View-ViewModel: 模型视图查看模型)能更容易的实现和使用.。

If you choose not to embrace interfaces, then you are locking yourself out of new and effective programming frameworks and techniques. I’ll be covering Dependency Injection in depth in a later chapter, and you’ll see then that without interfaces, Dependency Injection would not be as easy as it could be.

如果你选择不去拥抱接口,那么你将会困住自己而无缘新的有效的编程框架和技术.我将会在后面的章节中深度的使用依赖注入,而你将会明白离开了接口,依赖注入将不那么易用.。

Still not convinced? I’ll put it another way: all the cool kids are doing interfaces, and you want to be part of the cool kid group, right?

还不信服?那我换种方式来说服你: 所有的酷程序员都使用接口,而你也想成为酷程序员中的一员,对吧?