LINQPad

Succinctly®

by José Roberto Olivas Mendoza

Syncfusion | Technology Resource Portal

LINQPad Succinctly

By

José Roberto Olivas Mendoza

Foreword by Daniel Jebaraj

Syncfusion

Deliver innovation with ease

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Morgan Weston, social media marketing manager, Syncfusion, Inc.
Proofreader: John Elderkin

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn’t everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors’ tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

I'm an IT businesses entrepreneur, a software developer, and a huge technology fan. My company went to market in 1990, focused mostly on custom software development. We started with COBOL as our main programming language, and since then we've evolved up to .NET and Microsoft Office technologies. Through these years, some of my core activities have included research about cutting-edge technologies and searching for new tools that can help us automate some or all processes regarding our products' development lifecycle.

In 2010, electronic invoicing was set as a requirement for business transactions in Mexico, which forced us to upgrade all of our business-related products within a six-month window. In those days, the effort of our development team was huge. Fortunately, we delivered all of our products right on time.

We didn't expect any further changes regarding electronic invoicing, but we were wrong. Two years later, in 2012, the Mexican Tax Authority implemented a lot of modifications regarding invoicing. The deadline for these changes gave us only four months to upgrade all of our products.

Since then, new requirements about electronic invoicing have been published by the Mexican Tax Authority every year. This has made it almost impossible for us to comply with them on time without a huge effort from our company.

LINQPad went into action in 2015 as a solution for testing our .NET reusable libraries related to electronic invoicing. As a result, testing for those libraries was done in a fraction of the time compared with our previous efforts. The delivery of our products can now be made with less time and effort.

As a great lover of technology, I've been exploring the product since the first day we started using it, and it has pleased me immensely. Because it supports LINQ, our company has an easy way to learn how to implement it in our .NET applications without the need to build any kind of project for doing so. Also, testing became instantaneous. This helped us improve our Scrum practices, allowing us to include more features for a product in a single sprint. This increases maturity level for each of our products in less time.

For these reasons, LINQPad will definitely remain a part of our toolbox for many years.

Who Is This Book For?

This book is written primarily for those IT professionals who work with the .NET platform because LINQPad is a utility targeted for these kinds of developers. It's best that you have C# programming language skills and some basic knowledge about LINQ.

The book starts with a detailed overview of LINQPad and its features. The second chapter covers the LINQPad installation process, including prerequisites and where to get the free edition of LINQPad. The rest of the book explains how to get the most out of LINQPad, starting with some basics. Themes like querying databases and code prototyping are explained in separate chapters.

This book will focus on the free edition of LINQPad4 and the C# programming language. A few LINQPad5 samples will be explained when their features are unique for this version. Also, Visual Studio 2015 will be employed to write some of the code examples, with the .NET framework 4.5 as the target framework for assemblies building. Also, Microsoft SQL Server Management Studio will be used for databases treatment. All samples described in this book can be downloaded here.

I hope that all readers enjoy this material and that it helps you use LINQPad as a tool for improving your development lifecycle in order to deliver applications in less time.

Chapter 1 Introduction

What is LINQPad?

LINQPad is a software utility developed by Joseph Albahari, and it's intended to be used as a part of .NET application development. It is used for interactively querying SQL databases using LINQ, as well as interactively writing C# code without the need for an IDE (such as Visual Studio). This feature expands its use as a "test tool" where C# code can be quickly prototyped outside the IDE. It also supports the VB.NET, SQL, and F# languages. LINQPad supports the following technologies:

- Entity Framework
- LINQ to Objects
- LINQ to SQL
- LINQ to XML

The product starts with a freemium version, which lacks an autocomplete feature for the C# language. To get this feature, the user should purchase a commercial license.

Regardless of the type of license, either free or commercial, there are three versions of the product:

- LINQPad 2.x for .NET framework 3.5
- LINQPad 4 for .NET framework 4.0/4.5. Current stable release: 4.58.0
- LINQPad 5 for .NET framework 4.6: Current stable release: 5.05.02

Note: LINQPad 5 can run side-by-side with LINQPad 4.

LINQPad editions

There are four editions for the LINQPad product: Free, Pro, Developer, and Premium. Except for the first one, the user will need to pay a license fee for using the product. Depending on which edition is purchased, LINQPad unlocks a series of additional features to those included in the free edition. The user installs the free edition of LINQPad first, then buys a license for upgrading to the desired edition beyond the free one. This license applies for both LINQPad 4 and LINQPad 5, so the user doesn't need to pay a fee for each version.

Note: According to the product's website, the users who purchased a license of LINQPad 4 prior January 1, 2015 must purchase an upgrade to LINQPad 5.

The following table summarizes all features available for LINQPad, in accordance with the edition being used.

Table 1: LINQPad Editions Comparison Table

	Edition			
Features	**Free**	**Pro**	**Developer**	**Premium**
Run LINQ queries or any C#/VB/F# expression/program	Yes	Yes	Yes	Yes
Results output to rich text or data grids	Yes	Yes	Yes	Yes
Automatic translation to SQL, fluent-lambda, and IL	Yes	Yes	Yes	Yes
Reference user's own assemblies, plus the entire .NET Framework	Yes	Yes	Yes	Yes
Use of custom extensions, visualizers, and data source providers	Yes	Yes	Yes	Yes
Full C#/F#/VB autocompletion and tooltips		Yes	Yes	Yes
Smart tags for importing additional namespaces/references (C#/VB)		Yes	Yes	Yes
Code outlining and auto-formatting (C#/VB)		Yes	Yes	Yes
Rename symbol, go to definition and find references (C#/VB)		Yes	Yes	Yes
Built-in and custom code snippets			Yes	Yes
Cross-database querying for SQL Server			Yes	Yes
Directly edit SQL data in grids and save changes back to database			Yes	Yes

	Edition			
Features	**Free**	**Pro**	**Developer**	**Premium**
Full NuGet integration			Yes	Yes
Integrated debugger, with call stack, threads, local variables/watch windows, breakpoints and single-stepping				Yes

LINQPad’s main specifications

LINQPad offers a series of benefits to the user, such as:

- Full support for C# 6.0 and VB 14 in LINQPad 5.
- Reporting of execution time in its status bar, allowing performance testing.
- Cloning of code snippets with a key combination (Ctrl+Shift+C), allowing their execution side-by-side.
- Custom extensions writing, which allows the user to code methods and classes to make them available for all LINQPad queries.
- Custom assemblies referencing.
- Building typed data contexts on the fly, including drivers and formatters for SQL Server, SQL Azure, SQLCE, Oracle, SQLite, and MySQL.
- Object rendering with the embedded Dump method.
- Access control for the Dump method in user’s custom types.
- Full command line experience with the lprun.exe utility.
- Support for writing custom visualizers.
- Support for presentation mode, in order to allow cordless presenter devices.

How LINQPad works

LINQPad is a client/server application that works in an unusual way. While most client/server applications have many clients and one server, with LINQPad, there is one client and there are many servers. In this case, the LINQPad user interface is the client as expected, and for each query that is going to be executed, LINQPad creates a separate server, which has a class that runs in its own process. This process runs the query in isolation. The isolation prevents queries from interfering with each other or with the user interface, and it allows LINQPad to safely cancel a query without affecting other application domains.

LINQPad uses a customized communications layer that runs atop Windows shared memory, providing a faster, more reliable, and fully asynchronous communications layer.

Microsoft Roslyn libraries are used to compile queries in LINQPad 5. Regarding database objects, LINQPad needs a typed DataContext for backing any database object referenced in queries, because C# and VB are statically typed. LINQPad builds these typed DataContexts on the fly by using Reflection.Emit instead of generating and compiling source code, for performance reasons. Also, it uses LINQ to SQL rather than Entity Framework because LINQ to SQL is lighter and faster in building meta models when instantiated.

LINQPad's emission engine itself runs in a separate application domain, because Reflection.Emit must instantiate types in memory whose assemblies cannot be unloaded. This domain is regularly refreshed to avoid an ever-increasing memory footprint.

So much of LINQPad's work goes on in the background, such as querying database schemas, emitting typed DataContexts, and compiling and executing queries. For this reason, every potentially time-intensive feature operates asynchronously to maintain a responsive user interface.

LINQPad's `Dump` method (which will be discussed later) feeds the output into an XHTML stream. This stream is displayed using an embedded web browser (the user can notice this by right-clicking a query result and choosing **View Source**). LINQ to XML is used to transform the output into XHTML as one LINQ query. The deferred expansion of results and fetching of additional data when the user clicks on hyperlinks is made via JavaScript.

LINQPad uses third-party libraries from several software publishers. These include Actipro's SyntaxEditor, the SharpDevelop project, Microsoft Roslyn, and NRefactory. The libraries from these publishers are included as embedded resources in LINQPad, so it ships as a single executable.

All updates for LINQPad are downloaded into the application's data folder. Then, LINQPad checks that the new assembly has the same strong name. If so, LINQPad forwards to that executable, which then writes itself back to the original one. This operation can fail, depending on the permissions given to the folder where LINQPad was downloaded. Every time it's started, LINQPad always checks to see if there's a later, signed version of itself in the application's data folder.

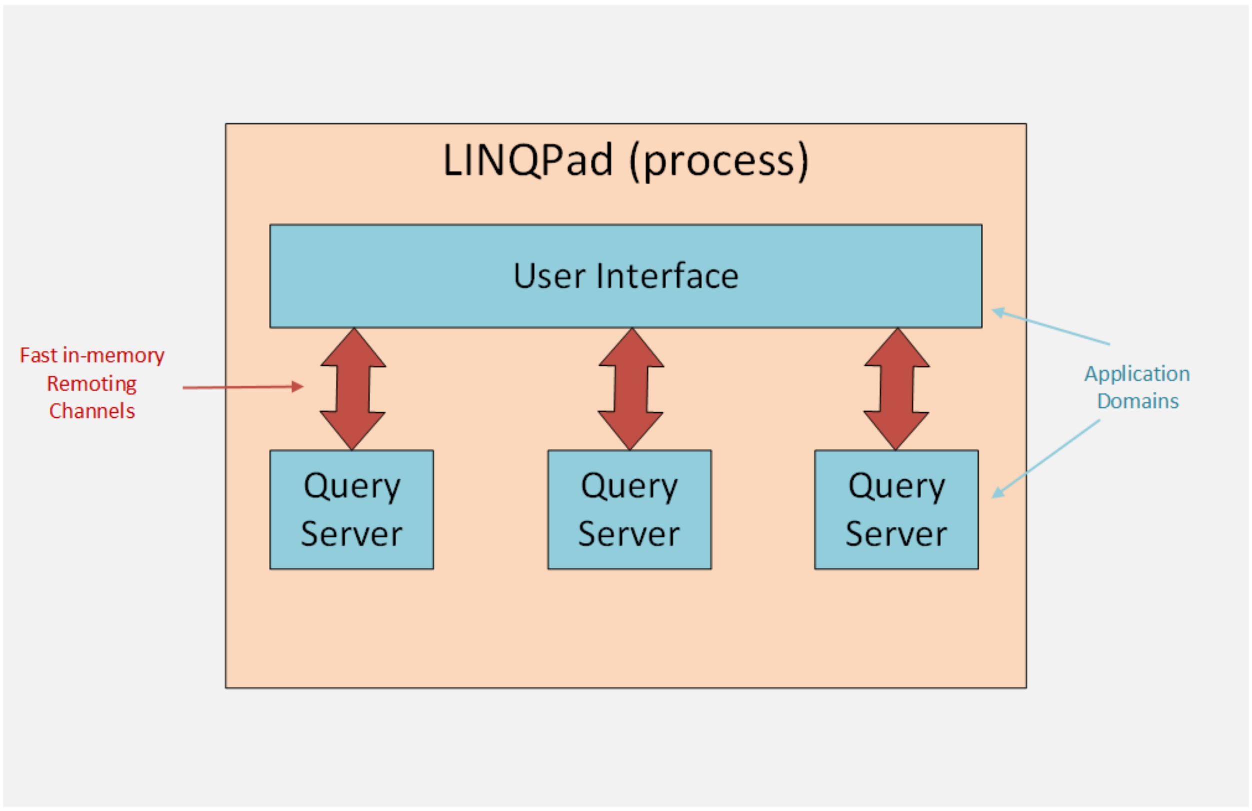


Figure 1: Context diagram about how LINQPad works

Chapter summary

LINQPad is a software utility developed by Joseph Albahari, and it's intended to be used as a part of .NET application development. It is used to interactively query SQL databases using LINQ, as well as to interactively write C# code without the need for an IDE, such as Visual Studio.

LINQPad supports VB.NET, SQL, and F# languages. Also, LINQPad supports Entity Framework, LINQ to Objects, LINQ to SQL, and LINQ to XML dialects.

The product starts with a freemium version, which lacks several features, such as autocomplete for the C# language. Regardless of the type of license, free or commercial, there are three versions for the product: LINQPad 2.x (for .NET Framework 3.5), LINQPad 4 (for .NET Framework 4.0/4.5), and LINQPad 5 (for .NET Framework 4.6). Also, there are four editions for each one of these versions: Free, Pro, Developer, and Premium.

Depending on which version is purchased, LINQPad automatically enables extra features above all those included in the free version. Some of these features include Full C#/F·/VB autocompletion and tooltips, code outlining and formatting for C#/VB, built-in and custom code snippets, and full NuGet integration.

LINQPad is a client/server application, which works as one client and many servers. In this case, the LINQPad user interface is the client, as expected, and for each query that is going to be executed, LINQPad creates a separate server. For communications between client and servers, LINQPad uses a customized communications layer that runs atop Windows shared memory.

Microsoft Roslyn libraries are used to compile queries in LINQPad 5. All database objects are backed by using typed DataContexts, because C# and VB are statically typed. All DataContexts are built on the fly by using Reflection.Emit. LINQPad uses LINQ to SQL rather than Entity Framework. All background work (querying database schemas, compiling and executing queries) operates asynchronously, in order to maintain a responsive user interface.

LINQPad uses third-party libraries from several software publishers within the UI. These includes Actipro’s SyntaxEditor, the SharpDevelop project, Microsoft’s Roslyn, and NRefactory. The libraries from these are included as embedded resources in LINQPad, so it ships as a single executable.

All updates for LINQPad are downloaded into the application’s data folder. Every time it’s started, LINQPad checks to see if there’s a later, signed version of itself in the application’s data folder.

Chapter 2 Installing LINQPad

Getting started

LINQPad can be installed on any computer running Windows. The following requirements are suitable for the product.

Hardware:

- Intel Dual Core 1 GHZ processor or above
- 2 GB RAM (4 GB recommended)
- 500 GB Hard disk drive
- VGA Monitor (1024x768 resolution minimum)
- Keyboard and mouse
- Internet connection (for downloading the product)

Software:

- Windows 7 SP1 or above
- .NET Framework 3.5/4.0/4.5/4.6
- Visual Studio 2010 or above (for developing purposes)

Getting LINQPad

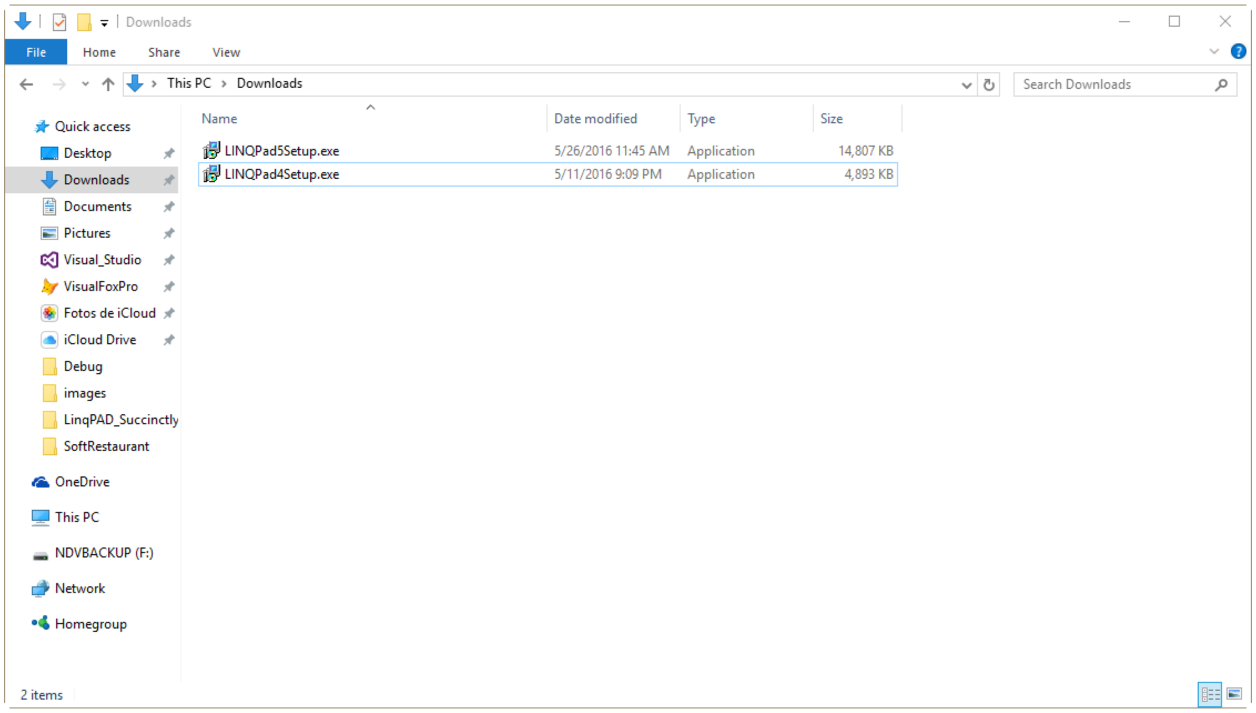
LINQPad can be downloaded for free from the following locations:

- LINQPad 5
- LINQPad 4

LINQPad installation process

As previously mentioned in the “What is LINQPad?” section of Chapter 1, LINQPad 4 and LINQPad 5 can both be installed in the same computer, and they can work side-by-side. For purposes of this section, the LINQPad 4 installation process will be detailed, but the same process applies to LINQPad 5.

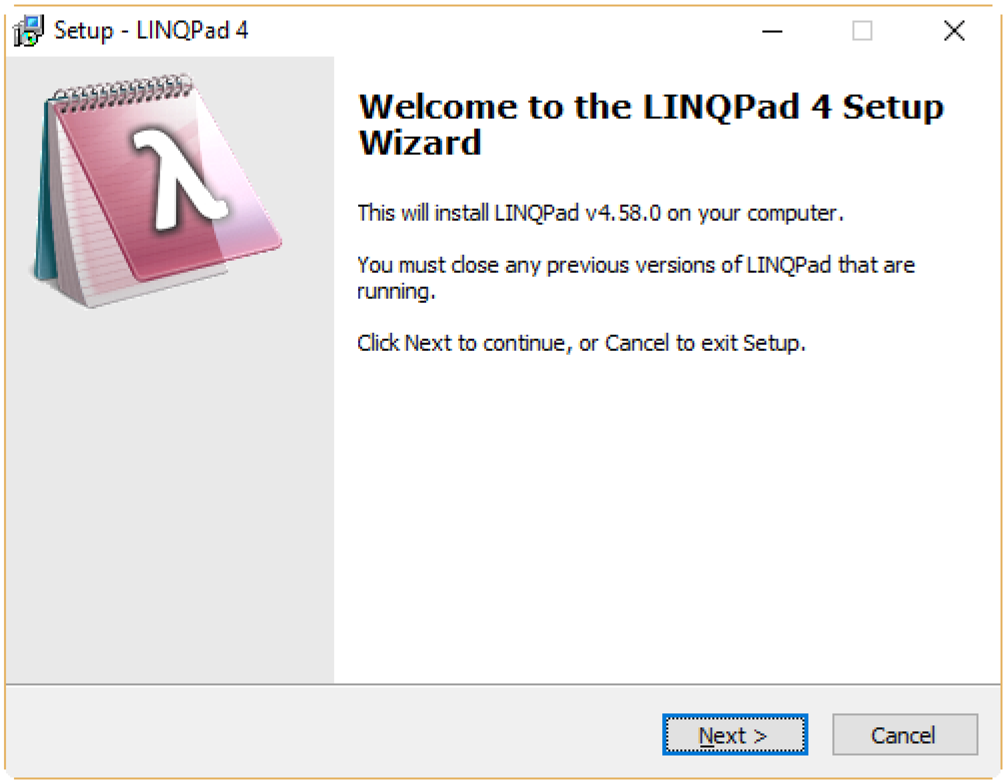
To start LINQPad installation, download LINQPad from one of URLs listed in the previous section. Then, double-click on the **LINQPad4Setup.exe** file (or **LINQPad5Setup.exe** for LINQPad5) located in the **Downloads** folder.



Name	Date modified	Type	Size
LINQPad5Setup.exe	5/26/2016 11:45 AM	Application	14,807 KB
LINQPad4Setup.exe	5/11/2016 9:09 PM	Application	4,893 KB

Figure 2: LINQPad setup files in Downloads folder

The first dialog box that appears is shown in the following figure.



Welcome to the LINQPad 4 Setup Wizard

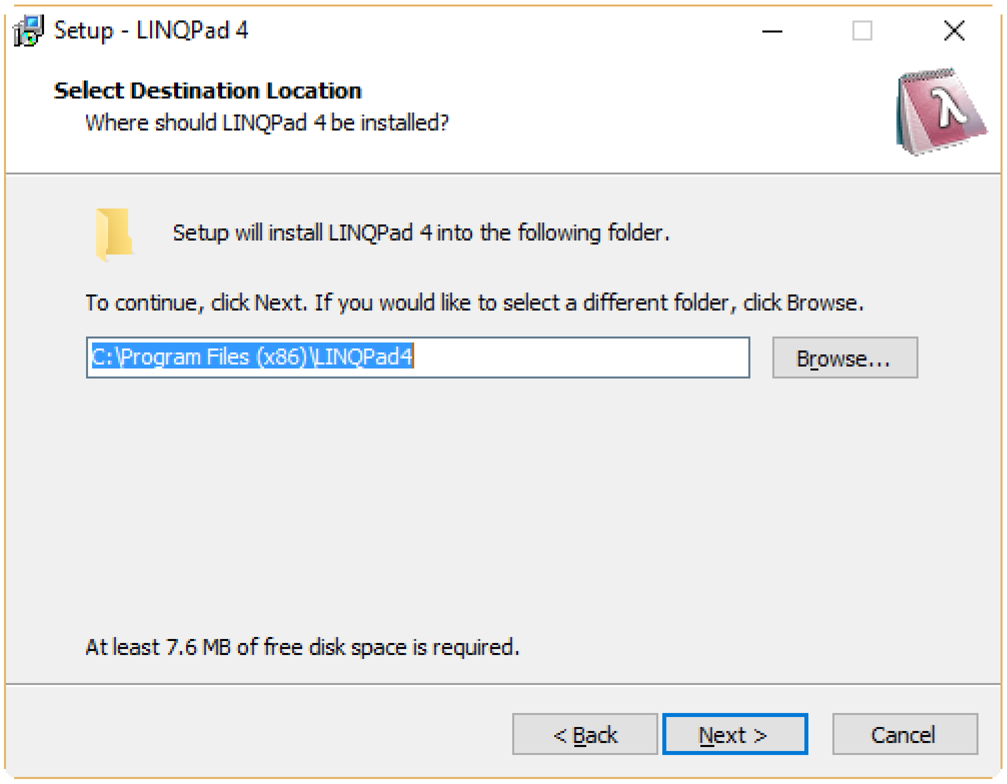
This will install LINQPad v4.58.0 on your computer.

You must close any previous versions of LINQPad that are running.

Click Next to continue, or Cancel to exit Setup.

Figure 3: LINQPad installation's first dialog box

Click **Next** to continue, and the Select Destination Location dialog box appears.


Setup - LINQPad 4

Select Destination Location

Where should LINQPad 4 be installed?

Setup will install LINQPad 4 into the following folder.

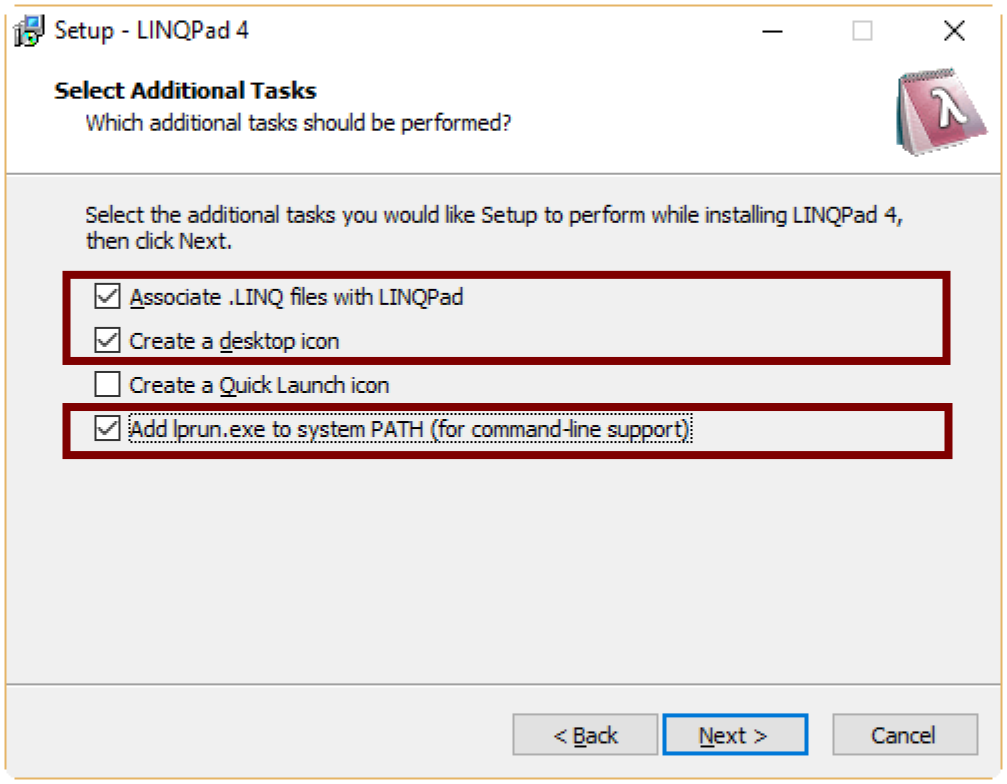
To continue, click Next. If you would like to select a different folder, click Browse.

C:\Program Files (x86)\LINQPad4

At least 7.6 MB of free disk space is required.

Figure 4: Select Destination Location dialog box

It’s recommended that you use the default location provided by the installation program. In this case, click **Next**, and the Select Additional Tasks dialog box appears.



Setup - LINQPad 4

Select Additional Tasks

Which additional tasks should be performed?

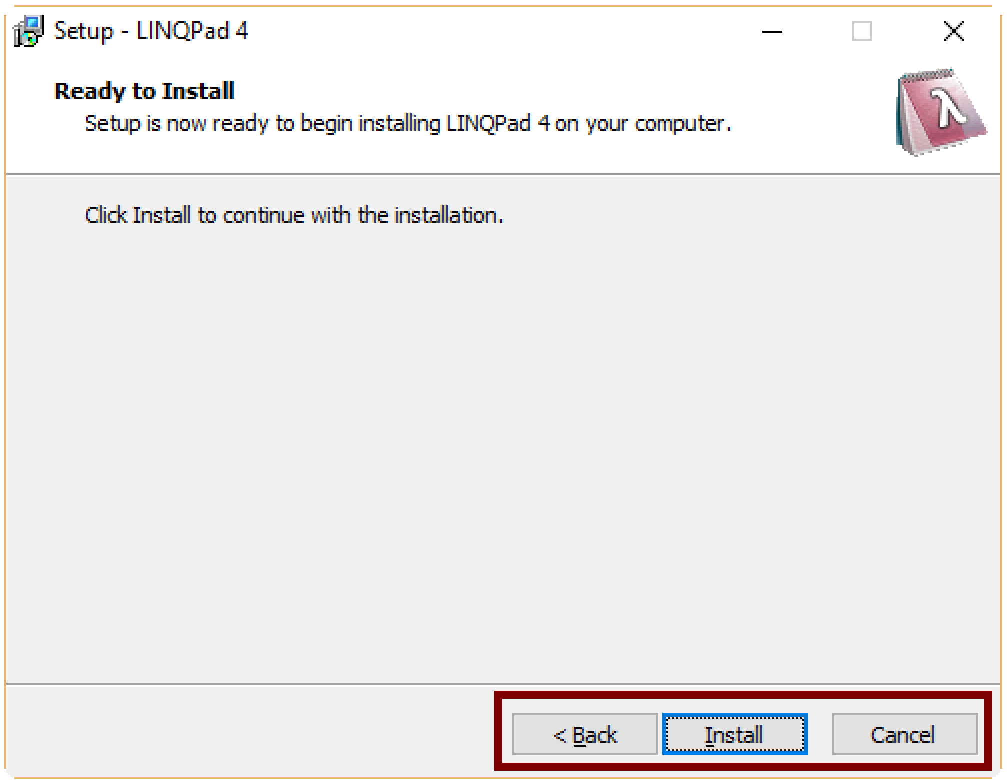
Select the additional tasks you would like Setup to perform while installing LINQPad 4, then click Next.

- [✓] Associate .LINQ files with LINQPad
- [✓] Create a desktop icon
- [ ] Create a Quick Launch icon
- [✓] Add lprun.exe to system PATH (for command-line support)

Figure 5: Select Additional Tasks dialog box

In this dialog, you select a set of additional tasks to be performed by the installation program. These tasks include the association of every file with the .linq extension to be opened by LINQPad, creating a desktop icon for the program, creating and icon in the Quick Launch bar, and adding the lprun.exe command to the PATH system variable.

The previous figure shows the tasks that need to be performed for the practical purposes of this book. Click **Next** to continue, and the Ready to Install dialog appears.


Setup - LINQPad 4

Ready to Install

Setup is now ready to begin installing LINQPad 4 on your computer.

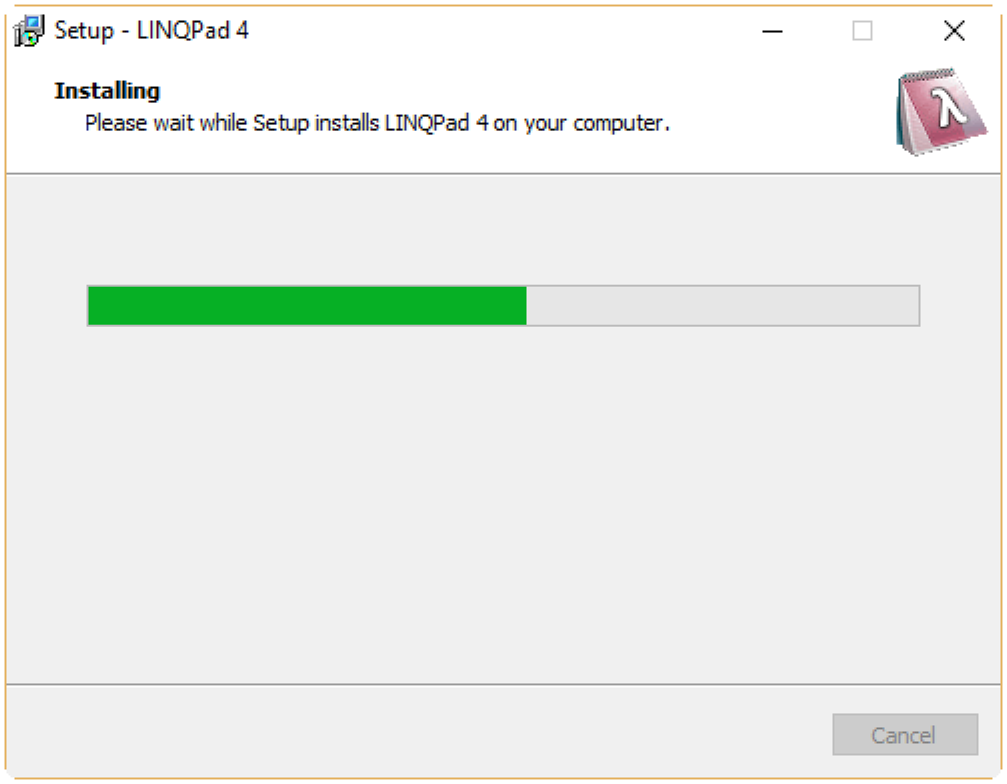
Click Install to continue with the installation.

Figure 6: Ready to Install dialog box

At this point, you have three options.

- Click **Back** to change any of the previous selections made.
- Click **Cancel** to abort the installation process.
- Click **Install** to start the installation process.

If you click **Install**, the Installing dialog box appears and the installation process starts.

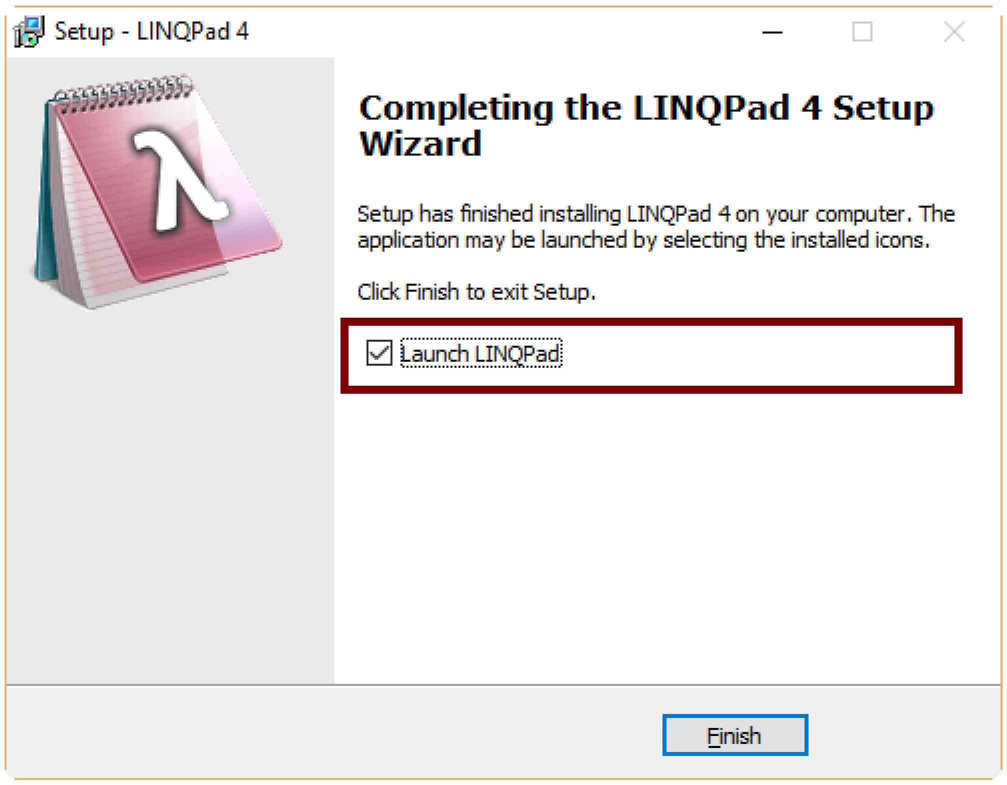

Setup - LINQPad 4

Installing

Please wait while Setup installs LINQPad 4 on your computer.

Figure 7: Installing dialog box

When the setup program completes LINQPad installation, the Installation Finished dialog box appears.


Setup - LINQPad 4

Completing the LINQPad 4 Setup Wizard

Setup has finished installing LINQPad 4 on your computer. The application may be launched by selecting the installed icons.

Click Finish to exit Setup.

☑ Launch LINQPad

Finish

Figure 8: Installation Finished dialog box

At the end of the installation process, make sure the **Launch LINQPad** check box is selected in the dialog box, and click **Finish** to launch LINQPad.

LINQPad deployment structure

LINQPad installation creates a set of folders and files when it deploys the application. These folders and files hold all the necessary elements that make LINQPad work properly. Such elements are queries, plugins, code snippets, data context drivers, connections, default references, and user preferences. Some of these elements will be explained later in this book. The folders and files created by the installation process are displayed in the following table.

Table 2: LINQPad folder structure

Folder	Purpose
Documents\LINQPad Queries	Holds all queries created by the user. LINQPad uses this folder to display those queries in a tree view named My Queries.

Folder	Purpose
Documents\LINQPad Plugins	Holds any custom assembly referenced by all or some queries.
Documents\LINQPad Snippets	Holds custom code snippets (doesn't apply for free edition).
%LocalAppData%\LINQPad\drivers	Holds custom data context drivers that can be used in all queries.
%AppData%\LINQPad\ConnectionsV2.xml	Stores all connections that appear in the Connections tree view.
%AppData%\LINQPad\DefaultQuery.xml	Stores all assembly references that will be used by default in queries.
%AppData%\LINQPad\RoamingUserOptions.xml	Stores the user prefrences that dictate LINQPad behavior.

Note: If a same structure is created in the folder where LINQPad.exe is located, this structure will take precedence over the structure displayed in the previous table.

Creating a portable deployment of LINQPad

LINQPad can be deployed with a USB memory stick in order to use it as a portable application. To succeed in this matter, a folder named LINQPad (for practical purposes) should be created in the memory stick. Then, the following folder structure should be created within the LINQPad folder.

Code Listing 1

```
LINQPad/
         LINQPad.exe
         LINQPad.exe.config
         Que
ries/                -- (for saving queries)
         plugins/                -- (plugins + 'My Extensions' queries here)
         snippets/               -- (for saving custom snippets)
         drivers/                -- (for saving custom drivers)
         ConnectionsV2.xml       -- (used to populate Schema tree view)
         DefaultQuery.xml        -- (default namespaces/references for new queries)
```

```
        RoamingUserOptions.xml  -- (user preferences)
        Lprun.exe               -- (for command line support)
```

Now you can use LINQPad by inserting the memory stick in a USB port, then executing the **LINQPad.exe** file located in the LINQPad folder.

Note: The "portable application" concept is applied to a program that can be executed in any computer by simply running its .exe file without a previous installation process.

Chapter summary

This chapter describes the LINQPad installation process, which starts with downloading the LINQPad installation program. Once the installation program is downloaded, the user should double-click on the **LINQPad4Setup.exe** (or **LINQPad5Setup.exe**) file to begin the installation.

All installation processes are guided by a series of dialogs allowing you to select different options along the process. These options include the program's location folder, file associations, icons creation, and environment variables setting for command line support.

When LINQPad is installed, a set of folders and files is created in order to make sure LINQPad works properly. These folders and files hold all necessary elements needed by LINQPad, such as queries, plugins, code snippets, data context drivers, connections, default references, and user preferences.

LINQPad can also be deployed as a portable application. To accomplish this task, a special directory structure should be created in a USB memory stick, including the linqpad.exe file within it.

Chapter 3 Beginning with LINQPad

Running LINQPad

To start LINQPad, double-click on the **LINQPad** icon on the Windows desktop. This action brings up the LINQPad user interface.

Figure 9: LINQPad icon on the Windows desktop

LINQPad user interface

The LINQPad user interface corresponds to the client application in which users will do most of their work.

User interface graphic elements

The following graphic elements of the interface, which we will be using for the purposes of this book, are shown in Figure 10.

1. **LINQPad main menu**: Holds all LINQPad commands available to work with the program.
2. **Connection's tree view area**: Shows all connections created by the user with the connection manager.
3. **Queries and Samples tree view area**: Shows all queries and extensions created by the user. It also shows all available samples within LINQPad, which are displayed in separate tabs.
4. **Query code tab window area**: Where the user will write the query code.
5. **Query code tab caption**: Shows the name of the query that is being edited.
6. **Add New Tab button**: Adds a new tab for query editing.
7. **Execute Query button**: Executes the query that is currently selected.

8. **Stop Execution button**: Stops query execution.
9. **Results View Type button**: Allows the user to switch between Rich Text and Data Grid, for viewing query results.
10. **Language Selector combo box**: Allows the user to select the programming language in which queries will be written.
11. **Connection Selector combo box**: Allows the user to select a previously created custom connection.
12. **Close Tab button**: Closes the tab currently selected.
13. **Premium Features activation link**: In the free version of LINQPad, this link allows the user to open the product's webpage to acquire a license, for activating paid license's features.
14. **Status Bar**: Shows several messages sent by the program, including query's time performance.

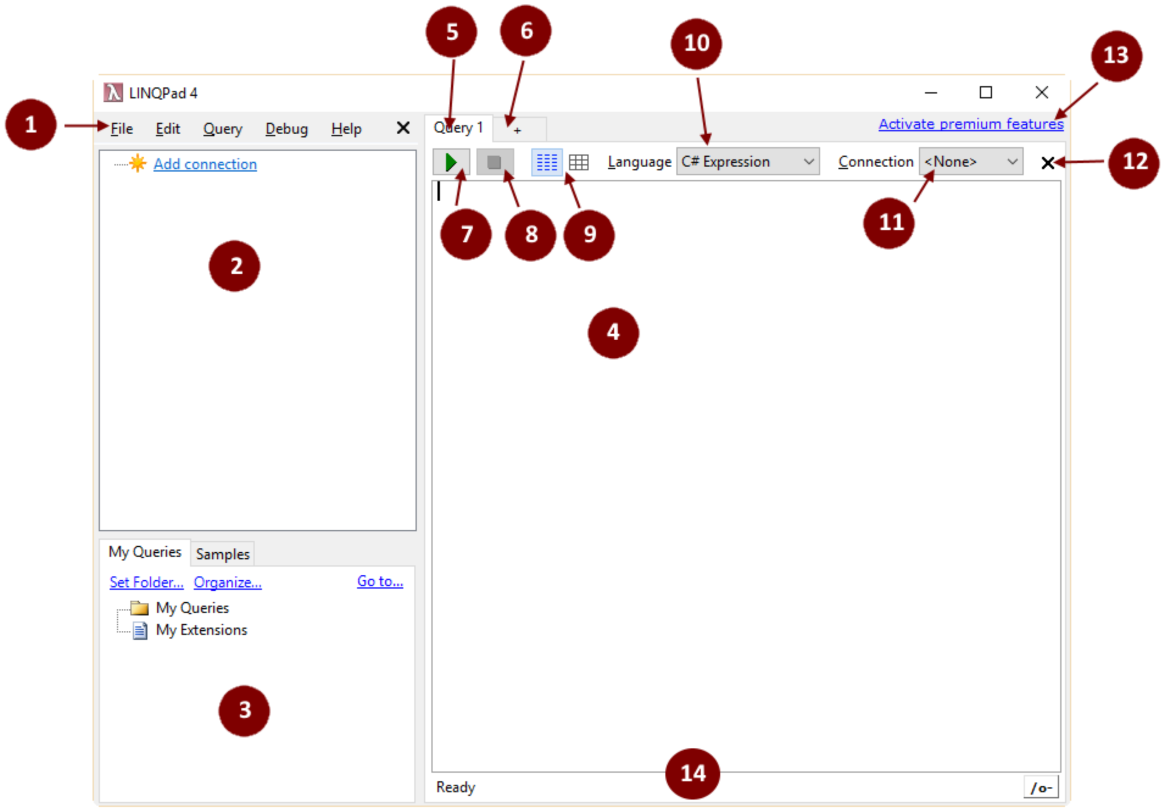


Figure 10: LINQPad user interface

Customizing the editor window

You can customize the editor window's font and other elements, such as the window background color or displayed code line numbers. This can be done by using the **Preferences** command located in the **Edit** menu.

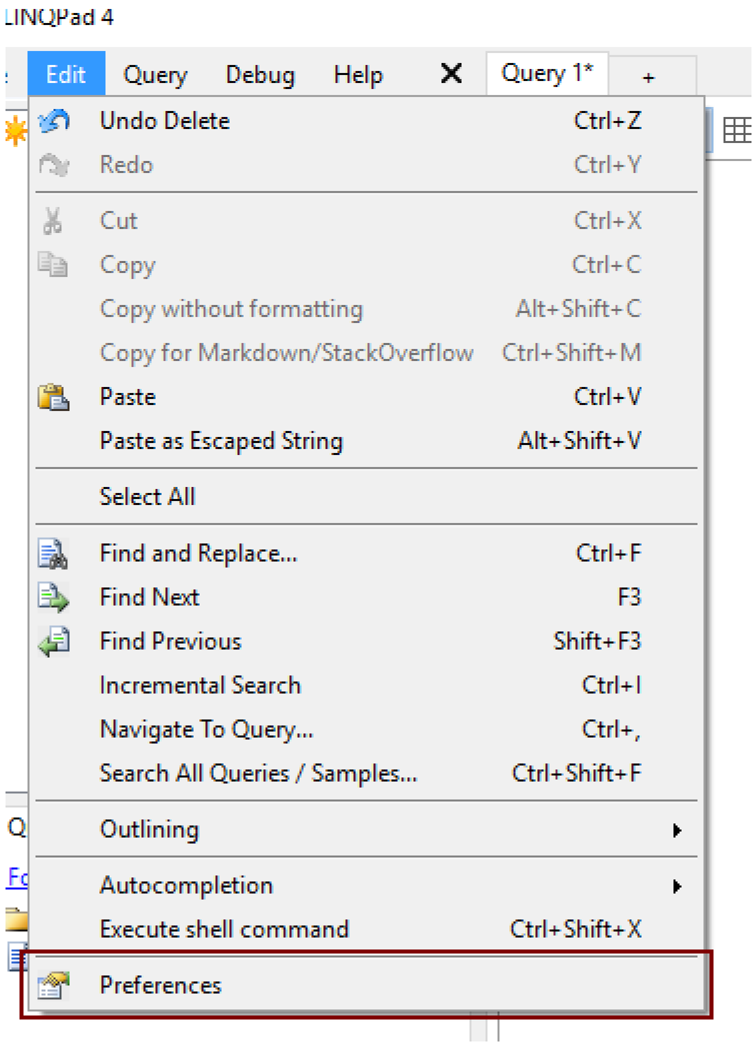
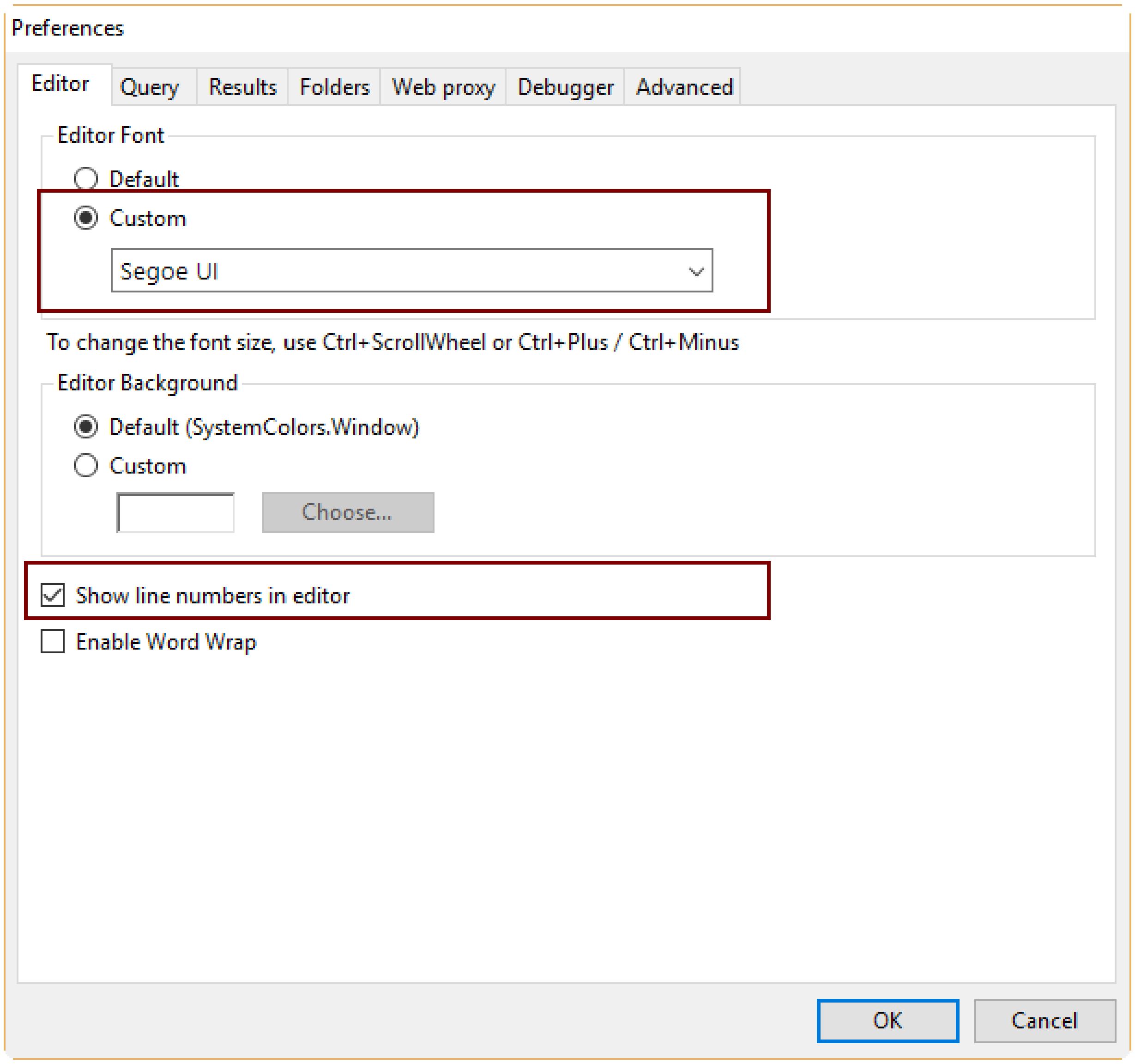

LINQPad 4
Edit
Query
Debug
Help
Query 1*
Undo Delete Ctrl+Z
Redo Ctrl+Y
Cut Ctrl+X
Copy Ctrl+C
Copy without formatting Alt+Shift+C
Copy for Markdown/StackOverflow Ctrl+Shift+M
Paste Ctrl+V
Paste as Escaped String Alt+Shift+V
Select All
Find and Replace... Ctrl+F
Find Next F3
Find Previous Shift+F3
Incremental Search Ctrl+I
Navigate To Query... Ctrl+,
Search All Queries / Samples... Ctrl+Shift+F
Outlining
Autocompletion
Execute shell command Ctrl+Shift+X
Preferences

Figure 11: Preferences command in Edit menu

For the purposes of this book, I customized my editor window by setting the font to Segoe UI and making code line numbers visible.



Editor Font

Default [✓ Custom]

Segoe UI

To change the font size, use Ctrl+ScrollWheel or Ctrl+Plus / Ctrl+Minus

Editor Background

[✓ Default (SystemColors.Window)] Custom

☑ Show line numbers in editor

☐ Enable Word Wrap

Figure 12: Editor Window Preferences

It can be noted in Figure 12 that font size customization is not available. That's because LINQPad allows the user to increase or decrease font size in two different ways: by using the Ctrl+Plus (increase) or Ctrl+Minus (decrease) key combinations, or by using the mouse scroll wheel while pressing the Ctrl key at the same time.

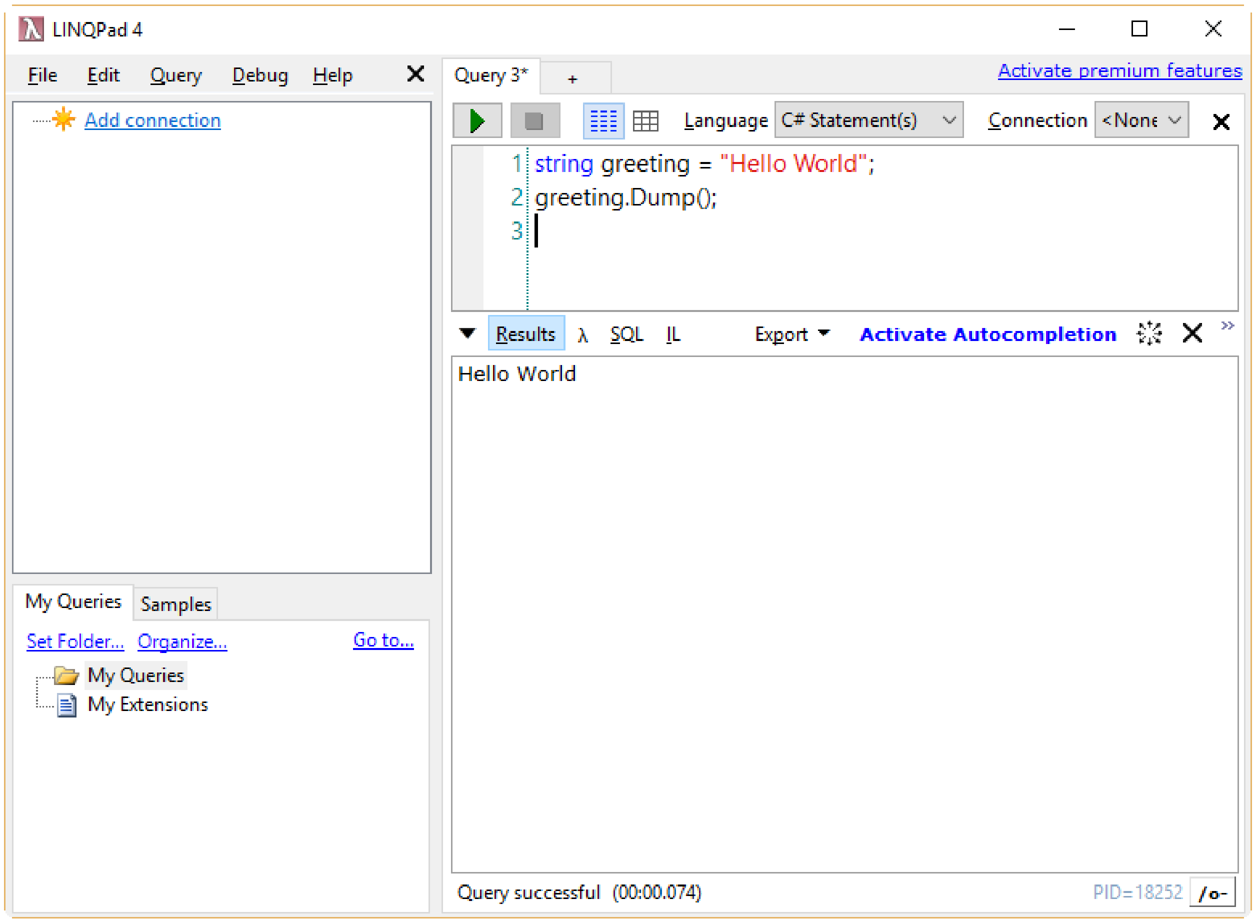
The ever-present Hello World

Our first meeting with LINQPad will take place by creating and running the traditional Hello World code. To do this, first set the **Language** drop-down control to **C# Statement(s)**, then type the following code into the Query window.

Code Listing 2

```
string greeting = "Hello World";
greeting.Dump();
```

Now, to run to previous code, click on the **Execute Query** (green arrow) button. The output for this code is shown in the following figure.



```
string greeting = "Hello World";
greeting.Dump();
```

```
Hello World
```

Query successful (00:00.074)

PID=18252

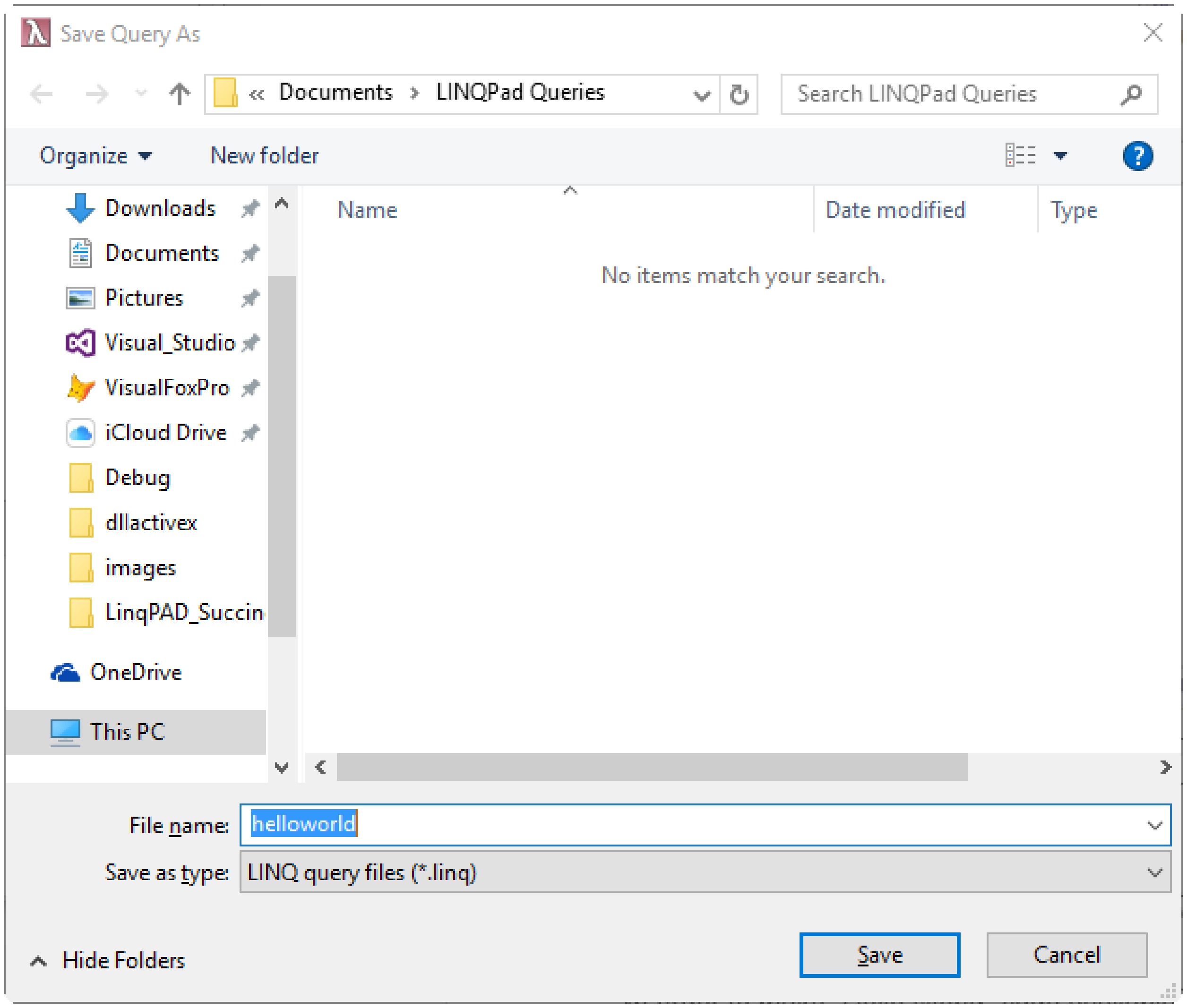
Figure 13: Hello World sample output

Saving the Hello World code

With LINQPad, all the queries you write can be easily accessed using the My Queries tree view located at the bottom-left side of the user interface.

In order to make the Hello World code available from tree view, you need to save it first. This can be done by pressing **Ctrl+S**. In the **Save Query As** dialog box, type the file name for saving the query. For the purposes of this book, the name will be **helloworld**. The file will be saved with a .linq extension.

The file name given to the query will appear beneath the My Queries node of the tree view after it is saved in the disk. Now you can quickly access the **helloworld** code when needed.

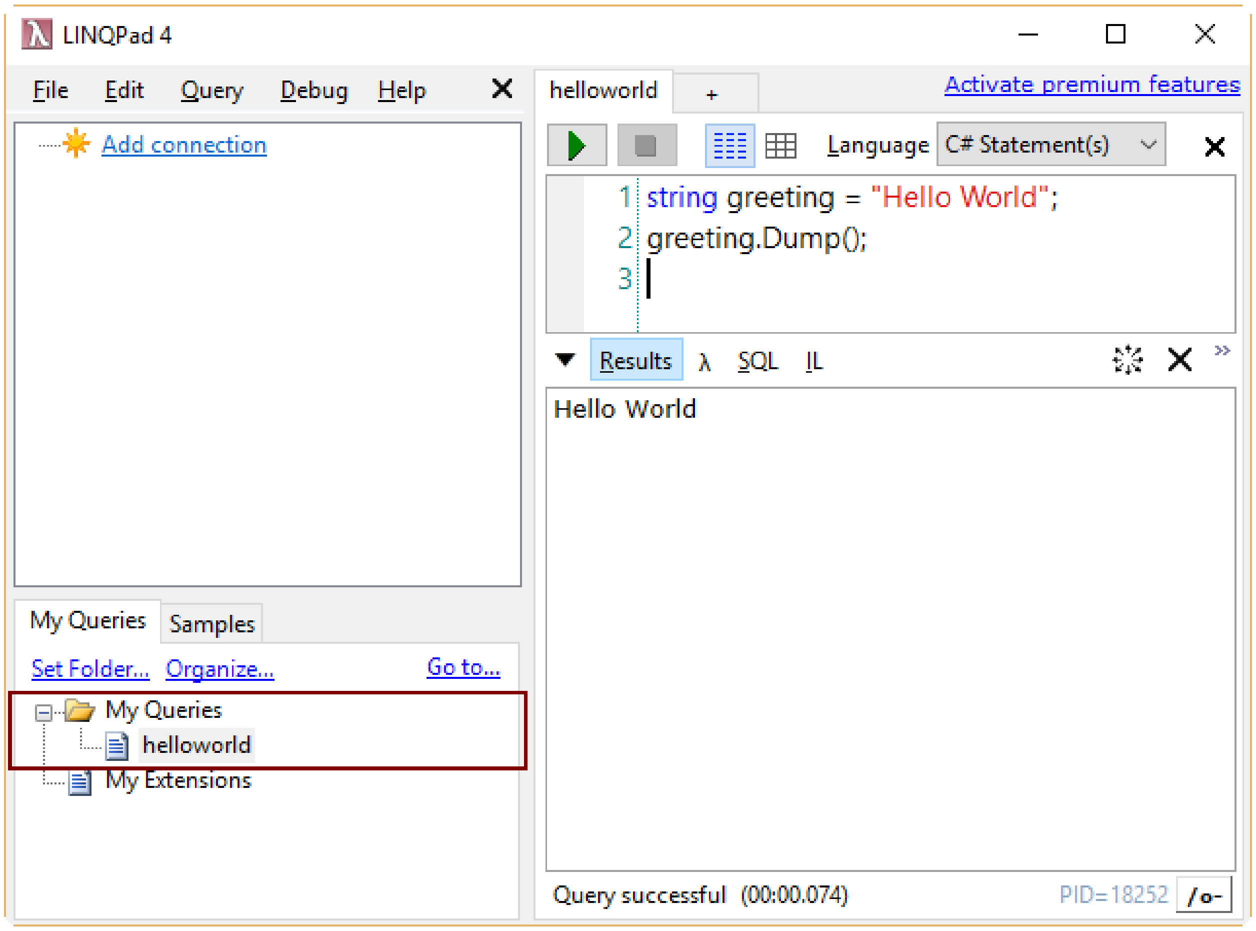


No items match your search.

File name: helloworld

Save as type: LINQ query files (*.linq)

Figure 14: The "Save Query As" dialog box



```
string greeting = "Hello World";
greeting.Dump();
```

```
Hello World
```

Query successful (00:00.074)

Figure 15: The helloworld query beneath the My Queries node.

Chapter summary

This chapter explained how to begin using LINQPad. To start LINQPad, double-click on the icon placed on the Windows desktop. This will launch the LINQPad user interface, which is the client application that will do most of the work.

We identified 14 graphic elements in LINQPad user interface for purposes of this book. These elements are: LINQPad main menu, Connection's tree view area, Queries and Samples tree view area, Query code tab window area, Query code tab caption, Add New Tab button, Execute Query button, Stop Execution button, Results View Type button, Language Selector combo box, Connection Selector combo box, Close Tab button, Premium Features activation link, and Status Bar.

You can customize the editor window by using the **Preferences** command from the **Edit** menu. Then, the settings for the window can be changed in the **Editor** tab of the **Preferences** dialog. For purposes of this book, the Segoe UI font will be used, and code line numbers will be displayed.

Font size customization is not available, because LINQPad allows the user to increase or decrease this size using Ctrl+Plus (increase) or Ctrl+Minus (decrease) keys combinations, or the mouse scroll wheel while pressing the Ctrl key at the same time.

A “Hello World” example was used to start with LINQPad work. The output produced by this code was displayed in the Results window.

The code of the “Hello World” example was saved into a file named helloword.linq in order to make it available for future use. After that, the filename appeared beneath the My Queries node located at the bottom-left of the user interface.

Chapter 4 LINQPad Basics

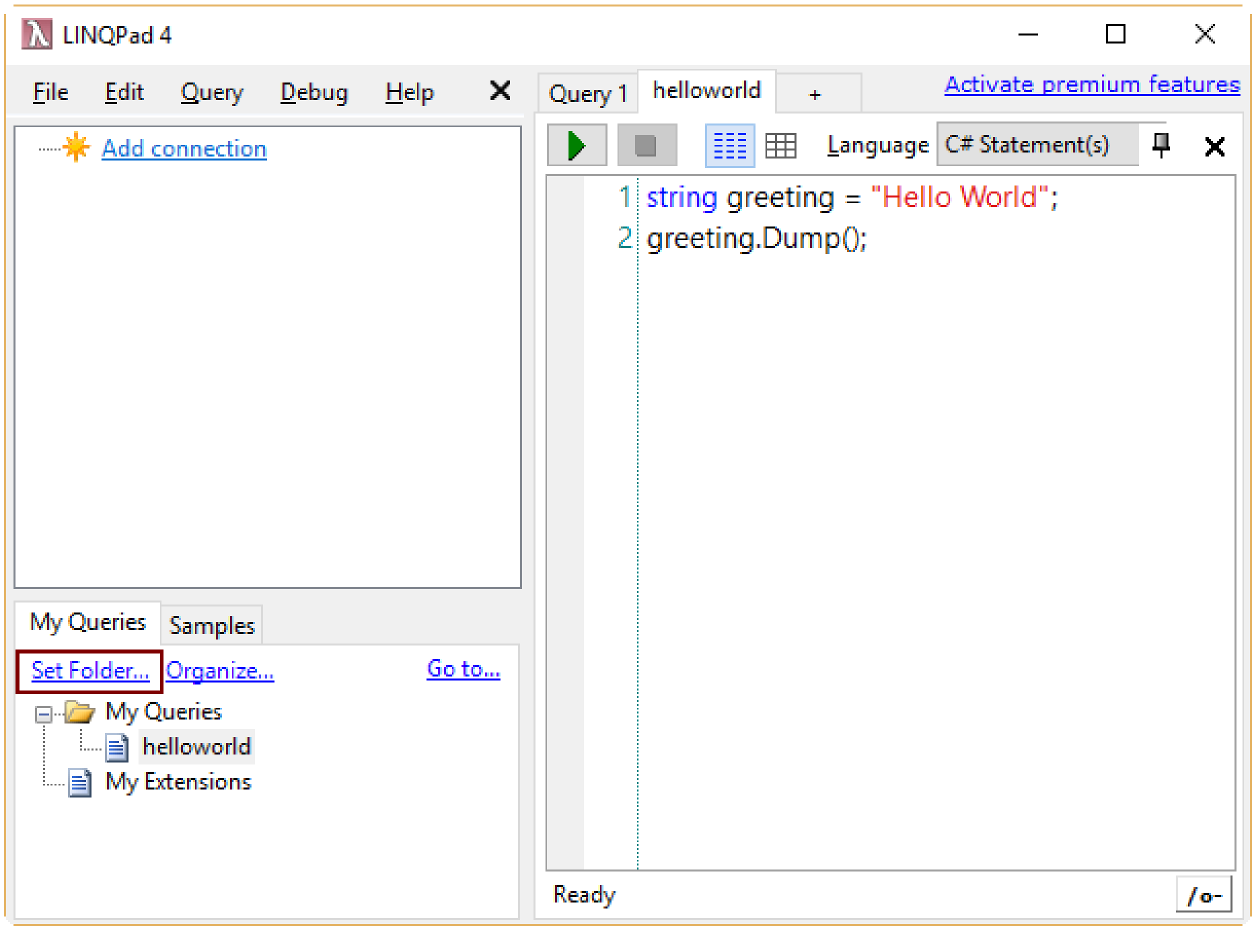
Queries: the basic concept in LINQPad

LINQPad uses the term *query* for every piece of code that is written in the editor. So from this point forward in *LINQPad Succinctly*, the term *query* will be used for pointing to every code sample displayed in it.

As discussed in the “Save Hello World code” section of the previous chapter, a query can be saved for future use. When a query is saved, LINQPad will display its filename beneath a tree view node called My Queries. This tree view is located at the bottom-left side of the user interface.

Changing the location where queries are saved

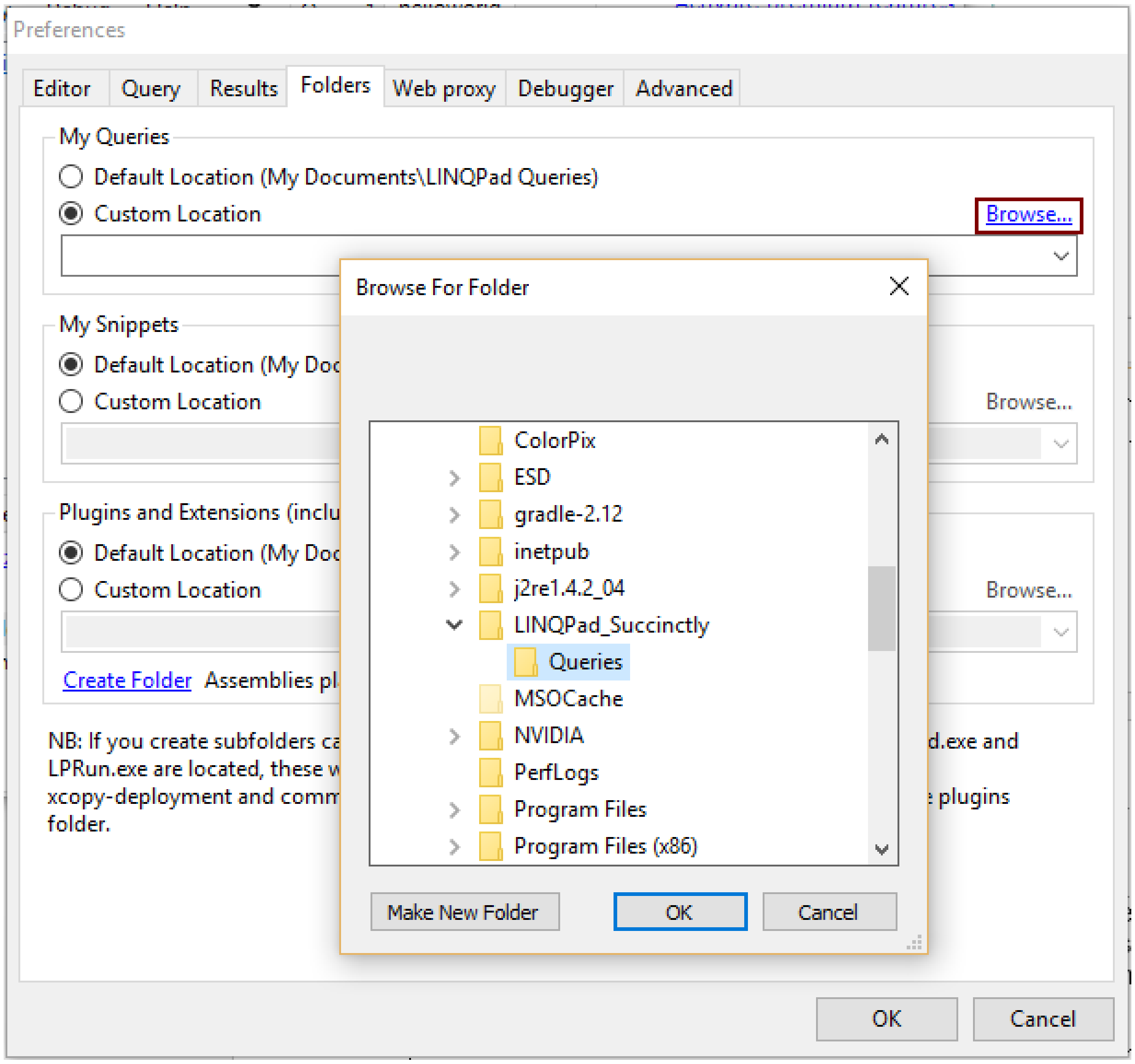
LINQPad stores queries in a folder named LINQPad Queries, which by default is located in the Documents folder that belongs to the current user. This behavior can be changed by clicking the **Set Folder** hyperlink located above the **My Queries** tree view node. This action will bring up the Preferences dialog box with the Folders tab selected.



```
string greeting = "Hello World";
greeting.Dump();
```

Figure 16: The Set Folder hyperlink

All queries discussed in this book will be stored in a folder named C:\LINQPad_Succinctly\Queries. To select this folder, first create it on disk, and then click the **Custom Location** button located under the **My Queries** framework in the **Preferences** dialog box. Next, click the **Browse** hyperlink to the right of the same option button, and the Browse For Folder dialog box will appear.



My Queries

( ) Default Location (My Documents\LINQPad Queries)

(●) Custom Location

My Snippets

(●) Default Location (My Doc

( ) Custom Location

Plugins and Extensions (inclu

(●) Default Location (My Doc

( ) Custom Location

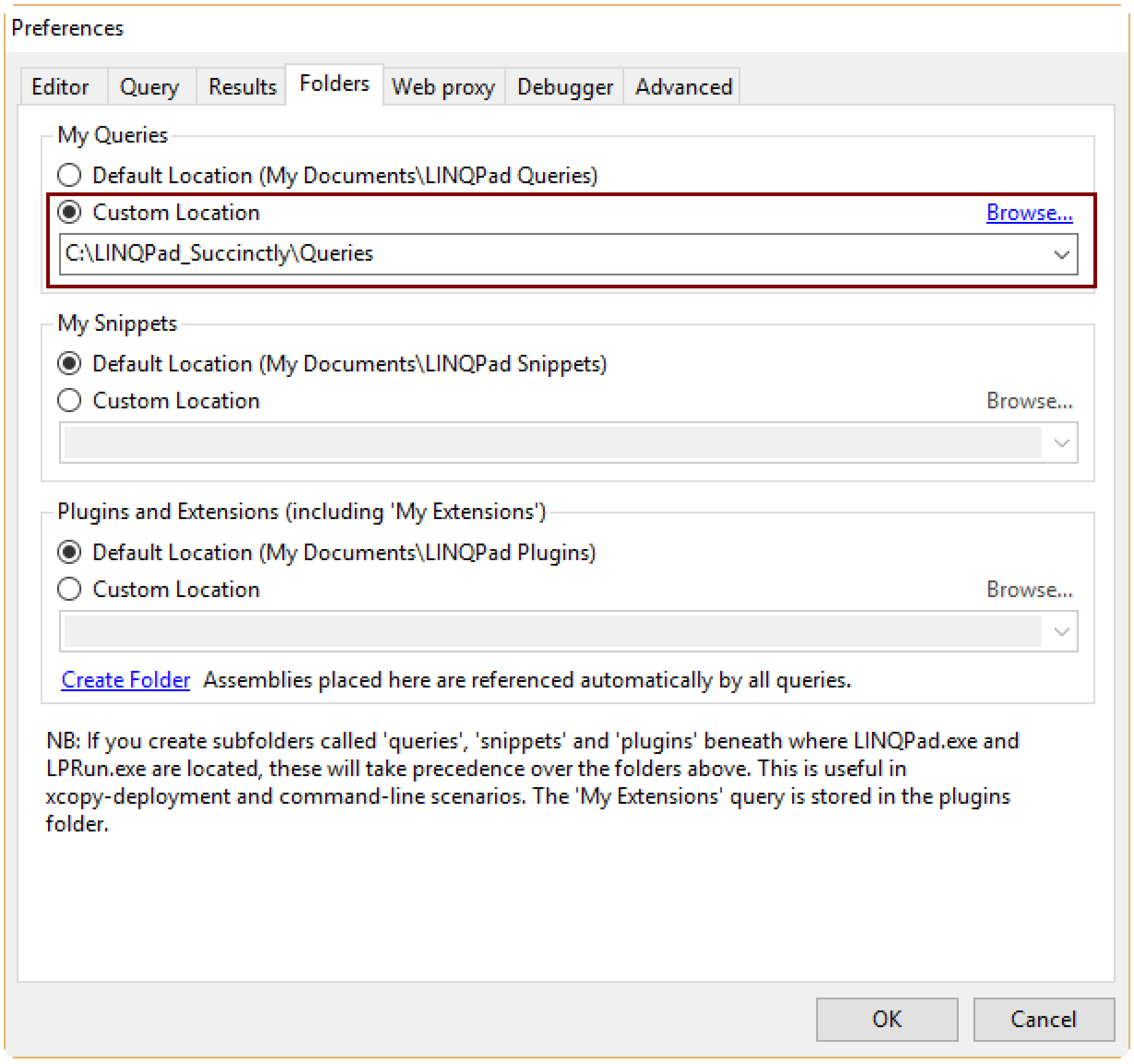
Assemblies pl

NB: If you create subfolders ca d.exe and LPRun.exe are located, these w xcopy-deployment and comm e plugins folder.

ColorPix
ESD
gradle-2.12
inetpub
j2re1.4.2_04
LINQPad_Succinctly
Queries
MSOCache
NVIDIA
PerfLogs
Program Files
Program Files (x86)

Figure 17: The Browse For Folder dialog box

Select **C:\LINQPad_Succinctly\Queries**, and then click **OK**. After that, the folder name will appear in the combo box list located under the Custom Location option button.



My Queries

( ) Default Location (My Documents\LINQPad Queries)
[✓ Custom Location] Browse...
C:\LINQPad_Succinctly\Queries

My Snippets

[✓ Default Location (My Documents\LINQPad Snippets)]
( ) Custom Location Browse...

Plugins and Extensions (including 'My Extensions')

[✓ Default Location (My Documents\LINQPad Plugins)]
( ) Custom Location Browse...

Create Folder Assemblies placed here are referenced automatically by all queries.

NB: If you create subfolders called 'queries', 'snippets' and 'plugins' beneath where LINQPad.exe and LPRun.exe are located, these will take precedence over the folders above. This is useful in xcopy-deployment and command-line scenarios. The 'My Extensions' query is stored in the plugins folder.

Figure 18: The Preferences dialog after the C:\LINQPad_Succinctly\Queries folder

Click **OK** to accept all changes made.

Searching for a specific query

In a working environment with a few queries, locating a specific one may be a simple task. But when those queries increase in number, this can be a hard work. LINQPad solves this issue by allowing you to search queries using a term. To do this, click on the **Go to** hyperlink located on the **My Queries** tree view node.

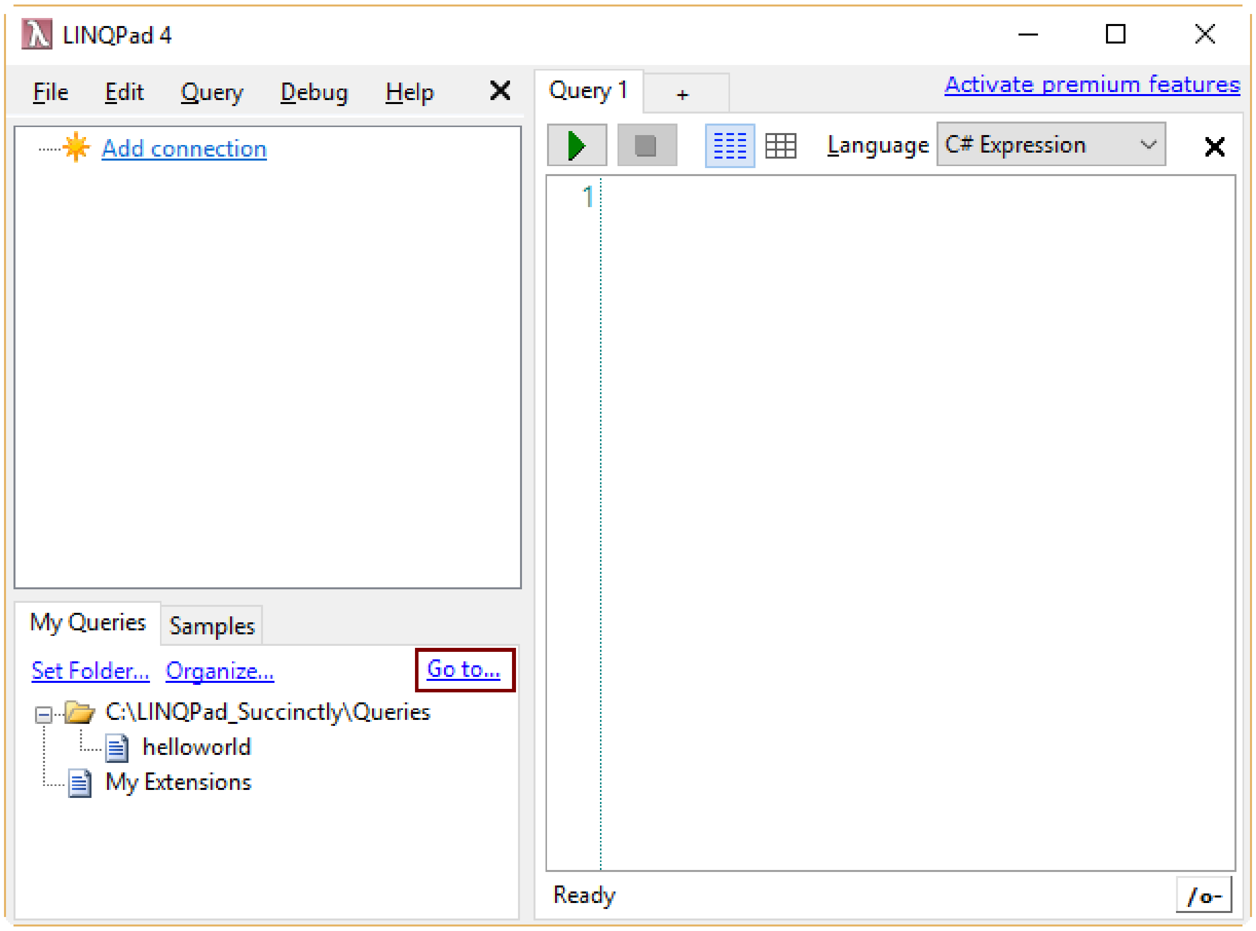


Figure 19: The Go to hyperlink.

The Navigate To Query dialog box will appear on the screen. At this point, the user can type a term into the **Search Terms** combo box, and LINQPad will display all queries with names that contain the term typed. Typing the term **hel** will cause LINQPad to display the helloword.linq query in the Result list.

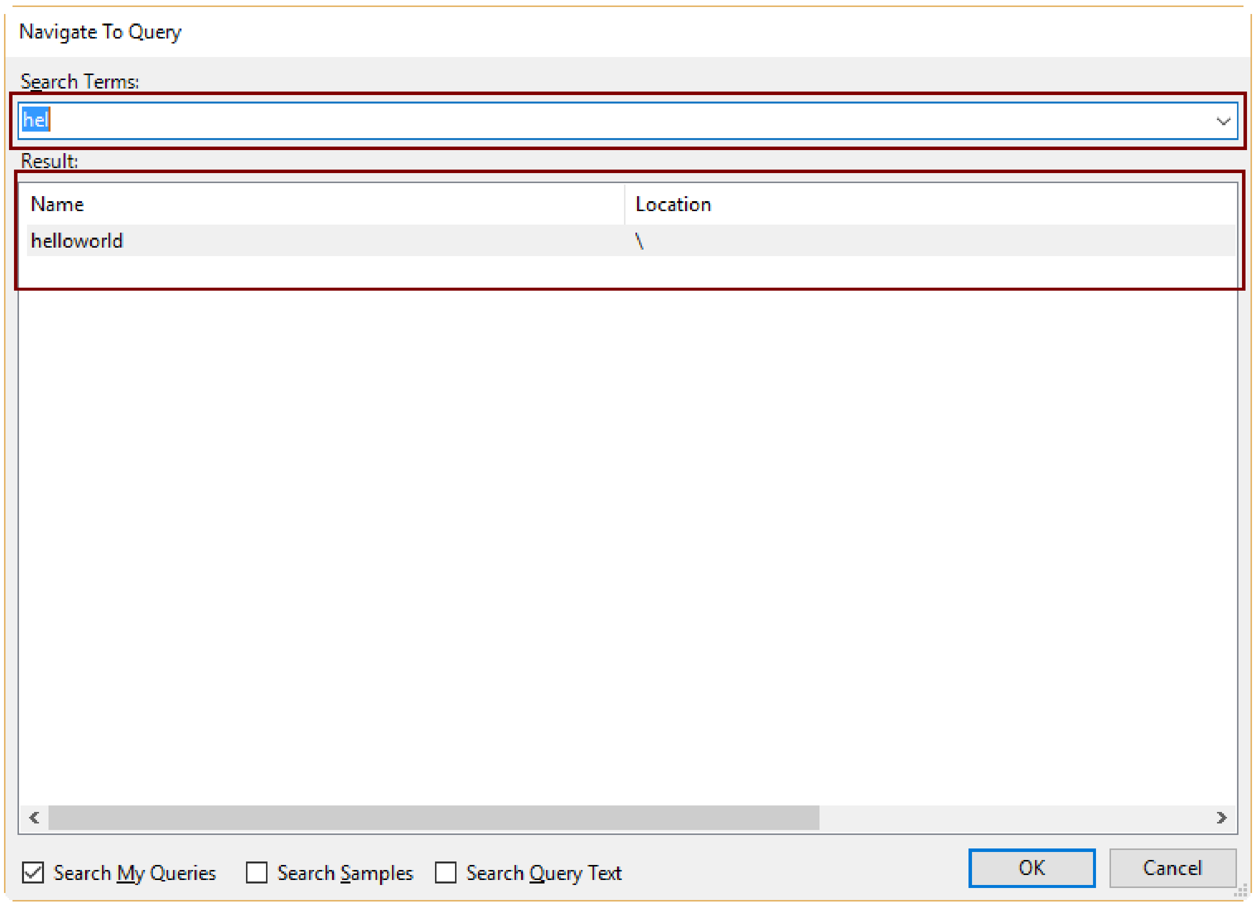


Figure 20: The Navigate to Query dialog

Select the **helloworld** query from the list, and then click **OK** to open this query in the editor.

Programming languages available in LINQPad

In the “LINQPad user interface” section of previous chapter, 14 graphic elements were identified. One of these elements was the Language Selector combo box. As explained in this section, this graphic element allows the user to choose the programming language that will be used to write queries.

LINQPad has the following programming languages available:

- C# Expression: An expression written in C#, with no semicolon at the end of line.
- C# Statements: Several statements written in C#. Each code line must end with a semicolon.
- C# Program: A C# `Main ()` method, which can be followed by another methods or another class definitions. Each code line must end with a semicolon.
- VB Expression: Same as C# Expression, but coded in VB.Net.
- VB Statements: Same as C# Statements, but coded in VB.Net.
- VB Program: Same as C# Program, but coded in VB.Net.
- SQL: One or more statements coded in SQL language.

- ESQL: Entity SQL Language Queries.
- F# Expression: An expression written in F#.
- F# Program: A program block written in F#.

Query file internal structure

When a query is saved, LINQPad associates the .linq file extension to its filename. Internally, a header in XML format is added before the code written in the query. The following sample displays this structure.

Code Listing 3

```
<Query Kind="Expression" />

// Here it is, the simplest "hello, world" ever!  (Just hit F5)

"Hello, world!"
```

As noted in the previous sample, the root node for the XML header saved with the query is precisely named **`Query`**. The attribute **`Kind`** for this node tells LINQPad which type of programming language should be used when compiling the query. This matches with the option chosen from the Language Selector combo box.

If a query uses a SQL connection (which will be explained in the following section), the internal structure for the query file will look like the following sample.

Code Listing 4

```
<Query Kind="Expression">
  <Connection>
    <ID>4dbf890a-9be0-431c-a6d4-4bb20c1ba604</ID>
    <Persist>true</Persist>
    <Server>.\SQLEXPRESS</Server>
    <SqlSecurity>true</SqlSecurity>
    <UserName>sa</UserName>

<Password>AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAApIpboxkMi0KXQm107ZYAoQAAAAACAAAAAAAQZgAAA
AEAACAAAADdjRylNWoymuJ4gbmgPQgfMFr1PooNmcD3wfZviEVW3QAAAAAOgAAAAIAACAAAADE6tComfTn
fJ/vn8LgxcYRUyRtOIOmztt8luz6FzRMohAAAAAa5ENguOQMGdGZdrnVXQi0QAAAALj09Vdcv09hYQIHq/n
iC9ydeO4NMEUvl8BpUNFKq4hYdG7sRBPTIOFe2a7M55OueQV5T916HCR6MaMom1sOSIA=</Password>
    <Database>uspostalcodes</Database>
    <ShowServer>true</ShowServer>
  </Connection>
</Query>

Places.Where(row => row.County_id == 1926)
```

Now the **`Query`** node of the XML header contains a node named **`Connection`**. This node stores all parameters needed to connect to the database server employed by the query.

With both of these samples, the XML header added to the query tells LINQPad which programming language and SQL connection the query uses.

Managing SQL connections

We can manage connections to SQL servers using the LINQPad user interface. This can be done in the Connection's tree view area, located at the left side of the interface, by clicking on the **Add connection** hyperlink.

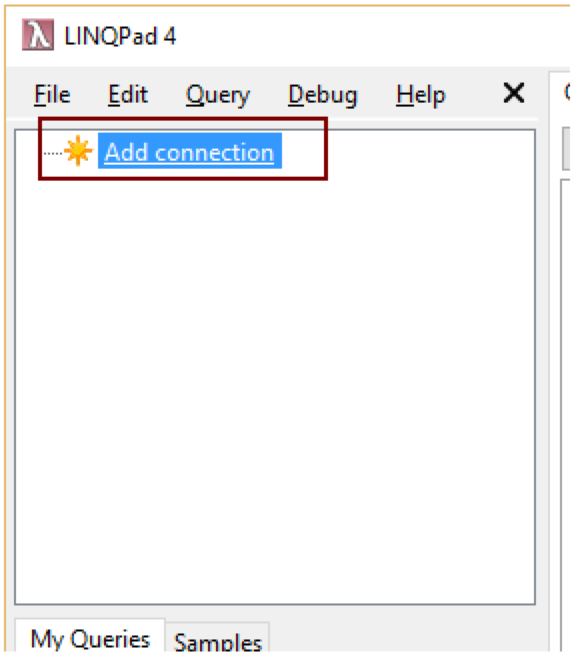
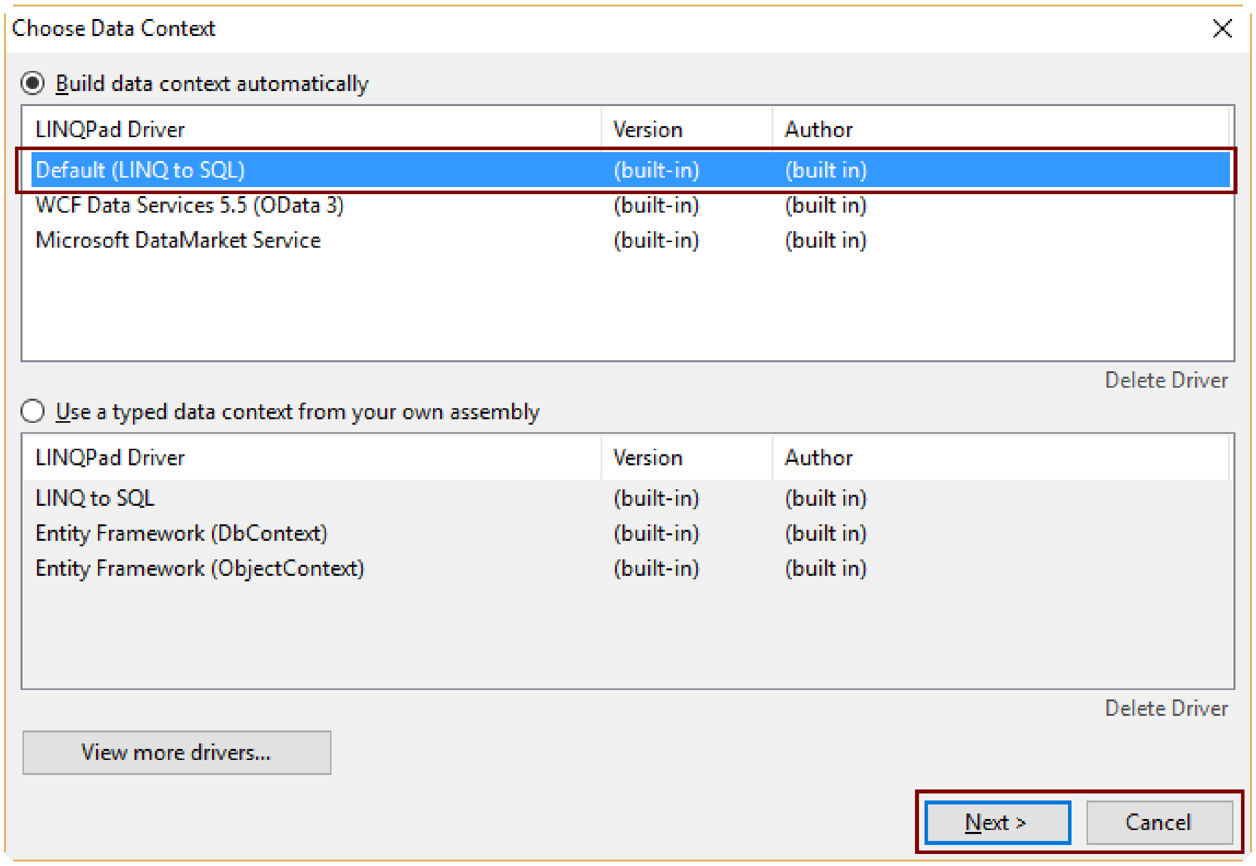


Figure 21: The Add connection hyperlink in Connection's tree view area

Creating a SQL connection

When you click the **Add connection** hyperlink, the **Choose Data Context** dialog box will be displayed. In this dialog, you can choose the LINQPad driver that will be used to make the connection. In this case, the **LINQ to SQL** driver (which is the default) will be used.

To complete driver selection, click **Next**. (Clicking **Cancel** will abort the process.)


Choose Data Context

(●) Build data context automatically

LINQPad Driver	Version	Author
Default (LINQ to SQL)	(built-in)	(built in)
WCF Data Services 5.5 (OData 3)	(built-in)	(built in)
Microsoft DataMarket Service	(built-in)	(built in)

Delete Driver

( ) Use a typed data context from your own assembly

LINQPad Driver	Version	Author
LINQ to SQL	(built-in)	(built in)
Entity Framework (DbContext)	(built-in)	(built in)
Entity Framework (ObjectContext)	(built-in)	(built in)

Figure 22: The Choose Data Context dialog

The LINQPad Connection dialog box is shown (titled “LINQ to SQL Connection” in LINQPad 5) in order to set connection’s parameters. The example for this book will create a Microsoft SQL Server connection. Select the **SQL Server** option from the **Provider** frame, located at the top of the dialog box.

It is assumed that a Microsoft SQL Server Express instance is installed in the local computer, and that this instance it was named SQLEXPRESS during the SQL’s installation process. It’s also assumed that this instance was configured for a SQL Authentication security mode.

Regarding how databases will be managed, LINQPad gives three options:

- Display all databases stored in the server, using a TreeView.
- Attach a database file stored in the local hard drive to the server.
- Specify a specific database from the server, or create a new one.

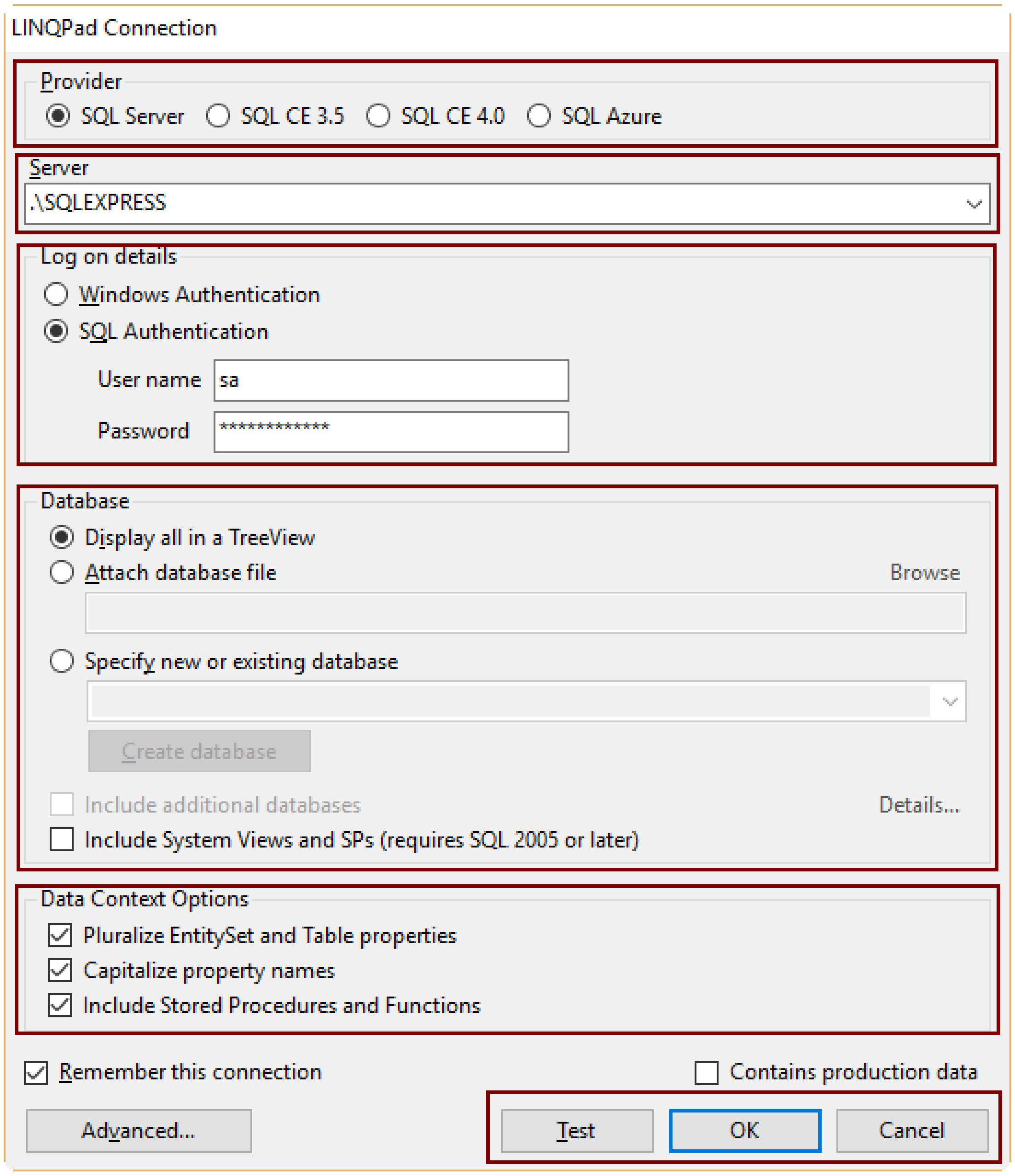
For purposes of this book, the first option will be selected.

Finally, the user should configure a set of options for the connection’s Data Context.

- **Pluralize EntitySet and Table properties**. All entities and table properties are shown in plural.

- **Capitalize property names**. All name for properties appear with the first letter in uppercase.
- **Include Stored Procedures and Functions**. When checked, all functions and stored procedures belonging to any database that appears in the tree view are displayed.

The connection used in this book requires all default options being left with no changes.


LINQPad Connection

Provider
(● SQL Server) ○ SQL CE 3.5 ○ SQL CE 4.0 ○ SQL Azure

Server
.\SQLEXPRESS

Log on details
○ Windows Authentication
(● SQL Authentication)

User name	sa
Password	************

Database
(● Display all in a TreeView)
○ Attach database file — Browse
○ Specify new or existing database
Create database
☐ Include additional databases — Details...
☐ Include System Views and SPs (requires SQL 2005 or later)

Data Context Options
☑ Pluralize EntitySet and Table properties
☑ Capitalize property names
☑ Include Stored Procedures and Functions

☑ Remember this connection — ☐ Contains production data

Figure 23: LINQPad Connection dialog box

You can test the connection before accepting the parameters by clicking the **Test** button, located at the bottom of the dialog box. Once the test is passed, click **OK** to create the connection and display it in the user interface.

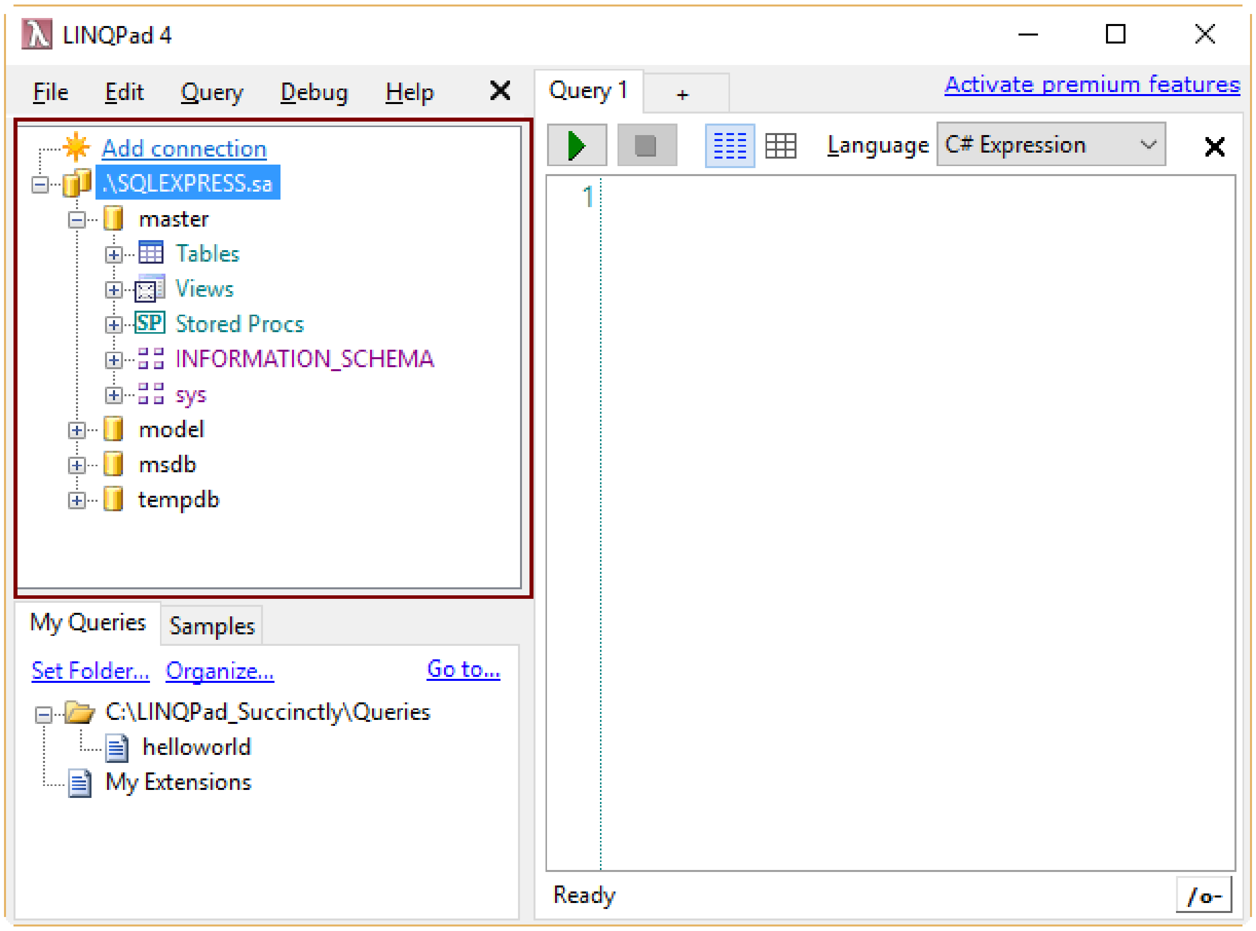


Figure 24: The new connection appears in the Connection's tree view area.

By default, LINQPad names a connection using the server name followed by a period and the name of the authenticated user, when SQL Authentication is chosen. In this case, the name displayed in the tree view is .\SQLEXPRESS.sa. because I connected as the SQL Server sys admin. If Windows Authentication is selected in the LINQPad Connection dialog box, the name given to the connection will contain the server name only.

Note: When a specific database is selected in the LINQPad Connection dialog box, the connection's name will contain the name of the database that was chosen, at the end.

The Connection context menu

The Connection context menu will pop up when the user right-clicks on the connection's name.

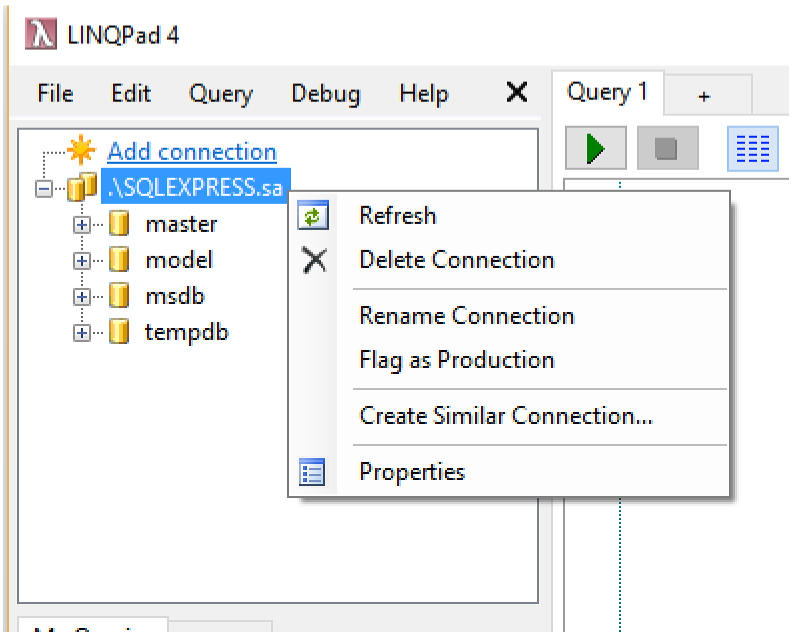


Figure 25: The Connection context menu

The commands available from this menu are:

- **Refresh**: Queries the server and reloads all entities and their properties.
- **Delete Connection**: Removes the selected connection from the tree view.
- **Rename Connection**: Allows you to customize the name for the connection.
- **Flag as Production**: Sets a flag to indicate that the connection used is in a production environment. When clicked, the word “Production” will appear at the right of the connection’s name. Then, the command will appear in the context menu as “Unflag as Production.”
- **Create Similar Connection**: Shows the LINQPad Connection dialog with all the parameters for the selected connection, in order to create an exact new one or a connection with almost the same parameters values.
- **Properties**: Shows the LINQPad Connection dialog box, in order to change connection’s parameters.

Browsing LINQPad samples

The LINQPad user interface provides a special tab named Samples. This tab is located beside the My Queries tree view area.

To work with a specific sample, click on the name of the desired one. LINQPad will automatically bring up the code for that sample into the editor.

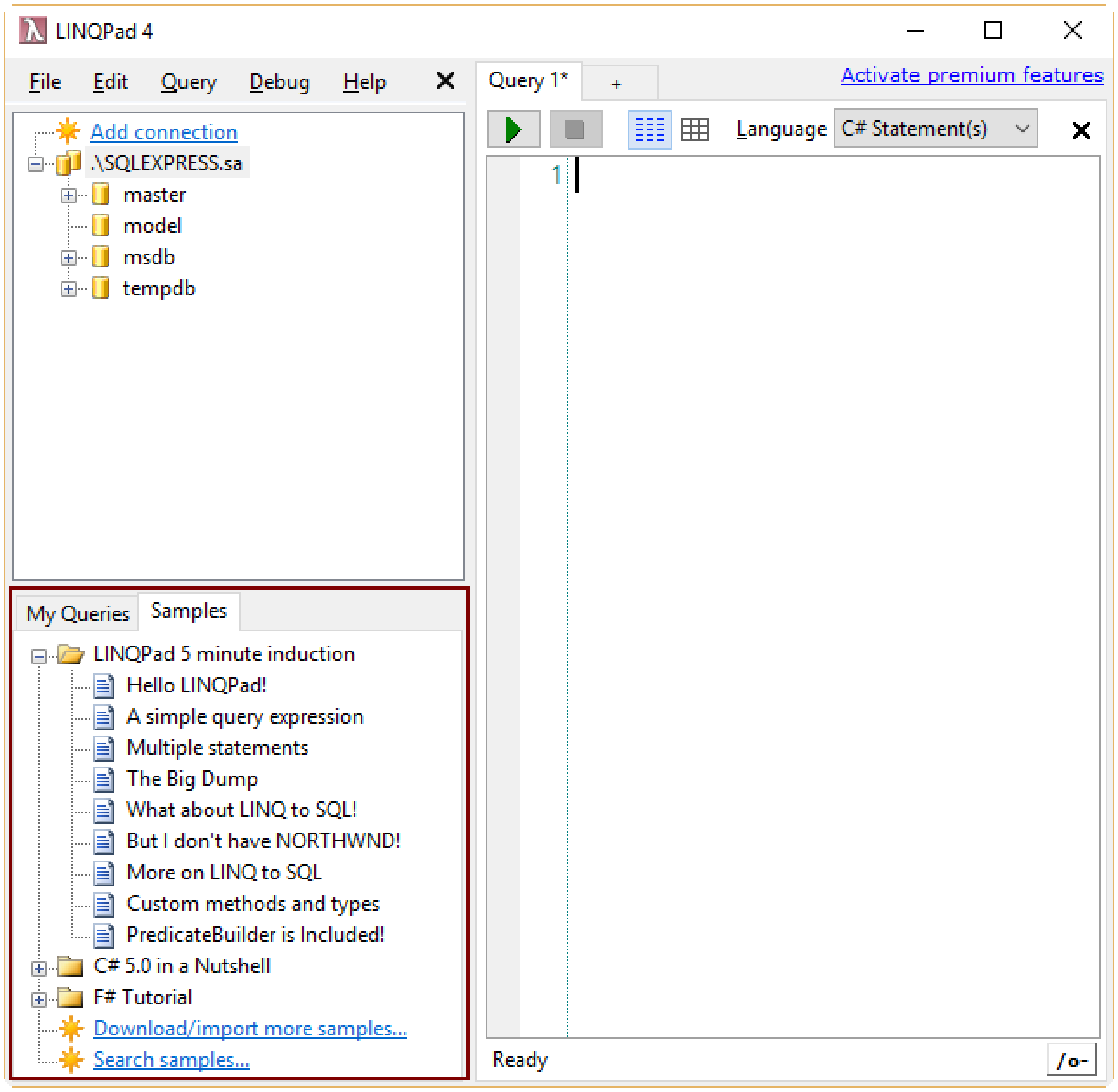


Figure 26: The Samples tab

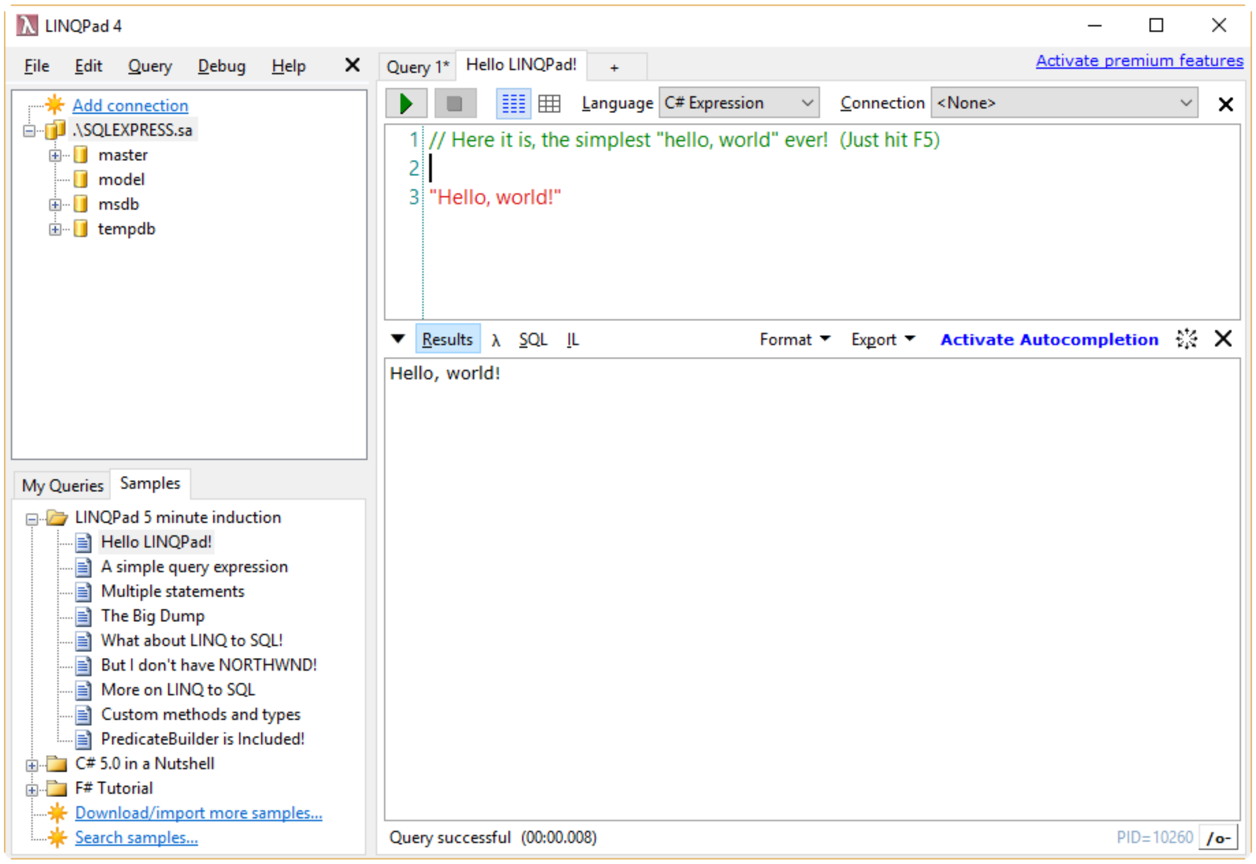
Referring to the previous figure, if you click the **Hello LINQPad!** sample, the following code will be displayed in the editor.

Code Listing 5

```
// Here it is, the simplest "hello, world" ever!  (Just hit F5)
"Hello, world!"
```

The previous code shows a simpler "Hello World" example than the one discussed in the "The ever-present Hello World" section in Chapter 3.

As the code suggests, the user should hit **F5** in order to execute the code. This is the equivalent of clicking on the **Execute Query** button.



```
// Here it is, the simplest "hello, world" ever!  (Just hit F5)

"Hello, world!"
```

```
Hello, world!
```

Query successful (00:00.008)

Figure 27: Output for the “Hello LINQPad!” sample

A LINQ-to-objects sample

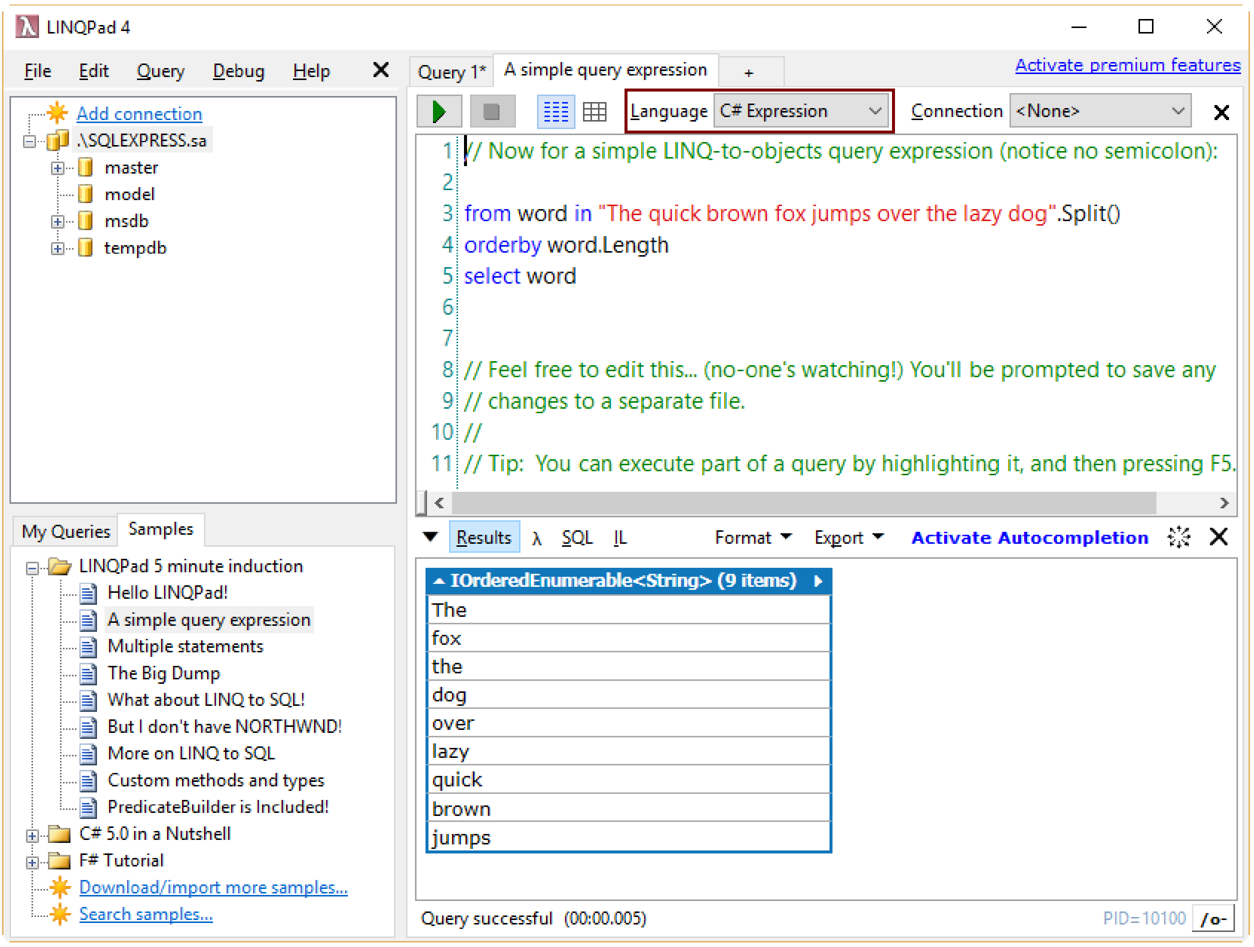
To demonstrate LINQ-to-objects support, the “A simple query expression” sample will be used. The code for this sample is the following.

Code Listing 6

```
// Now for a simple LINQ-to-objects query expression (notice no semicolon):

from word in "The quick brown fox jumps over the lazy dog".Split()
orderby word.Length
select word


// Feel free to edit this... (No-one’s watching!) You'll be prompted to save any
// changes to a separate file.
//
// Tip:  You can execute part of a query by highlighting it, and then pressing F5.
```

The code is essentially a one-line LINQ-to-objects query expression that returns a list of words from a string. The **`String Split`** method creates that list. Then, the **`orderby`** clause sorts the list based on every word’s length. In this case, the list is sorted beginning with the smallest word and ending with the largest one. Finally, the **`select`** statement iterates the list sequentially.

When the expression is executed, it produces the following output.



```
// Now for a simple LINQ-to-objects query expression (notice no semicolon):

from word in "The quick brown fox jumps over the lazy dog".Split()
orderby word.Length
select word


// Feel free to edit this... (no-one's watching!) You'll be prompted to save any
// changes to a separate file.
//
// Tip:  You can execute part of a query by highlighting it, and then pressing F5.
```

```
IOrderedEnumerable<String> (9 items)
The
fox
the
dog
over
lazy
quick
brown
jumps
```

Figure 28: Output for the "A simple query expression" sample

The Language Selector combo box displayed in the previous figure shows **C# Expression**, which is the type of language used for a one-line statement. In this case, no semicolon should be used at the end of the code line.

A multiple statements sample

LINQPad also can execute multiple statements at a time. The "Multiple statements" sample will be used to demonstrate this feature. The code for this sample is the following.

Code Listing 7

```
// Setting the query language to "C# Statement(s)" permits multiple statements:

var words =
        from word in "The quick brown fox jumps over the lazy dog".Split()
        orderby word.ToUpper()
        select word;
```

```
var duplicates =
       from word in words
       group word.ToUpper() by word.ToUpper() into g
       where g.Count() > 1
       select new { g.Key, Count = g.Count() };

// The Dump extension method writes out queries:

words.Dump();
duplicates.Dump();

// Notice that we do need semicolons now!
```

The first statement of this sample does almost the same job as the sample discussed in the previous section. But in this case, the list is sorted alphabetically, using the uppercase format for each word in it. After that, the list is stored in the words collection.

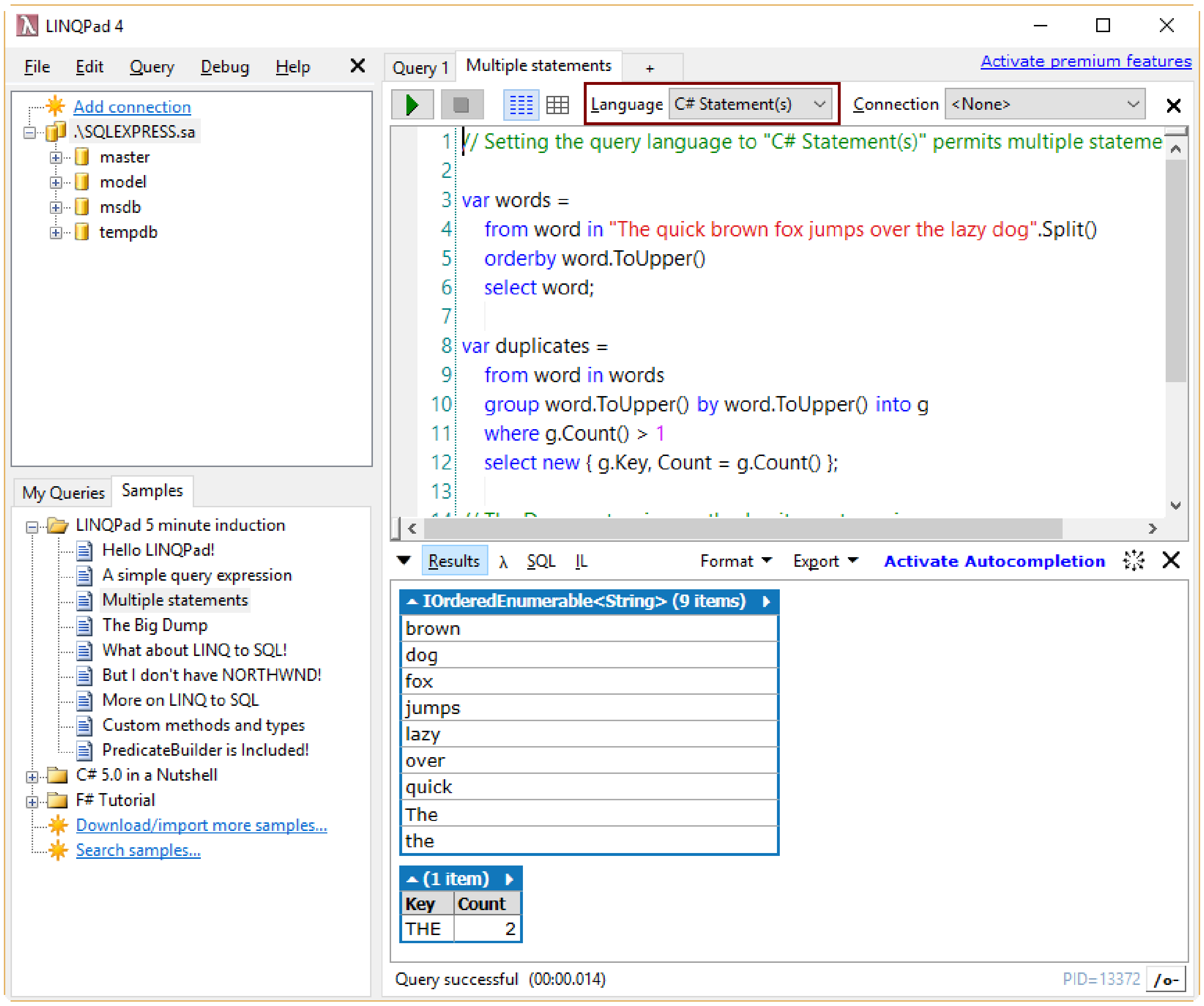
The next statement is intended to get a list of duplicate words along with how many of these words exists into the words collection. The **`from`** statement iterates the words list, and uses the **`group`** clause to gather those words in uppercase format that are exactly the same. The criteria used to create groups is the uppercase format of each one of them. Every group is stored into **`g`** variable as an IGrouping interface, which is a collection of data records of string type **`(word.ToUpper ())`**.

A special **`Key`** property is added. This property holds the grouped value (in this case, the uppercase format of each word) that is associated with all records in the collection. The **`where`** clause makes **`g`** available for the select clause only when the **`Count`** method of the collection returns a value greater than **`1`**. When this happens, a new **`<string, int>`** collection item is added to the **`duplicates`** collection, using the grouped value **`(g.Key)`** as a key for the item and storing the value returned by the **`g.Count`** method (which is actually the number of duplicates).

The **`Dump`** extension method (which will be explained in the following section) generates the output for each one of the objects.

As stated previously, semicolons need to be used in a multiple statements query. Also, **C# Statement(s)** must be selected from the **Language** combo box.

The execution of the query will display the following output.



Language C# Statement(s)
Connection <None>

```
// Setting the query language to "C# Statement(s)" permits multiple stateme

var words =
    from word in "The quick brown fox jumps over the lazy dog".Split()
    orderby word.ToUpper()
    select word;

var duplicates =
    from word in words
    group word.ToUpper() by word.ToUpper() into g
    where g.Count() > 1
    select new { g.Key, Count = g.Count() };
```

```
IOrderedEnumerable<String> (9 items)
brown
dog
fox
jumps
lazy
over
quick
The
the
```

(1 item)

Key	Count
THE	2

Query successful (00:00.014)   PID=13372

Figure 29: A multiple statements query

The Dump method

The **Dump** extension method is probably one of the most powerful features of LINQPad. The purpose of this method is to display the value(s) of almost any type of object, generating an output to be usually displayed on screen.

How Dump works

The **Dump** extension method is not a shortcut for .NET's **Console.WriteLine** method. Instead, it is a recursive method that generates a display for the values of objects, in HTML format. It supports collections, DataTables, Images, and other types. It can also format collections of complex objects into tables.

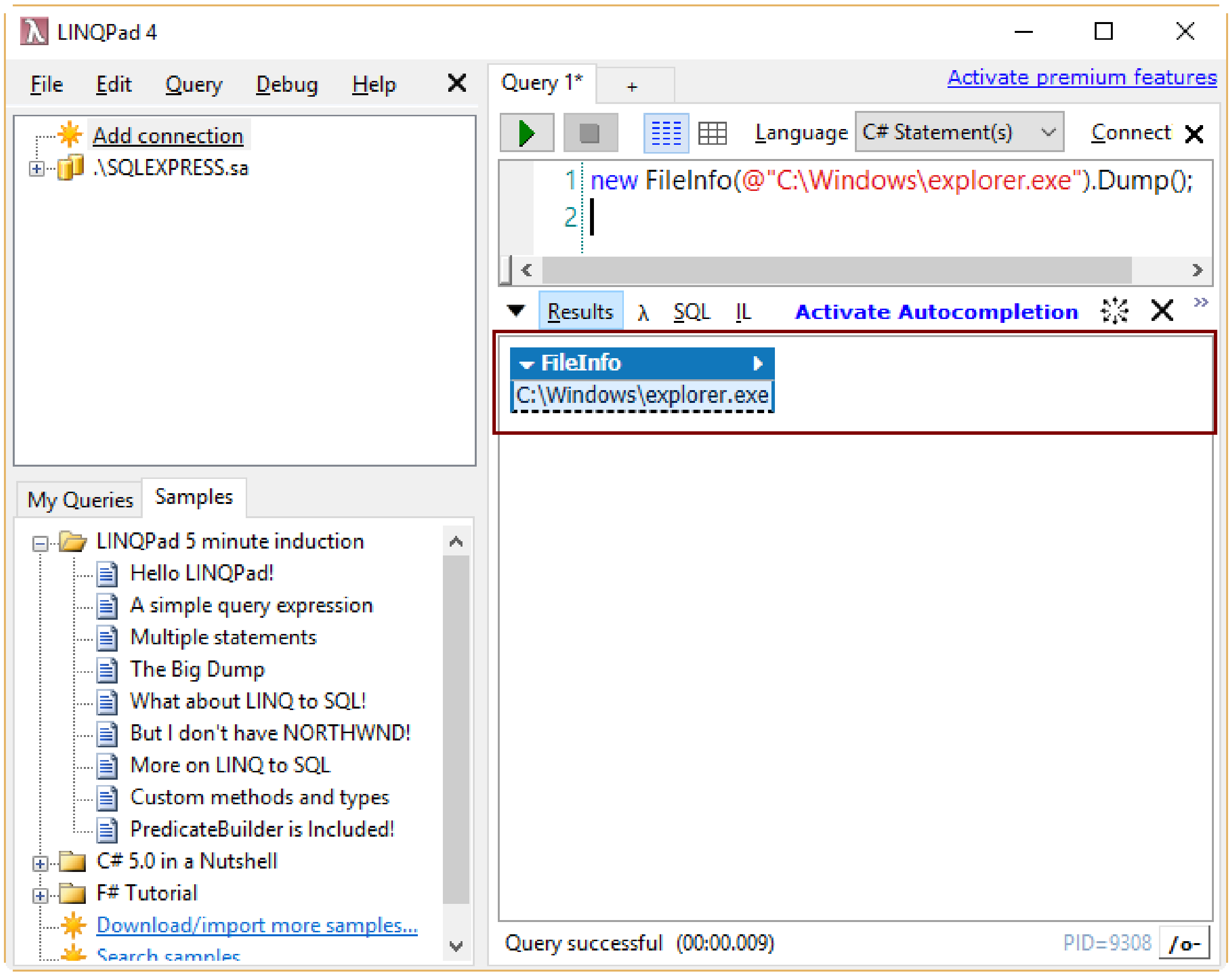
The **Dump** method knows how objects dumped are built, and what values are stored into them, by using the .NET Reflection mechanism. It then turns these objects into HTML and displays the result in a grid. This mechanism makes results more readable and provides the user with a great way to debug complex objects.

To demonstrate **Dump**, the user should select **C# Statement(s)** from the **Language** combo box, and then type the following code in the editor.

Code Listing

```
new FileInfo(@"C:\Windows\explorer.exe ").Dump();
```

When the previous code is executed, the output displayed will be the following.



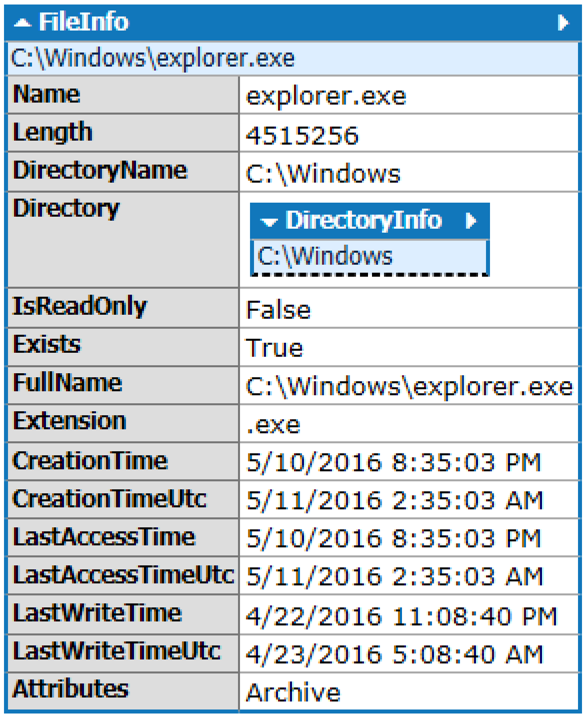
```
new FileInfo(@"C:\Windows\explorer.exe").Dump();
```

```
C:\Windows\explorer.exe
```

Query successful (00:00.009)   PID=9308

Figure 30: Dump method's output

Figure 30 shows an example of the **Dump** method's output. Note that the column header for the grid has a blue background. Also, a white down arrow is displayed at the left. This means that the output result can be expanded, getting more details on it.



FileInfo	
C:\Windows\explorer.exe	
Name	explorer.exe
Length	4515256
DirectoryName	C:\Windows
Directory	DirectoryInfo C:\Windows
IsReadOnly	False
Exists	True
FullName	C:\Windows\explorer.exe
Extension	.exe
CreationTime	5/10/2016 8:35:03 PM
CreationTimeUtc	5/11/2016 2:35:03 AM
LastAccessTime	5/10/2016 8:35:03 PM
LastAccessTimeUtc	5/11/2016 2:35:03 AM
LastWriteTime	4/22/2016 11:08:40 PM
LastWriteTimeUtc	4/23/2016 5:08:40 AM
Attributes	Archive

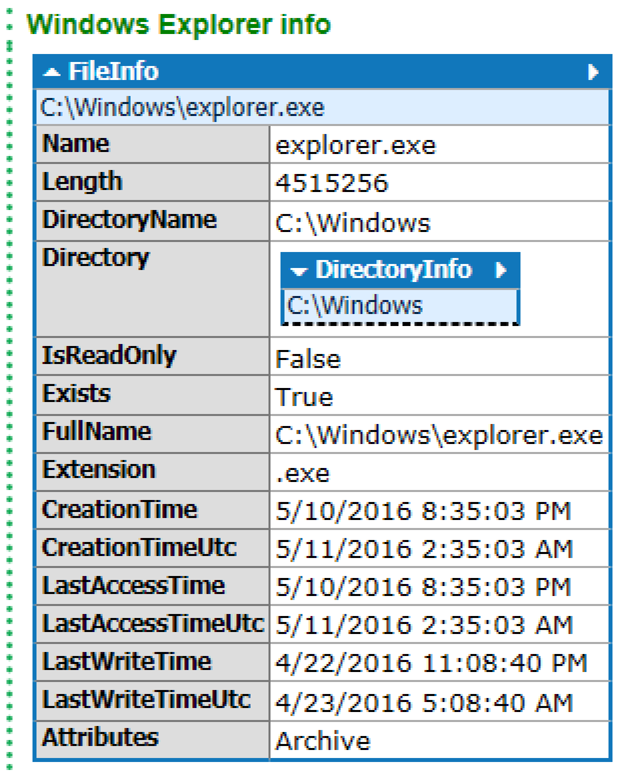
Figure 31: Dump method's output expanded

The **`Dump`** method allows you to pass a string parameter in order to display a description before the result. The following example demonstrates this feature.

Code Listing 8

```
new FileInfo(@"C:\Windows\explorer.exe ").Dump("Windows Explorer info");
```

When the previous code is executed, the following output will be displayed.



```
Windows Explorer info
FileInfo
C:\Windows\explorer.exe
Name                explorer.exe
Length              4515256
DirectoryName       C:\Windows
Directory           DirectoryInfo
                    C:\Windows
IsReadOnly          False
Exists              True
FullName            C:\Windows\explorer.exe
Extension           .exe
CreationTime        5/10/2016 8:35:03 PM
CreationTimeUtc     5/11/2016 2:35:03 AM
LastAccessTime      5/10/2016 8:35:03 PM
LastAccessTimeUtc   5/11/2016 2:35:03 AM
LastWriteTime       4/22/2016 11:08:40 PM
LastWriteTimeUtc    4/23/2016 5:08:40 AM
Attributes          Archive
```

Figure 32: Dump method's output with a description

Some issues with the Dump method

The **`Dump`** method works great on small collections and most objects. The user might notice long running times with big collections or some specific object types. Complex objects with many levels of hierarchy, or with a large number of properties, can be difficult to read into the result panel.

Chapter summary

This chapter was dedicated to showing basic concepts about LINQPad. One of these concepts is the term “query.” LINQPad uses this term for every piece of code written in the editor, and it allows us to save this code in code. All queries saved are displayed beneath a tree view node called My Queries.

LINQPad stores queries in a folder named LINQPad Queries, located into the user’s Documents folder. This behavior can be changed by clicking the **Set Folder** hyperlink, and browsing for a new folder in the **Preferences** dialog box. All queries discussed in this book will be saved in a folder named C:\LINQPad_Succinctly\Queries.

When the user works with a few queries, locating a specific one is simple. But when those queries increase in number, this can be hard work. To solve this issue, LINQPad allows you to search queries by using a term. This can be done by clicking on the **Go to** hyperlink located on the **My Queries** tree view node. Then, the user can type the term in the **Navigate to Query** dialog, and LINQPad will show all queries that match that term.

LINQPad can compile code in ten programming language variations. These variations are C# Expression, C# Statements, C# Program, VB Expression, VB Statement, VB Statements, VB Program, SQL, ESQL, F# Expression, and F# Program.

All query filenames are associated with the .linq file extension. Internally, a header in XML format is added at the beginning of the file. This header tells LINQPad which kind of programming language and SQL connection are used by the query.

LINQPad allows you to manage connections to SQL servers in the Connection’s tree view area by clicking on the **Add Connection** hyperlink. Then, the Choose Data Context dialog box displays, which allows the user to choose which LINQPad driver will be used. For purposes of this book, we’ll use the LINQ to SQL driver (which is the default).

After the LINQPad driver is chosen, the LINQPad Connection dialog box is shown to set the connection’s parameters. A Microsoft SQL Server connection will be created, assuming that a SQLEXPRESS instance configured with SQL Authentication security mode is installed in the local computer. Also, the default Data Context options will be used for the connection.

LINQPad names a connection using the server name followed by a period and the name of the authenticated user, when SQL Authentication is chosen. The name displayed for the connection created will be .\SQLEXPRESS.sa. When Windows Authentication is selected, the name for the connection will contain the server name only.

When the connection is displayed in the user interface, a context menu is available when the user right-clicks on the connection’s name. This menu contains a set of commands for managing that connection.

A few LINQPad queries samples were discussed in this chapter, too. The discussion starts with the traditional “Hello World” sample, and continues with a LINQ-to-objects and a multiple statements examples.

The last section of this chapter explains the `Dump` extension method, which is one of the most powerful features of LINQPad. This method renders almost any type of object and turns it into HTML by using reflection, and then displays the result in a grid.

The `Dump` method works great on small collections and most objects, but when big collections are processed, the user might notice long running times.

Chapter 5 Querying Databases with LINQ-to-SQL

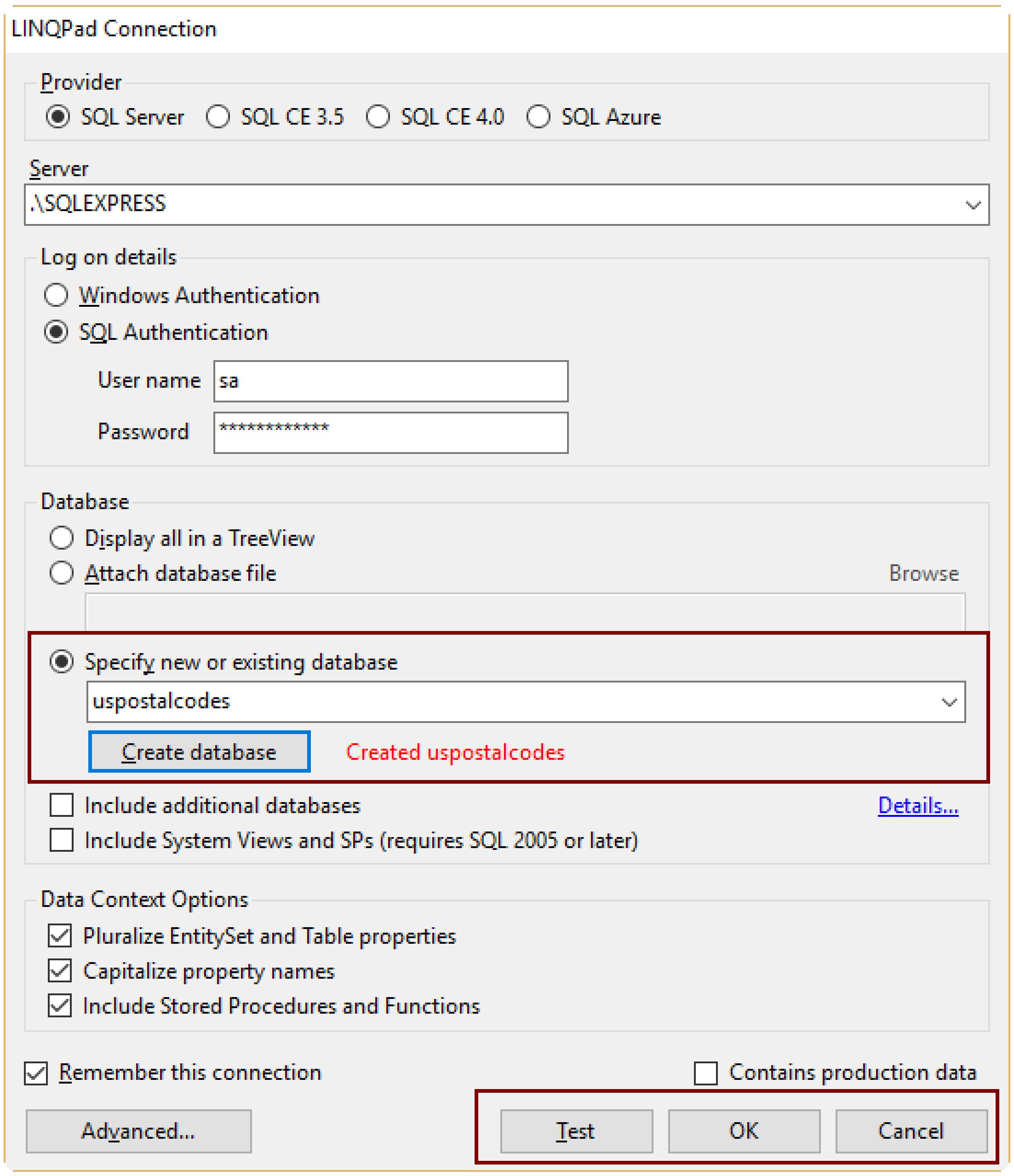
This chapter will explain how to perform database queries with LINQPad. The default LINQ-to-SQL driver will be used to run those queries. All samples explained in this chapter use a database that will be populated with all United States postal codes. The name given to this database will be **uspostalcodes**.

Creating a database from LINQPad

A SQL database can be created from LINQPad, as explained in the "Managing SQL connections" section of Chapter 4. 2.

To create the **uspostalcodes** database, the user should click on the **Add connection** hyperlink, and then select the **LINQ-to-SQL** driver from the **Choose Data Context** dialog box. Click **Next** to bring up the **LINQ to SQL Connection** dialog box, where you can create the database.

To create the **uspostalcodes** database (after the rest of connection parameters are chosen), select the **Specify new or existing database** option from the **Database** options group, and then type the name **uspostalcodes** into the combo box located below this option. The **Create database** button will be enabled after name's typing is finished; click this button to make LINQPad create the database. LINQPad will display the message "Created uspostalcodes" beside the Create database button after database creation. Click **OK** to save the connection info and make it appear in the user interface.


LINQPad Connection

Provider: (●) SQL Server ( ) SQL CE 3.5 ( ) SQL CE 4.0 ( ) SQL Azure

Server: .\SQLEXPRESS

Log on details
( ) Windows Authentication
(●) SQL Authentication
User name: sa
Password: ************

Database
( ) Display all in a TreeView
( ) Attach database file
(●) Specify new or existing database
uspostalcodes
Created uspostalcodes

☐ Include additional databases
☐ Include System Views and SPs (requires SQL 2005 or later)

Data Context Options
☑ Pluralize EntitySet and Table properties
☑ Capitalize property names
☑ Include Stored Procedures and Functions

☑ Remember this connection
☐ Contains production data

Figure 33: LINQPad Connection dialog box appears after database creation

Creating database's entities

If you look at the connection displayed in the user interface, you will notice that no entity appears in the tree view. This means that LINQPad created the database only. Entities creation for the database is up to the user. Unfortunately, there's no easy way to create database's entities with LINQ-to-SQL. For this book, we'll use Microsoft SQL Server Management Studio.

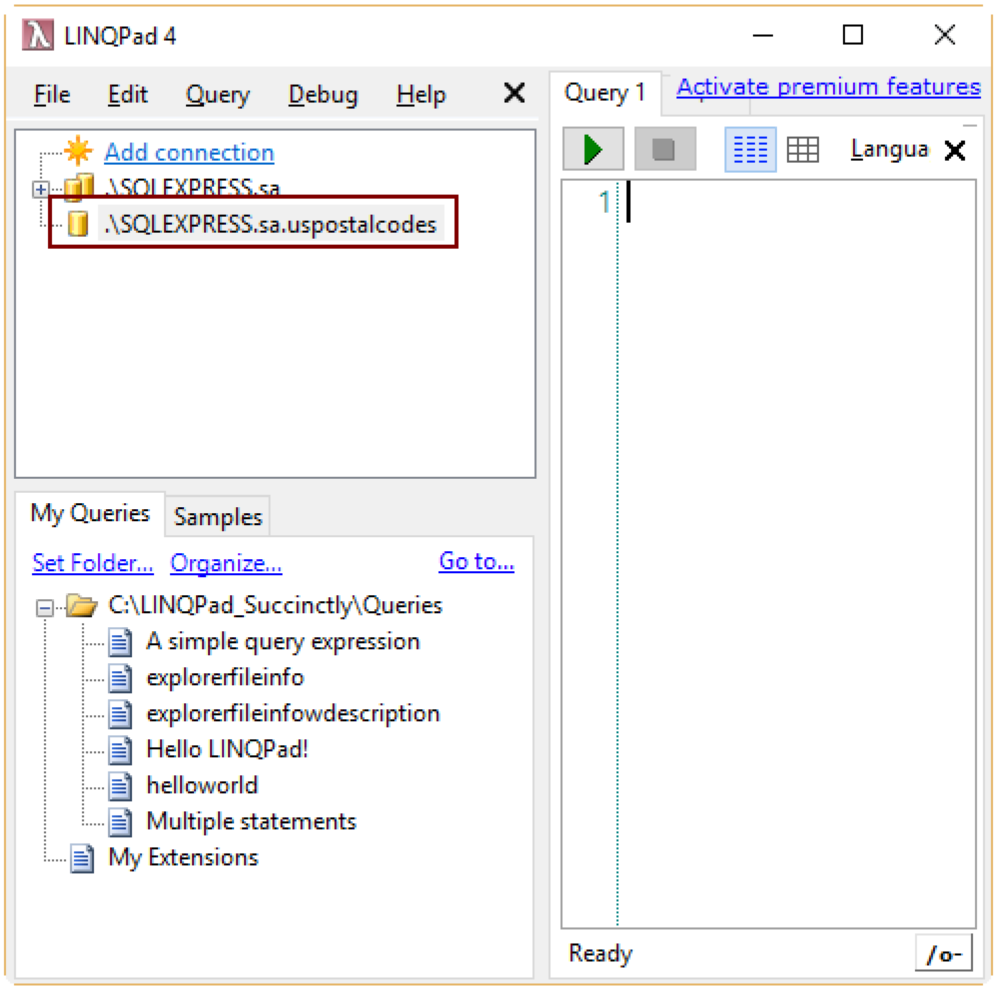


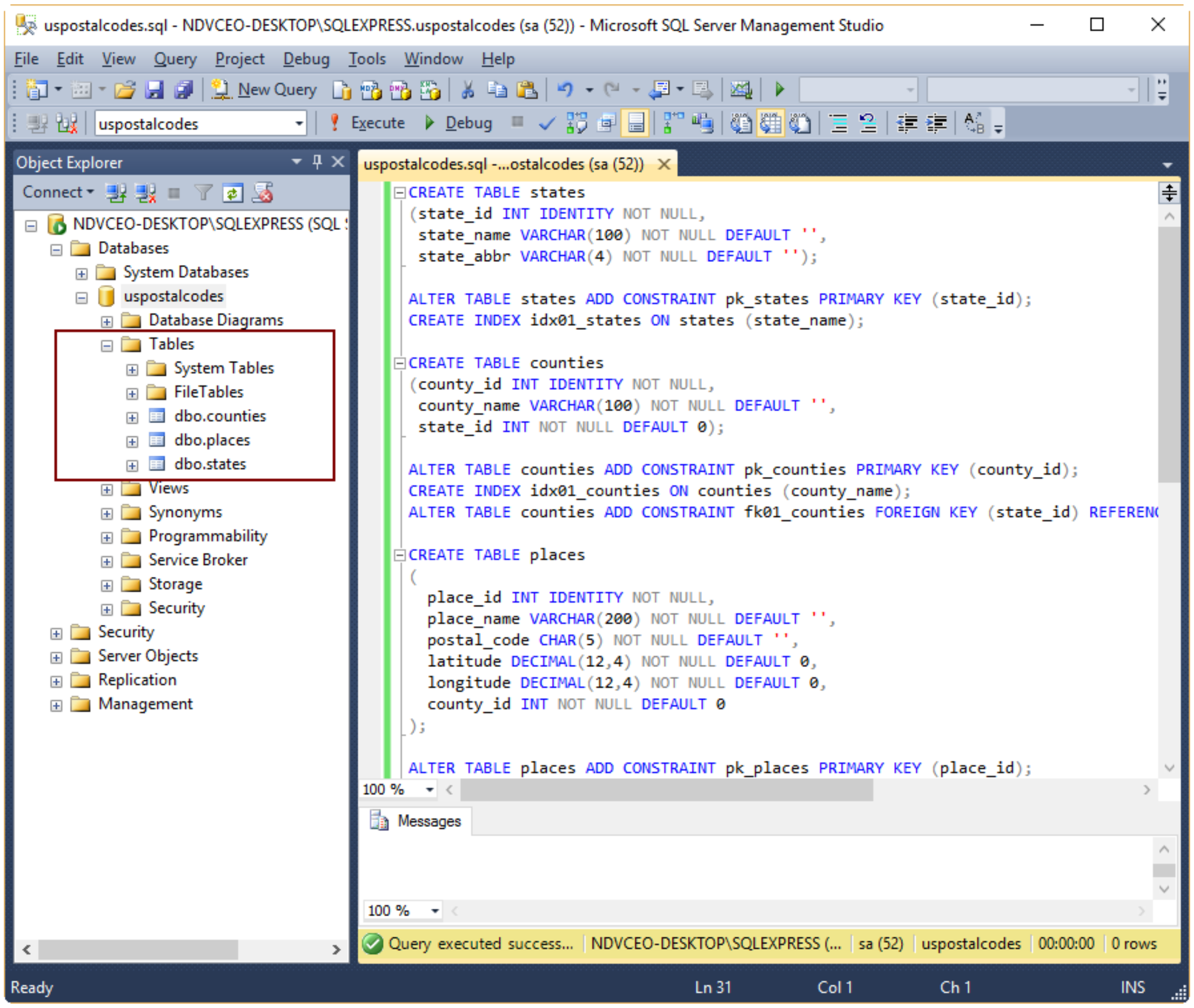
Figure 34: LINQPad user interface with uspostalcodes connection

You can launch SQL Server Management Studio from the **Run** dialog by typing **ssms**. Then, you should log in into SQL Server Management Studio and select the **uspostalcodes** database in the **Object Explorer**. Then, click the **New Query** button located in the toolbar to open a query tabbed window. Type and execute the following code.

Code Listing 9

```
CREATE TABLE states
(state_id INT IDENTITY NOT NULL,
 state_name VARCHAR(100) NOT NULL DEFAULT '',
 state_abbr VARCHAR(4) NOT NULL DEFAULT '');

ALTER TABLE states ADD CONSTRAINT pk_states PRIMARY KEY (state_id);
CREATE INDEX idx01_states ON states (state_name);

CREATE TABLE counties
(county_id INT IDENTITY NOT NULL,
 county_name VARCHAR(100) NOT NULL DEFAULT '',
 state_id INT NOT NULL DEFAULT 0);
```

```
ALTER TABLE counties ADD CONSTRAINT pk_counties PRIMARY KEY (county_id);
CREATE INDEX idx01_counties ON counties (county_name);
ALTER TABLE counties ADD CONSTRAINT fk01_counties FOREIGN KEY (state_id) REFERENCES
states (state_id);

CREATE TABLE places
(
  place_id INT IDENTITY NOT NULL,
  place_name VARCHAR(200) NOT NULL DEFAULT '',
  postal_code CHAR(5) NOT NULL DEFAULT '',
  latitude DECIMAL(12,4) NOT NULL DEFAULT 0,
  longitude DECIMAL(12,4) NOT NULL DEFAULT 0,
  county_id INT NOT NULL DEFAULT 0
);

ALTER TABLE places ADD CONSTRAINT pk_places PRIMARY KEY (place_id);
CREATE INDEX idx01_places ON places (place_name);
ALTER TABLE places ADD CONSTRAINT fk01_places FOREIGN KEY (county_id) REFERENCES
counties (county_id);
```



```
CREATE TABLE states
(state_id INT IDENTITY NOT NULL,
 state_name VARCHAR(100) NOT NULL DEFAULT '',
 state_abbr VARCHAR(4) NOT NULL DEFAULT '');

ALTER TABLE states ADD CONSTRAINT pk_states PRIMARY KEY (state_id);
CREATE INDEX idx01_states ON states (state_name);

CREATE TABLE counties
(county_id INT IDENTITY NOT NULL,
 county_name VARCHAR(100) NOT NULL DEFAULT '',
 state_id INT NOT NULL DEFAULT 0);

ALTER TABLE counties ADD CONSTRAINT pk_counties PRIMARY KEY (county_id);
CREATE INDEX idx01_counties ON counties (county_name);
ALTER TABLE counties ADD CONSTRAINT fk01_counties FOREIGN KEY (state_id) REFERENC

CREATE TABLE places
(
  place_id INT IDENTITY NOT NULL,
  place_name VARCHAR(200) NOT NULL DEFAULT '',
  postal_code CHAR(5) NOT NULL DEFAULT '',
  latitude DECIMAL(12,4) NOT NULL DEFAULT 0,
  longitude DECIMAL(12,4) NOT NULL DEFAULT 0,
  county_id INT NOT NULL DEFAULT 0
);

ALTER TABLE places ADD CONSTRAINT pk_places PRIMARY KEY (place_id);
```

Figure 35: Uspostalcodes database with entities created, displayed in SQL Server Management Studio

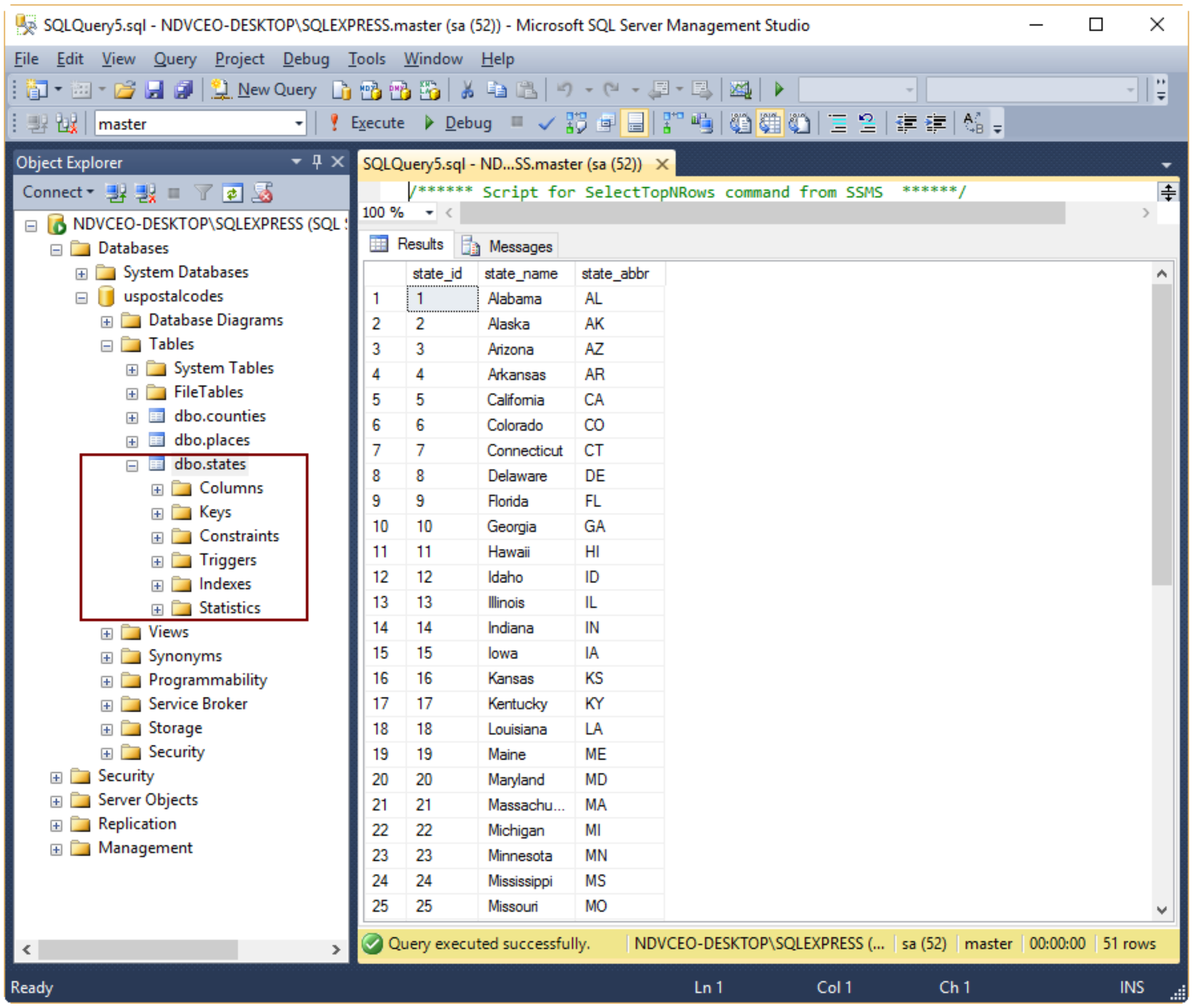
Populating a database

Now, the database needs to be populated with data. Three datasets will be used.

- A SQL script with all country states.
- A SQL script with all counties belonging to each state.
- A SQL script with all places, postal codes, and their geographic references.

Due the length limitations of this book, it's not possible to display these scripts here. These three scripts (**`states:table,sql`**, **`us_counties.sql`**, **`us_places.sql`**) are available in the code samples package, which can be downloaded here.

The following figure shows the states table after it has been populated with the scripts.



```
/****** Script for SelectTopNRows command from SSMS  ******/
```

	state_id	state_name	state_abbr
1	1	Alabama	AL
2	2	Alaska	AK
3	3	Arizona	AZ
4	4	Arkansas	AR
5	5	California	CA
6	6	Colorado	CO
7	7	Connecticut	CT
8	8	Delaware	DE
9	9	Florida	FL
10	10	Georgia	GA
11	11	Hawaii	HI
12	12	Idaho	ID
13	13	Illinois	IL
14	14	Indiana	IN
15	15	Iowa	IA
16	16	Kansas	KS
17	17	Kentucky	KY
18	18	Louisiana	LA
19	19	Maine	ME
20	20	Maryland	MD
21	21	Massachu...	MA
22	22	Michigan	MI
23	23	Minnesota	MN
24	24	Mississippi	MS
25	25	Missouri	MO

```
Query executed successfully. | NDVCEO-DESKTOP\SQLEXPRESS (... | sa (52) | master | 00:00:00 | 51 rows
```

Figure 36: States table, after being populated with data

Displaying database changes in LINQPad

As discussed in the “The Connection context menu” section of Chapter 4, a set of context menu’s commands are available for any connection displayed in the LINQPad user interface. In order to view all changes made to the **uspostalcodes** database, the user should right-click anywhere in the window, then select the **Refresh** command from the context menu. Now, the entities created in SQL Server Management Studio will appear on the screen.

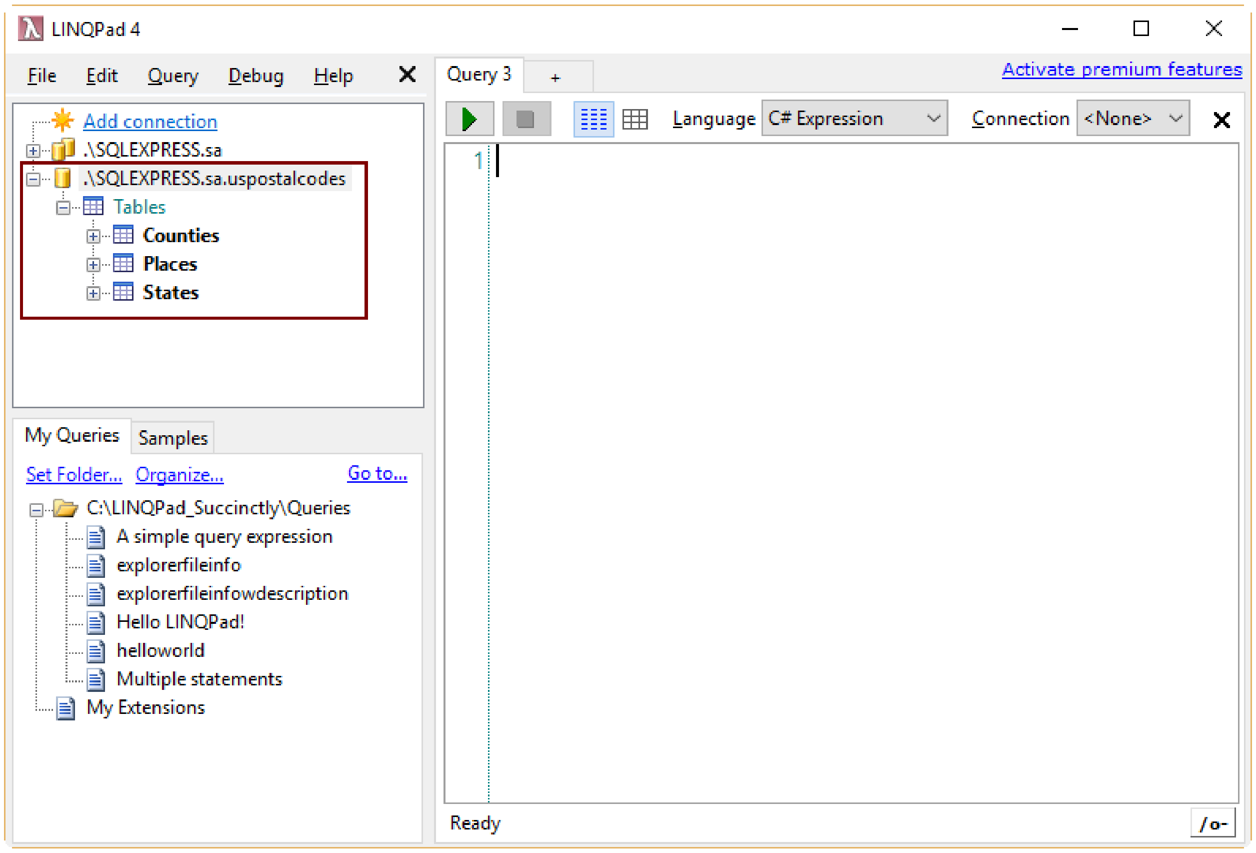


Figure 37: Uspostalcodes database entities, displayed in LINQPad user interface

Retrieving rows from a table

At this point, it's assumed that all tables are filled up with data. However, all the examples discussed in this section use the **`States`** table.

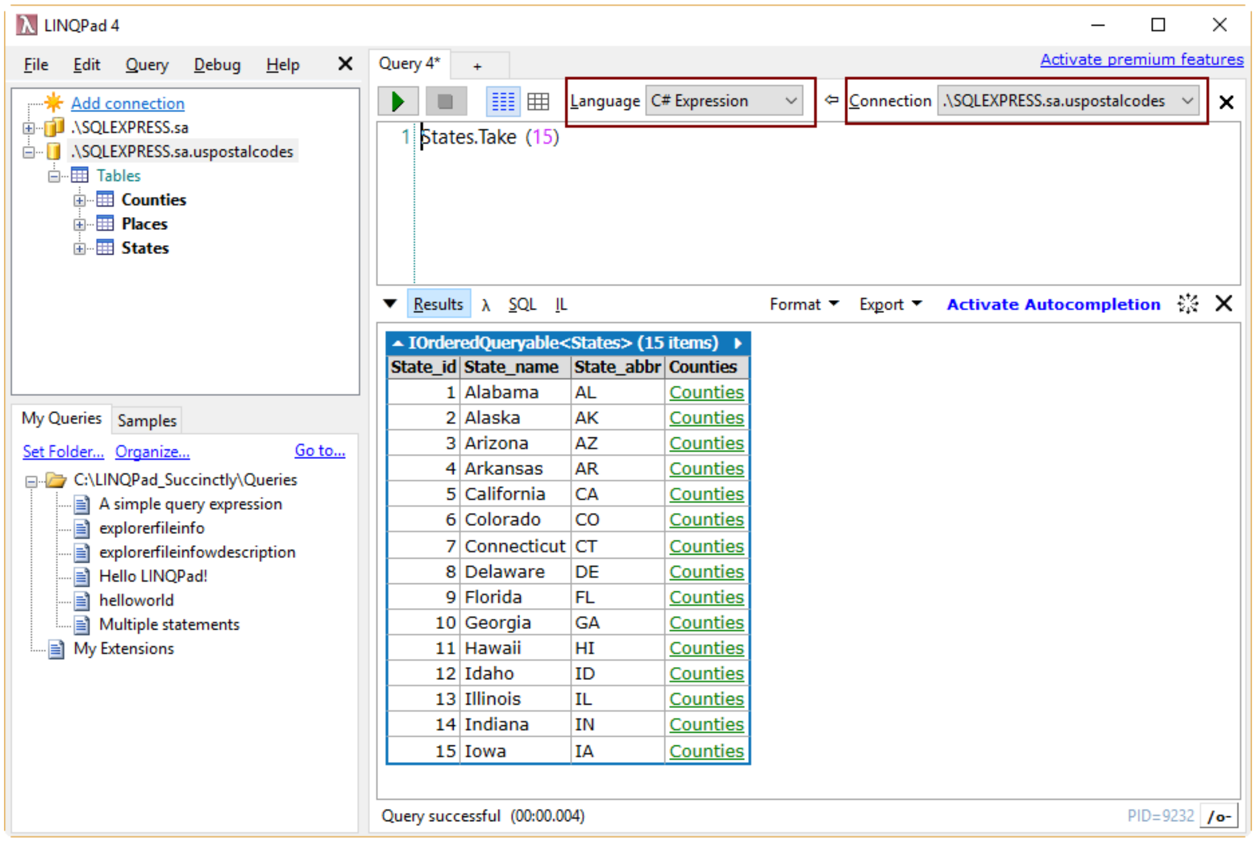
Returning a certain number of rows

The **`Take`** method returns a row set that is as large as the integer value passed as a parameter. To get the first 15 rows from the **`States`** table, select **C# Expression** from the **Language** selector, and select the **.\SQLEXPRESS.sa.uspostalcodes** connection from the **Connection** selector. After that, type and execute the following code.

Code Listing 10

```
States.Take(15)
```

The output for the previous code is displayed in Figure 38.



```
States.Take (15)
```

IOrderedQueryable<States> (15 items)

State_id	State_name	State_abbr	Counties
1	Alabama	AL	Counties
2	Alaska	AK	Counties
3	Arizona	AZ	Counties
4	Arkansas	AR	Counties
5	California	CA	Counties
6	Colorado	CO	Counties
7	Connecticut	CT	Counties
8	Delaware	DE	Counties
9	Florida	FL	Counties
10	Georgia	GA	Counties
11	Hawaii	HI	Counties
12	Idaho	ID	Counties
13	Illinois	IL	Counties
14	Indiana	IN	Counties
15	Iowa	IA	Counties

Query successful (00:00.004)

Language: C# Expression

Connection: .\SQLEXPRESS.sa.uspostalcodes

Figure 38: The first 15 rows from States table

Even though the **States** table has only three columns, note that in the previous figure a fourth column with the value of **Counties** is displayed in the result. Reviewing the script that creates all entities in the database, you can see that there's a SQL statement with a foreign key definition that uses the **Counties** table. This foreign key references the table **States** using the column named **State_id**. So, when the **Take** method is executed, LINQ-to-SQL finds the reference automatically, and LINQPad creates a hyperlink to the **Counties** table. This hyperlink is created for each row retrieved by the method. If the user clicks on any of these hyperlinks, LINQPad will display a row set from the **Counties** table where the **State_id** column value matches in both the **States** and **Counties** tables.

Note: When using LINQ-to-SQL, LINQPad automatically finds all references for the tables that are being part of the query. This means that when the user clicks on a related table hyperlink and the rows for that table are displayed, a nested reference hyperlink could be shown if another relation is found.

Using lambda expressions

The term *lambda expression* comes from lambda calculus, which is a mathematical notation applied for defining functions. These expressions are created using the following syntax.

Code Listing 11

```
(input parameters) => statement block or expression
```

An example of a lambda expression is displayed in the following code snippet.

Code Listing 12

```
y => y * 3
```

In the previous sample, the lambda expression specifies a parameter named **`y`**, and the statement for this expression multiplies the value of the parameter by **`3`**.

When a lambda expression is used within a query operator, it is evaluated by this query operator upon demand. This Boolean query evaluation is applied to all those elements in the input sequence, and only those elements that meet the filter condition are passed into the appended function. The following sample fetches the odd values from a given array and calculates and average for all of them.

Code Listing 13

```
int[] numbers = {1,1,2,3,5,8,13,21,34};
double average = numbers.Where(number => number % 2 == 1).Average();
Console.WriteLine(average);
```

In the previous example, the lambda expression within the **`Where`** method is evaluated for all numbers in the list. Each one of these numbers is used to obtain the remainder after being divided by two. If the result of the operation is equal to **`1`**, the number evaluated will be considered for the **`Average`** method. The result obtained is displayed in the following snippet.

Code Listing 14

```
7.3333333
```

Sorting rows

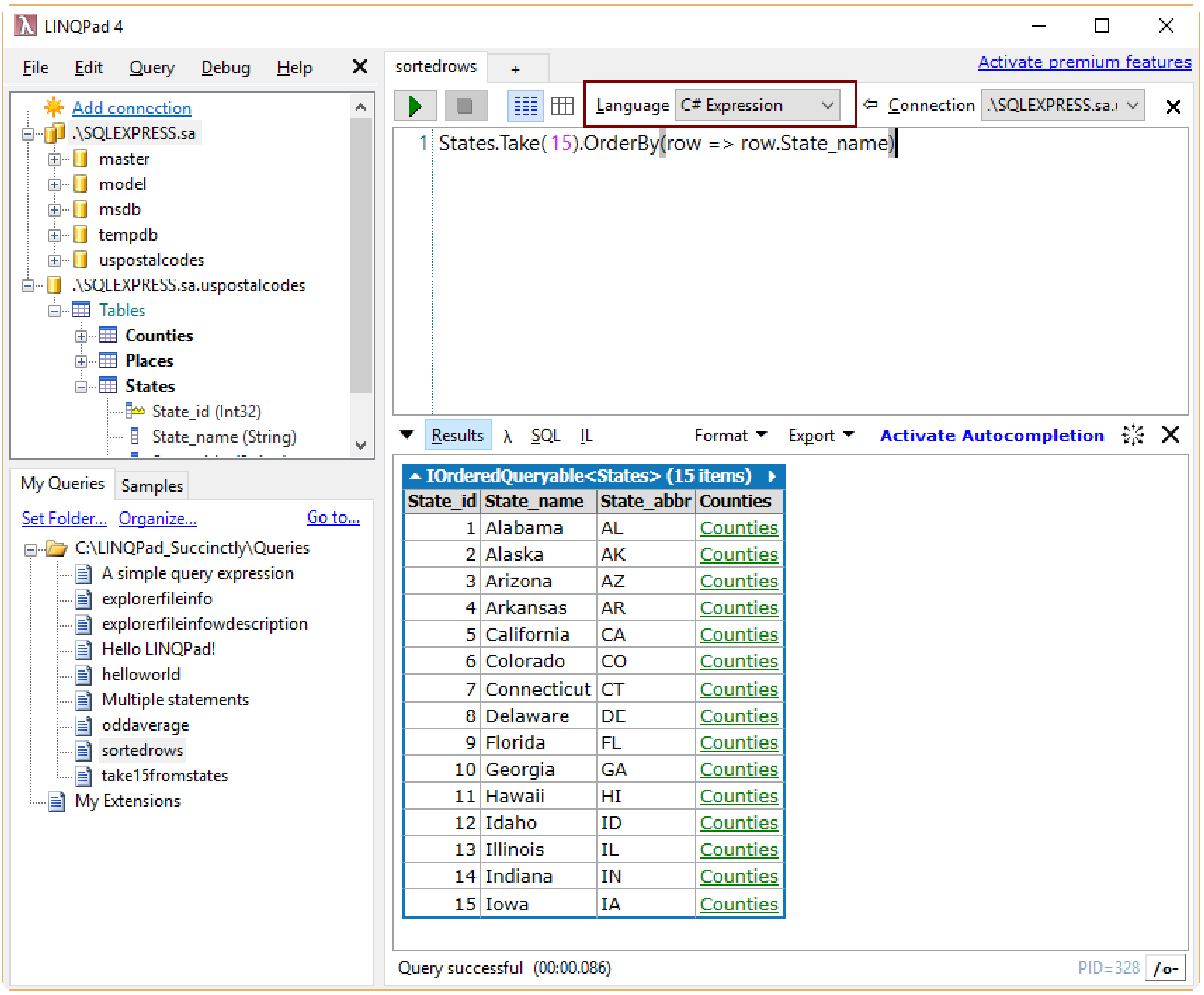
As in the **`Average`** calculation sample, a lambda expression can be used by LINQ-to-SQL query operators. The following code takes the first 15 rows from the **`States`** table and sort those rows using the values stored in the **`State_name`** column.

Code Listing 15

```
States.Take(15).OrderBy(row => row.State_name)
```

In this case, the lambda expression used in the **`OrderBy`** method takes a parameter named **`row`**, which receives each of the 15 rows returned by the **`Take`** method, in a one-by-one sequence. The statement executed in the expression sends the value of the **`State_name`** column to the **`OrderBy`** method for each row. When the row sequence ends, the **`OrderBy`** method performs an alphabetical sort for all of them, based on the values received.

The output for the previous code is shown in Figure 39.



Language: C# Expression

```
States.Take( 15).OrderBy(row => row.State_name)
```

IOrderedQueryable<States> (15 items)

State_id	State_name	State_abbr	Counties
1	Alabama	AL	Counties
2	Alaska	AK	Counties
3	Arizona	AZ	Counties
4	Arkansas	AR	Counties
5	California	CA	Counties
6	Colorado	CO	Counties
7	Connecticut	CT	Counties
8	Delaware	DE	Counties
9	Florida	FL	Counties
10	Georgia	GA	Counties
11	Hawaii	HI	Counties
12	Idaho	ID	Counties
13	Illinois	IL	Counties
14	Indiana	IN	Counties
15	Iowa	IA	Counties

Query successful (00:00.086)

Figure 39: The result for the query with rows sorted

Using aggregate functions

LINQPad allows you to use SQL aggregate functions such as **Count**, **Sum**, or **Avg** (average), as a LINQ-to-SQL query methods. Each one of these methods is treated as an extension method for each table in the model. Assuming that all tables have already been populated with data, the following sample will retrieve the total row count for the **Places** table.

Code Listing 16

```
Places.Count()
```

The output displayed should look like the following snippet.

Code Listing 17

```
43563
```

Lambda expressions can also be used to limit the results returned by a query. For example, if you need to know the number of places belonging to the county with the **id 1926**, you should execute the following code.

Code Listing 18

```
Places.Count(row => row.County_id == 1926)
```

Again, the lambda expression takes a row argument and executes a Boolean comparison statement, which will return **true** for every row that has a value of **1926** in the **County_id** column. Every time a true value is returned, the **Count** method increments its result by **1**. If there's no row that matches the Boolean statement, LINQPad returns nothing.

The output displayed should be the following:

Code Listing 19

```
24
```

Note: The Sum and Average (Avg) query methods require the use of lambda expressions.

Filtering and grouping rows

LINQPad allows you to filter and group rows with LINQ-to-SQL expressions using the following methods.

- **Where**: Takes a lambda expression that returns a Boolean **true** or **false** value, which will tell the method which elements must be included in the query's result.
- **GroupBy**: Takes a lambda expression that returns a value to group the rows. Then, returns an **IQueryable** collection of **IGrouping** objects. That is, a collection of groups. A special property named **Key** is added to each one of these groups. This property holds the value used to group the rows. The group itself is a collection of data rows taken from the entity that uses the method.

An example for filtering rows

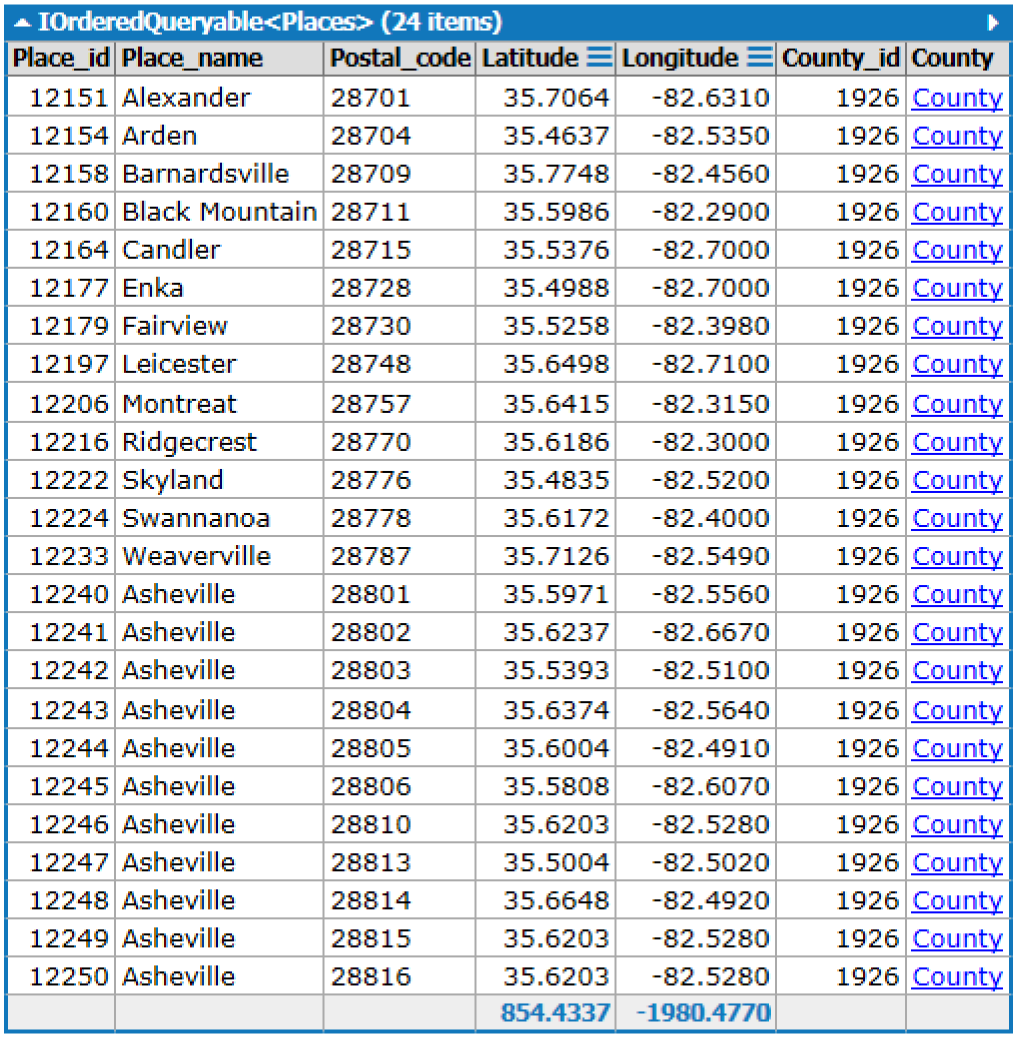
Assuming that a list of all places belonging to a county with the **id 1926** is required, the following code should be executed.

Code Listing 20

```
Places.Where(row => row.County_id == 1926)
```

The **Where** method will iterate the **Places** table in a sequence, and the lambda expression will receive each row as a parameter. Then, it will return a **true** value for those rows where the value of **County_id** column is equal to **1926**. A **true** value tells the **Where** method to include the row in the query's result. A **false** value will cause the **Where** method to discard the row.

The result for the previous code is shown in Figure 40.



IOrderedQueryable<Places> (24 items)

Place_id	Place_name	Postal_code	Latitude	Longitude	County_id	County
12151	Alexander	28701	35.7064	-82.6310	1926	County
12154	Arden	28704	35.4637	-82.5350	1926	County
12158	Barnardsville	28709	35.7748	-82.4560	1926	County
12160	Black Mountain	28711	35.5986	-82.2900	1926	County
12164	Candler	28715	35.5376	-82.7000	1926	County
12177	Enka	28728	35.4988	-82.7000	1926	County
12179	Fairview	28730	35.5258	-82.3980	1926	County
12197	Leicester	28748	35.6498	-82.7100	1926	County
12206	Montreat	28757	35.6415	-82.3150	1926	County
12216	Ridgecrest	28770	35.6186	-82.3000	1926	County
12222	Skyland	28776	35.4835	-82.5200	1926	County
12224	Swannanoa	28778	35.6172	-82.4000	1926	County
12233	Weaverville	28787	35.7126	-82.5490	1926	County
12240	Asheville	28801	35.5971	-82.5560	1926	County
12241	Asheville	28802	35.6237	-82.6670	1926	County
12242	Asheville	28803	35.5393	-82.5100	1926	County
12243	Asheville	28804	35.6374	-82.5640	1926	County
12244	Asheville	28805	35.6004	-82.4910	1926	County
12245	Asheville	28806	35.5808	-82.6070	1926	County
12246	Asheville	28810	35.6203	-82.5280	1926	County
12247	Asheville	28813	35.5004	-82.5020	1926	County
12248	Asheville	28814	35.6648	-82.4920	1926	County
12249	Asheville	28815	35.6203	-82.5280	1926	County
12250	Asheville	28816	35.6203	-82.5280	1926	County
			854.4337	-1980.4770		

Figure 40: The filtered query result

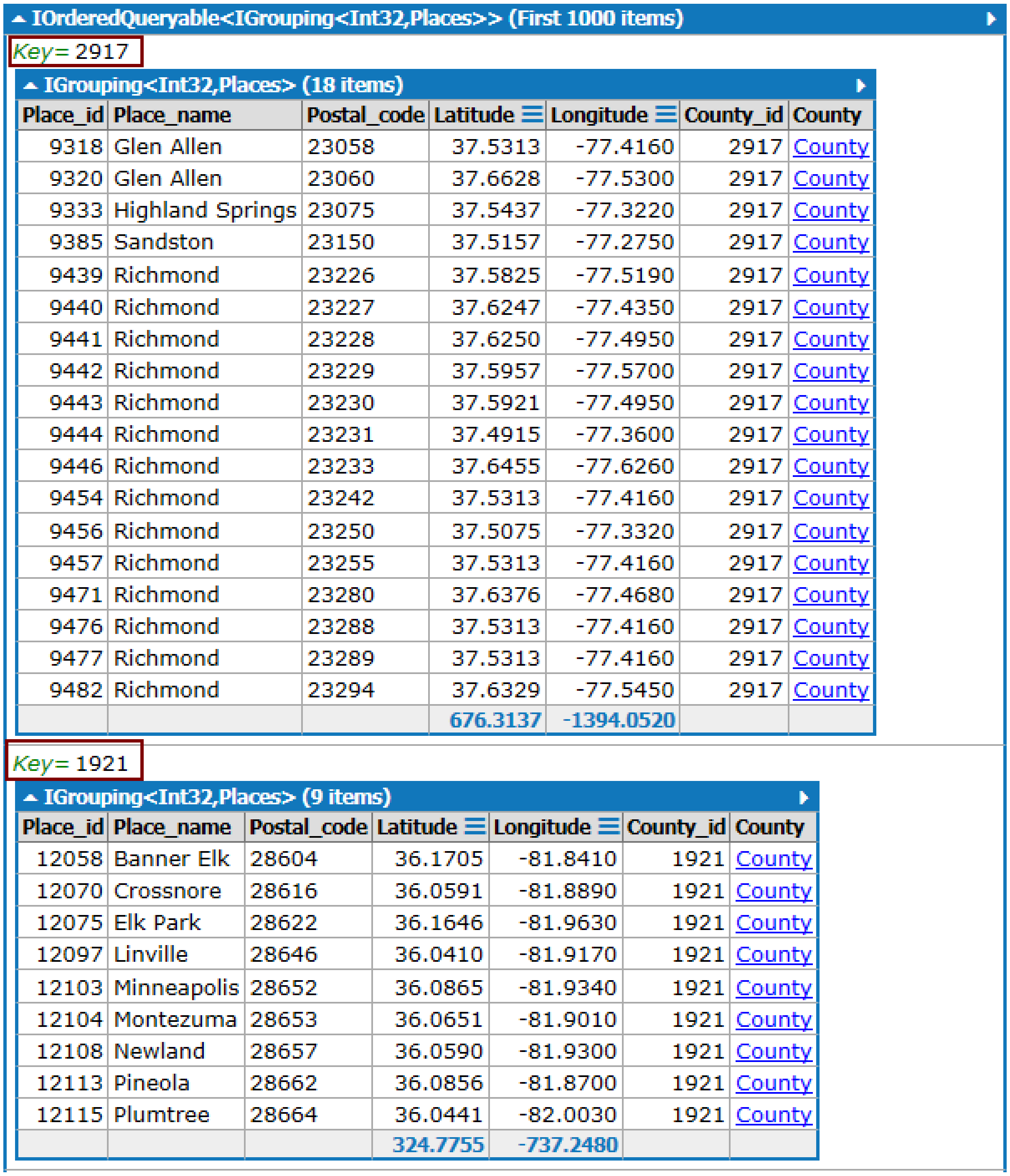
An example for grouping rows

The example explained in the previous section returns a row set based on the value for the **`County_id`** column. Now, it's going to be assumed that the user wants to retrieve all rows from the **`Places`** table, but grouped by county. This can be accomplished executing the following code.

Code Listing 21

```
Places.GroupBy(row => row.County_id)
```

The example uses the **`GroupBy`** method for the **`Places`** table. The method takes a lambda expression that returns the value for the **`County_id`** column. This value will be used to group the rows for the entire table. So, all rows that have the same value for **`County_id`** will be gathered in the same collection. The total number of collections depends on how many different values for **`County_id`** are in the table. Figure 41 shows a partial result for the query.


IOrderedQueryable<IGrouping<Int32,Places>> (First 1000 items)

Key= 2917

IGrouping<Int32,Places> (18 items)

Place_id	Place_name	Postal_code	Latitude	Longitude	County_id	County
9318	Glen Allen	23058	37.5313	-77.4160	2917	County
9320	Glen Allen	23060	37.6628	-77.5300	2917	County
9333	Highland Springs	23075	37.5437	-77.3220	2917	County
9385	Sandston	23150	37.5157	-77.2750	2917	County
9439	Richmond	23226	37.5825	-77.5190	2917	County
9440	Richmond	23227	37.6247	-77.4350	2917	County
9441	Richmond	23228	37.6250	-77.4950	2917	County
9442	Richmond	23229	37.5957	-77.5700	2917	County
9443	Richmond	23230	37.5921	-77.4950	2917	County
9444	Richmond	23231	37.4915	-77.3600	2917	County
9446	Richmond	23233	37.6455	-77.6260	2917	County
9454	Richmond	23242	37.5313	-77.4160	2917	County
9456	Richmond	23250	37.5075	-77.3320	2917	County
9457	Richmond	23255	37.5313	-77.4160	2917	County
9471	Richmond	23280	37.6376	-77.4680	2917	County
9476	Richmond	23288	37.5313	-77.4160	2917	County
9477	Richmond	23289	37.5313	-77.4160	2917	County
9482	Richmond	23294	37.6329	-77.5450	2917	County
			676.3137	-1394.0520		

Key= 1921

IGrouping<Int32,Places> (9 items)

Place_id	Place_name	Postal_code	Latitude	Longitude	County_id	County
12058	Banner Elk	28604	36.1705	-81.8410	1921	County
12070	Crossnore	28616	36.0591	-81.8890	1921	County
12075	Elk Park	28622	36.1646	-81.9630	1921	County
12097	Linville	28646	36.0410	-81.9170	1921	County
12103	Minneapolis	28652	36.0865	-81.9340	1921	County
12104	Montezuma	28653	36.0651	-81.9010	1921	County
12108	Newland	28657	36.0590	-81.9300	1921	County
12113	Pineola	28662	36.0856	-81.8700	1921	County
12115	Plumtree	28664	36.0441	-82.0030	1921	County
			324.7755	-737.2480		

Figure 41: Result for the GroupBy method

Note that in Figure 41, the value for the **`Key`** property is highlighted. Since the **`County_id`** column is of type **`integer`**, the property also takes this type. So, the result returned by the query is an **`IGrouping<Int32, Places>`** collection, as displayed in the same figure.

Querying for specific columns

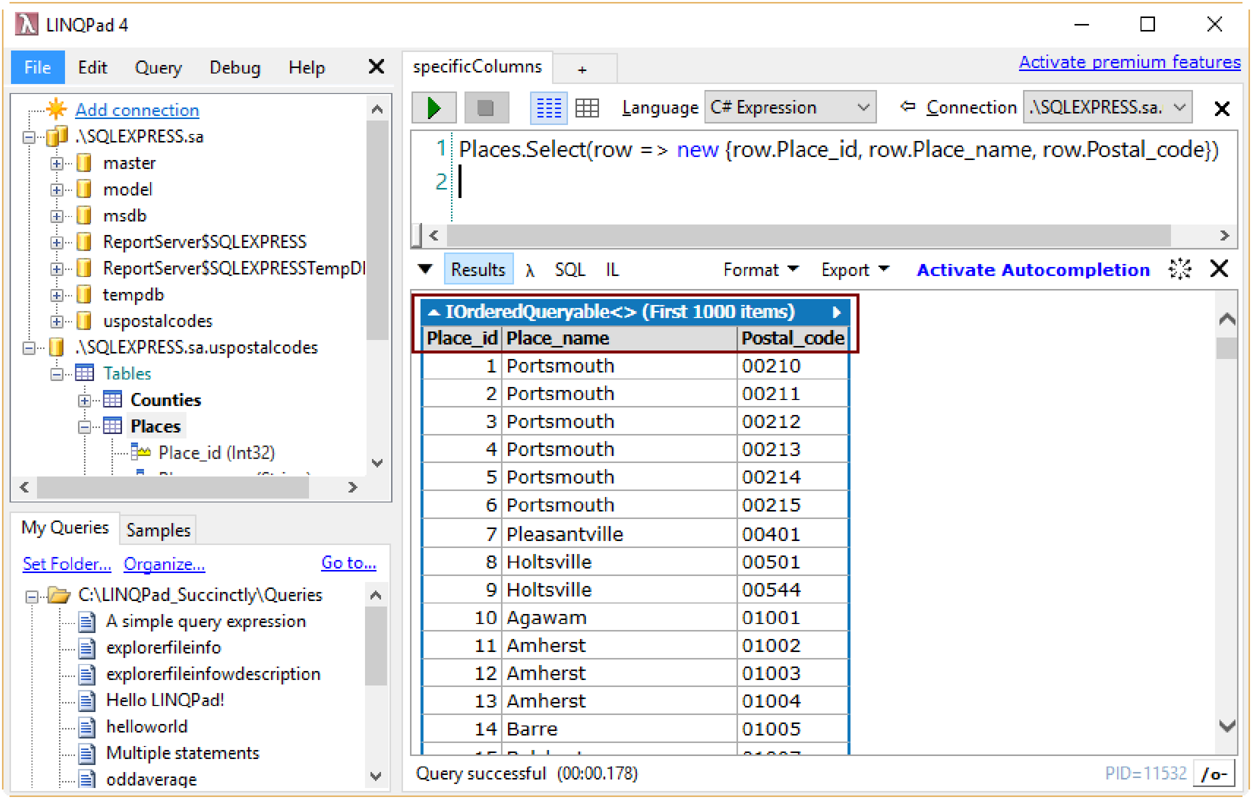
All queries discussed until now display all columns belonging to the table inquired. But LINQ-to-SQL allows us to choose a specific set of columns to be returned by a query. This task can be accomplished by using the **`Select`** method. The following sample returns the **`Place_id`**, **`Place_name`**, and **`Postal_code`** from the **`Places`** table.

Code Listing 22

```
Places.Select(row => new {row.Place_id, row.Place_name, row.Postal_code})
```

The **`Select`** method uses a lambda expression to query a table as a sequence, which is the **`Places`** table in this case. The **`row`** parameter of the lambda expression takes one row from the table at a time. The statement in the lambda expression is executed for each row passed to the parameter, and creates a new item for an **`IOrderedQueryTable<>`** collection. This item is populated with the contents of the **`Place_id`**, **`Place_name`**, and **`Postal_code`** columns.

A partial result of the previous query is shown in Figure 42.



```
Places.Select(row => new {row.Place_id, row.Place_name, row.Postal_code})
```

IOrderedQueryable<> (First 1000 items)

Place_id	Place_name	Postal_code
1	Portsmouth	00210
2	Portsmouth	00211
3	Portsmouth	00212
4	Portsmouth	00213
5	Portsmouth	00214
6	Portsmouth	00215
7	Pleasantville	00401
8	Holtsville	00501
9	Holtsville	00544
10	Agawam	01001
11	Amherst	01002
12	Amherst	01003
13	Amherst	01004
14	Barre	01005

Query successful (00:00.178)

Figure 42: A query with specific columns

Putting all together

All methods explained in the previous sections can be used together in order to retrieve more complex results.

The following code returns the columns **Place_id**, **Place_name**, and **Postal_code** for all places that belongs to the county with an **id** of **1926**.

Code Listing 23

```
Places.Where(row => row.County_id == 1926).Select(row => new {row.Place_id,
row.Place_name, row.Postal_code})
```

The previous example uses both the **Where** and **Select** methods to compose the result. In this case, the **Where** method will ensure that the **Select** method only inquires those rows with a value of **1926** for the **County_id** column. The **Select** method, as explained in the previous section, will read the table as a sequence and make an **IOrderedQueryTable<>** collection. All items for this collection will be populated with the values of **Place_id**, **Place_name**, and **Postal_code** columns.

The results displayed by the previous query are not sorted in any way, so the **OrdeBy** method can be used to display the results sorted by any of the columns given.

The following code will sort the result based on the value for the **Place_name** column.

Code Listing 24

```
Places.Where(row => row.County_id == 1926).OrderBy(row =>
row.Place_name).Select(row => new {row.Place_id, row.Place_name, row.Postal_code})
```

The previous sample uses the **OrderBy** method to sort the rows returned by the **Where** method. The **Select** method reads the rows after being sorted, creating an **IOrderedQueryTable<>** collection with the values of the **Place_id**, **Place_name**, and **Postal_code** columns.

LINQ-to-SQL with multiple statements

LINQPad also allows you to perform LINQ-to-SQL queries using multiple statements. To execute these kinds of queries, you should select **C# Statements** from the **Language selector** combo box.

Code Listing 25

```
var places = Places.GroupBy(row => row.County_id);
var placesSorted = from place in places orderby place.Key
                                   select place;
placesSorted.Dump();
```

The previous code returns a collection of rows grouped by using the value contained in the **County_id** column. This collection is stored in the **places** variable. The **from** statement used in the second line of code takes each group from places variable, in a one-by-one sequence. Each group is stored in the **place** variable, and the **select** statement adds the group to a collection. The collection is stored in the **placesSorted** variable. The **orderby** statement sorts the collection using the **Key** property value of each **place** group.

The PredicateBuilder class

LINQPad includes a special class called **PredicateBuilder**. This class is a powerful LINQ expression that simplifies queries programming when too many filter parameters are used. The class solves this issue by writing dynamic query statements based on the result of lambda expressions supplied to a set of special methods that are intended to act as logical comparison operators. The **PredicateBuilder** is a generic class, which means it can receive any entity reference of the database used by a connection. The use of the **PredicateBuilder** class results in much better performance and minimizes the amount of code to be written for filters implementation.

Using **PredicateBuilder** class for writing queries requires you to reference the class in LINQPad. To do this, press **F4** and check the **Include PredicateBuilder** check box, as shown in the following figure.

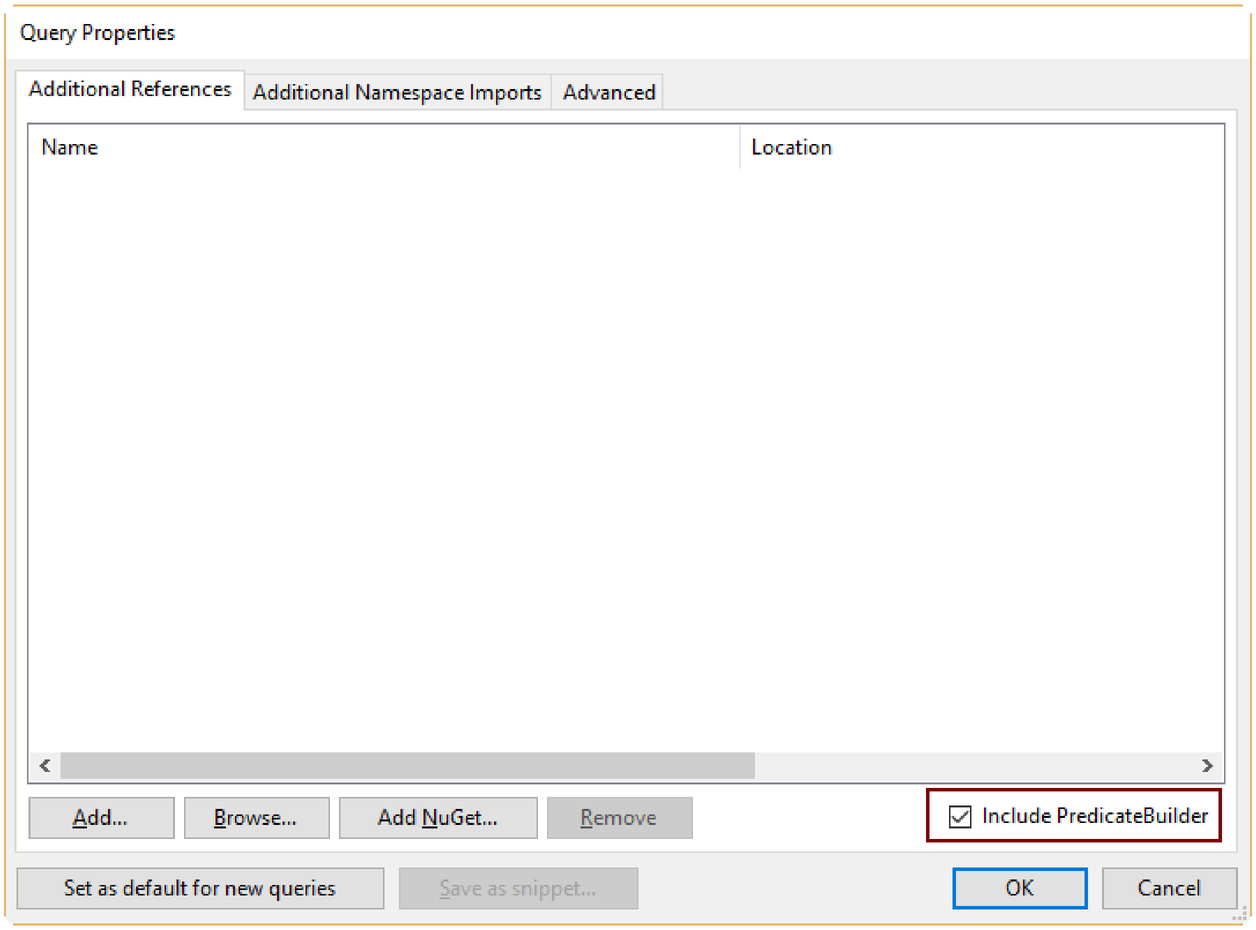


Figure 43: Query Properties dialog box, including PredicateBuilder class

Next, click **OK.** LINQPad will now be able to use the class in the current query.

Note: The Include PredicateBuilder check box must be checked every time a new query is written.

A query that uses PredicateBuilder

The following sample assumes that the user wants to get all counties with a value between **1** and **19** in the **`State_id`** column.

Code Listing 26

```
var resultSet = Counties.Where(row => (row.State_id > 0 && row.State_id <
20));
resultSet.Dump();
```

The previous code looks like any of the previous samples discussed in this book, and it works fine for getting the results desired. But if the user wants to apply a more complex filter condition, the lambda expression within the **`Where`** method will increase in code. At this point, the query will become more difficult to maintain.

The **`PredicateBuilder`** class solves the problem stated in the previous paragraph by being in charge of dynamically building the lambda expression for the **`Where`** clause. So, the user will not need to hard-code the expression within the **`Where`** clause. Instead, an instance of the **`PredicateBuilder`** class is passed to the **`Where`** method, and this instance it creates the lambda expression needed for the method to work. Now the code for the previous query will look like the following sample.

Code Listing 27

```
var predicate = PredicateBuilder.False<Counties>();
predicate = predicate.Or(row => row.State_id > 0);
predicate = predicate.And(row => row.State_id < 20);
var resultSet = Counties.Where(predicate);
resultSet.Dump();
```

As displayed in the previous code, the first line creates an instance of the **`PredicateBuilder`** class, referencing the **`Counties`** table. Then, the query calls the **`Or`** method of the **`PredicateBuilder`** class and passes to it a lambda expression. This expression returns every row with a value greater than **`0`** for the **`State_id`** column. The next line uses the **`And`** method to establish the other part of the filter condition. In this case, the expression will return every row with a value less than **`20`** for the **`State_id`** column. Now the instance of the **`PredicateBuilder`** class is passed to the **`Where`** method instead of the lambda expression, because this instance will be in charge of building that expression when the method is executed.

Note: For proper building of a lambda expression, the *Or* ***method must be called first.***

Maybe for a simple expression, like the one discussed in the samples of this section, using the **`PredicateBuilder`** class turns the code in more lines. But as the filter condition becomes more complex, the code turns simpler. The following code adds another filter condition to the query.

Code Listing 28

```
 var predicate = PredicateBuilder.False<Counties>();
predicate = predicate.Or(row => row.State_id > 0);
predicate = predicate.And(row => row.State_id < 20);
predicate = predicate.Or(row => row.County_name.Contains("Orange"));
var resultSet = Counties.Where(predicate);
resultSet.Dump();
```

As noted in the previous sample, the result for the query will return all rows that contain the phrase **`Orange`** in the **`County_name`** column, other than all rows in which the **`State_id`** column has a value greater than **`0`** and less than **`20`**. The lambda expression that the predicate instance will create is the following.

Code Listing 29

```
row => (row.State_id > 0 && row.State_id < 20) ||
row.County_name.Contains("Orange")
```

As noted in the previous sample, the expression is a little more complex, and if the user wants to retrieve rows that match more filter criteria, this expression would be larger. So, repeatedly calling the **And** or the **Or** method is easier than creating complex expressions.

A multiple keyword search

Sometimes you want to search rows in a database based on one or more keywords. Using a lambda expression for this purpose could make the query's code complex and hard to maintain. So, instead of hard-coding the lambda expression, a **PredicateBuilder** instance can be employed to perform the search dynamically, as in the following sample.

Code Listing 30

```
var keywords = new List<string> {"Orange","York","Fernando"};
var predicate = PredicateBuilder.False<Counties>();

foreach (string keyword in keywords)
{
        predicate = predicate.Or(row => row.County_name.Contains(keyword));
}
var resultSet = Counties.Where(predicate);
resultSet.Dump();
```

The previous code performs a search in the **Counties** table based on the keywords stored in the **keywords** variable, which is a list of strings. A **PredicateBuilder** instance is created and the **Or** method is called repeatedly using a **foreach** loop. For each item in the list, the **Or** method receives a lambda expression, which returns a **true** or a **false** value, whether the keyword is contained in the **County_name** column or not.

The predicate is used by the **Where** method associated to the **Counties** table, and returns all rows that contains any of the keywords in the list as a result.

Chapter summary

This chapter discussed how to create database queries with LINQPad using the default LINQ-to-SQL driver. All samples explained in this chapter use a database named **uspostalcodes**. To create this database, the **Add connection** hyperlink is used to call the **Choose Data Context** dialog box to select the LINQ-to-SQL driver. Then, the LINQ to SQL Connection dialog allows you to create the **uspostalcodes** database.

There is no way to create database entities with LINQ-to-SQL, so you'll use Microsoft SQL Server Management Studio to create the database tables. Once they're created, you'll use a series of scripts to populate tables with data.

LINQ-to-SQL has a series of methods that retrieve rows from a given table. For example, the **`Take`** method returns a row set as large as the value of the integer parameter passed to the method as a parameter, and the **`OrderBy`** method allows you to sort the rows returned by a query.

Some methods require a lambda expression in order to work properly. The term *lambda expression* comes from *lambda* calculus, which is a mathematical notation applied for defining functions. A lambda expression is composed by a parameter, and an expression that takes the parameter and uses it to return a value.

The methods that use lambda expressions are: **`OrderBy`** (to sort rows), **`Where`** (to filter rows based on a conditional expression), **`Sum`** (to summarize the value of a given column), **`Average`** (to get an average from a series of values of a given column), and **`GroupBy`** (to group rows based on the value of a given column).

LINQ-to-SQL queries can be written as a C# expression or as several C# statements, depending on the user's needs.

LINQPad provides a special class named **`PredicateBuilder`**. This class is intended to simplify query-writing when a complex filter condition is needed. The class solves this issue providing a set of special methods intended to act as logical comparison operators. Because **`PredicateBuilder`** is a generic class, it can receive any entity reference of the database used by a connection.

The value of the **`PredicateBuilder`** class comes from its ability to build any lambda expression dynamically, when the instance of the class is passed to any LINQ-to-SQL method. The expression is created using the values passed to the class instance using either the **`And`** or **`Or`** methods, which should be called repeatedly before using the instance.

Writing queries that use the **`PredicateBuilder`** class requires you to reference this class in LINQPad. This can be done by pressing **F4** and checking the **Include PredicateBuilder** check box that appears in the **Query Properties** dialog box. You need to perform this action every time you write a new query.

Chapter 6 LINQPad as a Code Scratchpad

One of the best uses of LINQPad is as a test and prototyping tool. As mentioned in the "Languages available in LINQPad" section of Chapter 4, LINQPad allows you to run code in the C#, F#, and VB languages. So no matter which of those languages you employ for development, LINQPad will be useful.

This book relies entirely on the C# programming language, so all code testing or prototyping will be written in C#.

How C# code can be executed in LINQPad

C# code can be written in LINQPad in three different ways:

- As an expression: A single-line C# expression with no semicolon is executed.
- As statements: Several C# lines of code are executed. Each line must end with a semicolon.
- As a program: LINQPad automatically adds a **`Main`** method to the query in order to execute statements or another methods or classes within the method. The methods or classes to execute can be defined after the closing brace of the **`Main`** method is created.

Testing a C# expression

Sometimes when a program is being coded, there's a need to test small pieces of code to ensure they will provide the expected result. One of these pieces of code is an expression.

The following example assumes that you need to test an expression that converts the actual **`DateTime`** value to a string, according to the XML Schema format.

Code Listing 31

```
XmlConvert.ToString(DateTime.Now,XmlDateTimeSerializationMode.Unspecified)
```

The static **`ToString`** method of the **`XmlConvert`** class is used to make the **`DateTime`** value conversion. This value is passed to the method as the first parameter. The second parameter passed to the method is the **`XmlDateTimeSerializationMode.Unspecified`** enumeration value. This value tells the **`ToString`** method that the time zone for the actual date and time should be ignored. The result is displayed in the following snippet.

Code Listing 32

```
2016-06-12T18:55:08.0376168
```

Executing C# statements

In the output generated in the sample explained in previous section, note that the **`seconds`** part of the converted **`DateTime`** value is returned with fractional parts of the **`seconds`** value. If the desired result is expected without those fractions, an additional statement should be executed to remove them from the string. Thus, a C# Statements query should be written to solve this topic.

Code Listing 33

```
var xmlschemaDatetime =
XmlConvert.ToString(DateTime.Now,XmlDateTimeSerializationMode.Unspecified);
xmlschemaDatetime = xmlschemaDatetime.Substring(0,xmlschemaDatetime.IndexOf("."));
xmlschemaDatetime.Dump();
```

The previous code sample corresponds to a C# Statements query. Unlike with a C# Expression query, writing C# Statements queries requires you to declare variables in order to assign the results coming from methods or calculations. The **`Dump`** method must be included to display the final results that come up from the queries.

The query displayed in the previous example uses the **`Substring`** and the **`IndexOf`** methods of the string class in order to remove the fractions from the **`seconds`** part of the converted **`DateTime`** value.

The expected result must contain no decimal fractions, so the **`Substring`** method needs to take only those characters before the decimal point. The **`IndexOf`** method is used to calculate the position of the decimal point. Because strings are zero-based indexed, if the decimal point appears in the 19th position, it really is the 20th character (20 – 1 = 19) of the string. Thus, if this is the case, the **`Substring`** method must take 19 characters, and that is exactly the number returned by the **`IndexOf`** method.

The result is displayed the following snippet.

Code Listing 34

```
2016-06-12T18:55:08
```

Running a C# program

The third way C# code can be executed in a LINQPad query is similar to how we run a C# Program. When this kind of query is selected from the Language Selector combo box, LINQPad automatically creates a **`Main`** method, which will serve as an entry point for the query.

A C# Program query may work for testing or prototyping pieces of code that might be part of a bigger project. Once these pieces of code pass all tests, they can be part of a Visual Studio solution. This practice can save you time, because you don't need to build another project for testing purposes. Even so, parts of the solution to be delivered could exist before the solution itself is created in Visual Studio.

The following code is a C# Program query version of the C# Statements query explained in the previous section.

Code Listing 35

```
void Main()
{
     var convertedDateTime = toXmlSchemaDateTime(DateTime.Now);
     convertedDateTime.Dump();
}

// Define other methods and classes here
public string toXmlSchemaDateTime(DateTime currentDateTime)
{
    var xmlschemaDatetime =
XmlConvert.ToString(DateTime.Now,XmlDateTimeSerializationMode.Unspecified);
    xmlschemaDatetime =
xmlschemaDatetime.Substring(0,xmlschemaDatetime.IndexOf("."));

       return (xmlschemaDatetime);
}
```

As noted in the previous example, the conversion process is now delegated to a program-defined method named **`toXmlSchemaDateTime`**. The method definition is written under the **`Main`** method's definition, as suggested by LINQPad by way of the comment added to the code. This approach allows you to convert any **`DateTime`** value to its **`XmlSchema`** equivalent, not only the current system's **`DateTime`**.

Now, the **`Main`** method creates a variable and calls the **`toXmlSchemaDateTime`** method with the value of the current system's **`DateTime`**. The **`Dump`** method displays the result.

An example for a test workbench usage

As mentioned in the previous section, the ability to test parts of code before being part of a solution makes LINQPad robust and powerful. In this section, it's going to be assumed that a class for managing **`DateTime`** values needs to be tested before the developer puts the code into a Visual Studio solution. A query for this purpose should look like the following sample.

Code Listing 36

```
void Main()
{
     var dateTimeManager = new DateTimeManager();
        dateTimeManager.toXmlSchemaDateTime(DateTime.Now).Dump("toXmlSchemaDateTime
method");
     dateTimeManager.backNDays(DateTime.Now,3).Dump("backNDays method");
        dateTimeManager.forwardNDays(DateTime.Now,15).Dump("forwardNDays method");
}

// Define other methods and classes here
public class DateTimeManager
{
      public string toXmlSchemaDateTime(DateTime currentDateTime)
       {
    var xmlschemaDatetime =
XmlConvert.ToString(DateTime.Now,XmlDateTimeSerializationMode.Unspecified);
```

```
           xmlschemaDatetime = 
xmlschemaDatetime.Substring(0,xmlschemaDatetime.IndexOf("."));

             return (xmlschemaDatetime);
       }

         public DateTime backNDays(DateTime currentDateTime, int daysToGoBack)
         {
                return 
(currentDateTime.Subtract(TimeSpan.FromDays(daysToGoBack)));
         }

         public DateTime forwardNDays(DateTime currentDateTime, int daysToGoBack)
         {
                return (currentDateTime.Add(TimeSpan.FromDays(daysToGoBack)));
         }
}
```

The previous code shows a class named **DateTimeManager**. The class has three methods:

- **toXmlSchemaDateTime**: Converts a **DateTime** value to a string complying with the format defined by the XmlSchema.
- **backNDays**: Returns a **DateTime** value, which is a given number of days before the **DateTime** value passed to the method.
- **forwardNDays**: Returns a **DateTime** value, which is a given number of days after the **DateTime** value passed to the method.

The previous code shows how all methods for the class can be tested at the same time. This is done by creating an instance of the class and calling each of the methods, along with **Dump**. A string is passed to each **Dump** call in order to identify the result for each one of the methods tested.

The output for the query is displayed in the following sample.

Code Listing 37

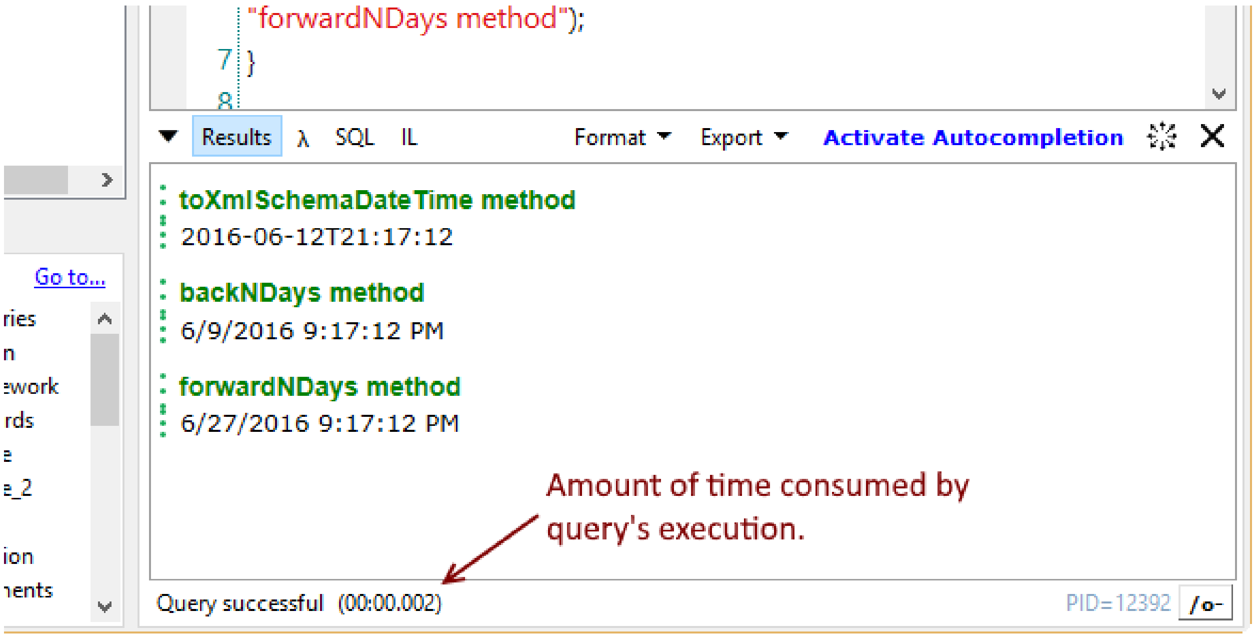
```
toXmlSchemaDateTime method
2016-08-03T19:05:05

backNDays method
7/31/2016 7:05:05 PM

forwardNDays method
8/18/2016 7:05:05 PM
```

If the results are as expected, the class code can be added to the solution.

Measuring query performance

When LINQPad is used for testing purposes, another important issue is measuring code performance. A common way to measure performance is by adding time-managing statements, and performing subtract operations.

LINQPad gives the user a very simple solution for this issue: every time a query execution is completed, the amount of time spent to get the results is displayed in the status bar.



```
"forwardNDays method");
}
```

```
toXmlSchemaDateTime method
2016-06-12T21:17:12

backNDays method
6/9/2016 9:17:12 PM

forwardNDays method
6/27/2016 9:17:12 PM
```

Query successful (00:00.002)

Figure 44: Time consumed by query's execution

As noted in Figure 44, the precision for performance measuring falls into milliseconds.

Referencing custom and default assemblies

LINQPad also allows the use of custom assemblies in queries. You can reference these assemblies in a query by selecting the **Query Properties** command from the **Query** menu.

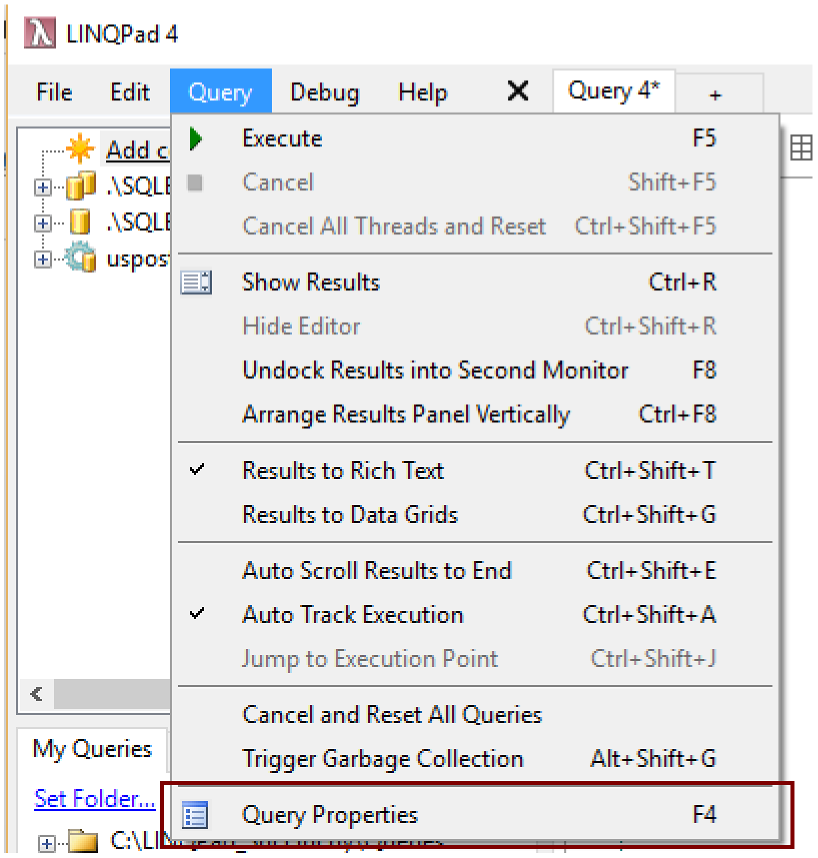

LINQPad 4
File
Edit
Query
Debug
Help
Query 4*
+
Add c
.\SQLE
.\SQLE
uspos
My Queries
Set Folder...
C:\LI
Execute F5
Cancel Shift+F5
Cancel All Threads and Reset Ctrl+Shift+F5
Show Results Ctrl+R
Hide Editor Ctrl+Shift+R
Undock Results into Second Monitor F8
Arrange Results Panel Vertically Ctrl+F8
Results to Rich Text Ctrl+Shift+T
Results to Data Grids Ctrl+Shift+G
Auto Scroll Results to End Ctrl+Shift+E
Auto Track Execution Ctrl+Shift+A
Jump to Execution Point Ctrl+Shift+J
Cancel and Reset All Queries
Trigger Garbage Collection Alt+Shift+G
Query Properties F4

Figure 45: The Query Properties command

Another way to reference the assemblies is by pressing the **F4** key, as the previous figure suggests. Whichever method you use, the Query Properties dialog box will be displayed so that you can add the needed references.

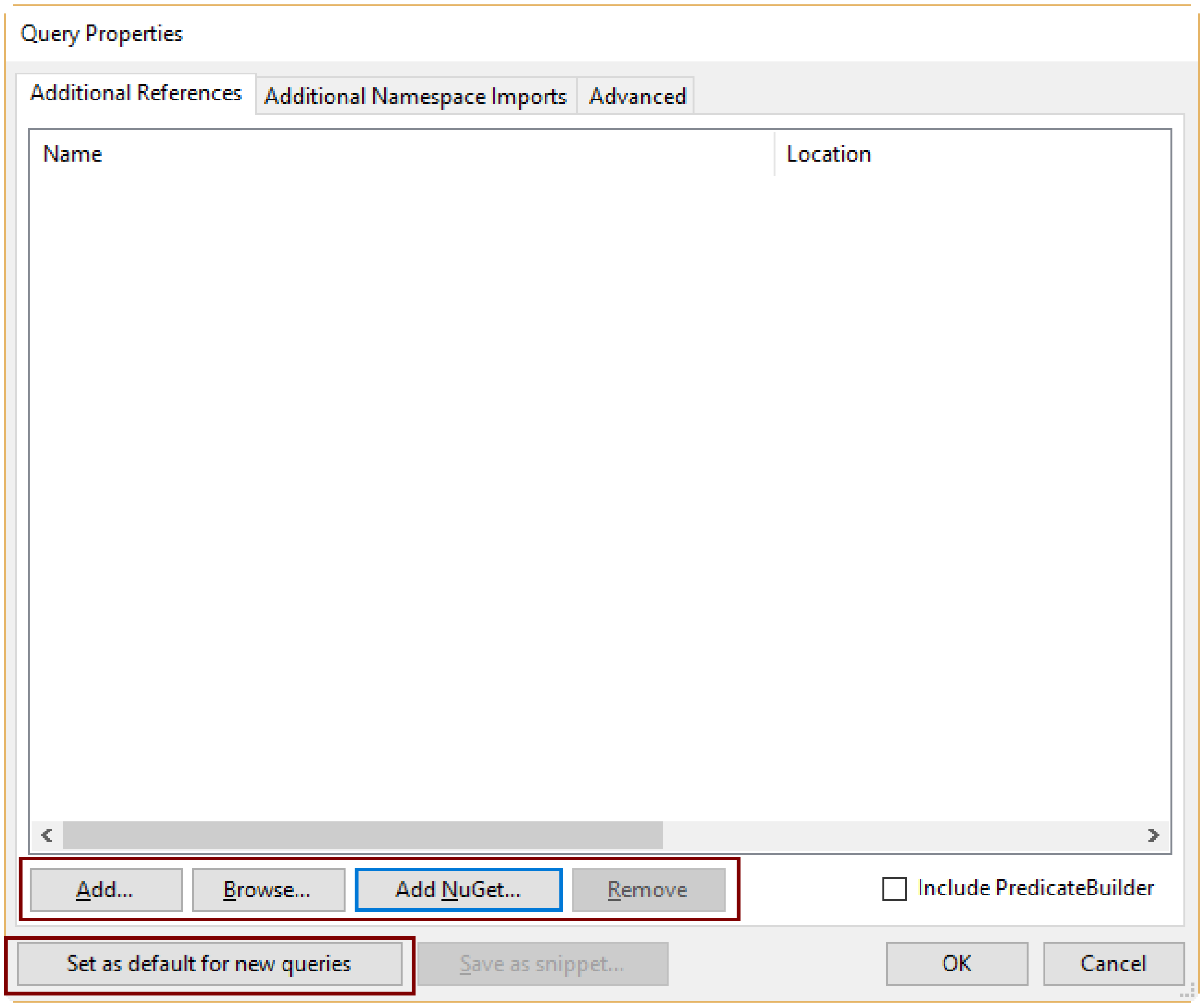
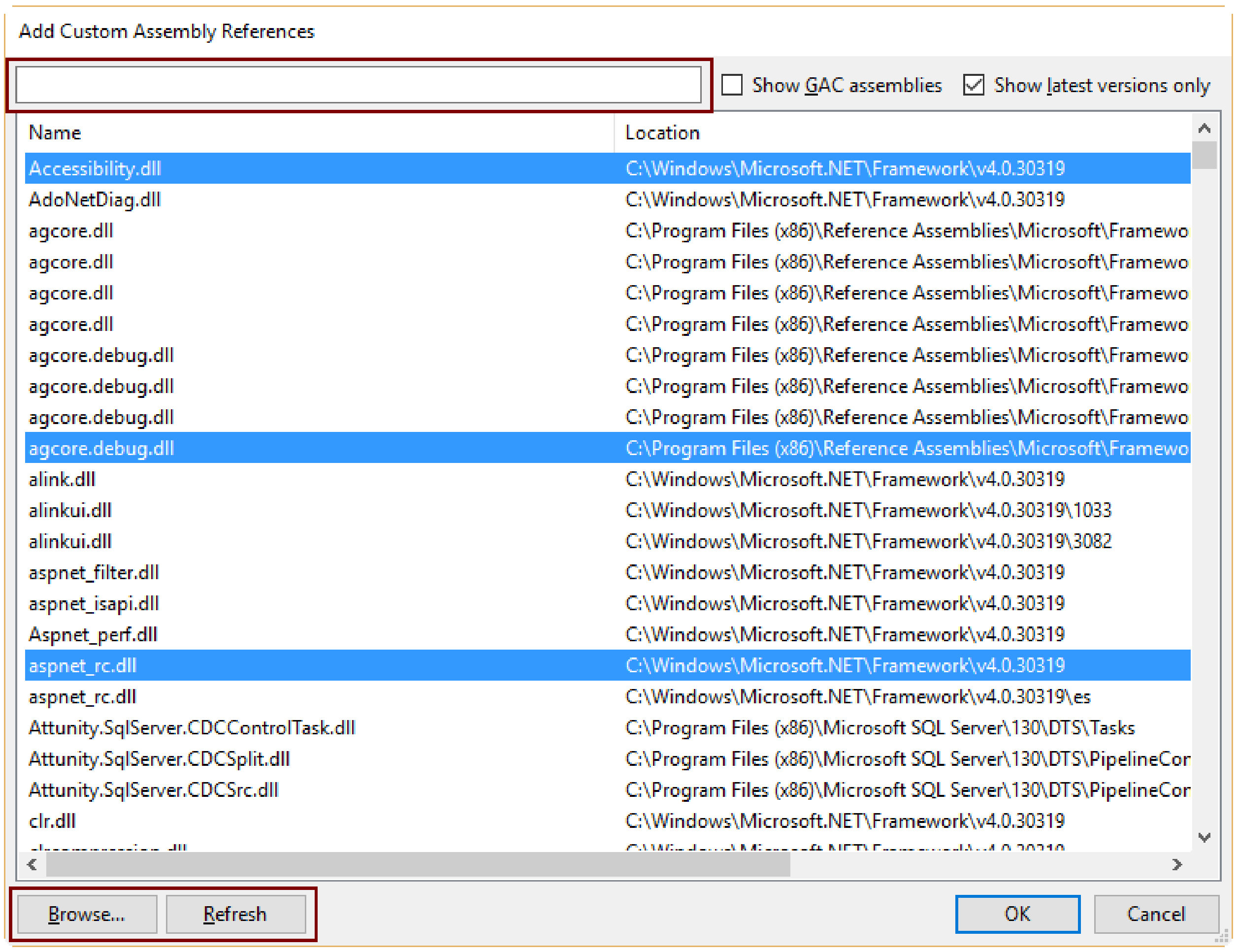


Figure 46: Query Properties dialog box

The dialog box provides a group of buttons that should be used to add an assembly reference to the query. The purpose for each of these buttons will be explained in the following sections.

Picking up an assembly from the system

The Add Custom Assembly References dialog box is employed to choose an assembly from those available in the system. To bring up this dialog, click the **Add** button located in the **Query Properties** dialog box. The Add Custom Assembly References dialog box displays all assemblies found in the system. You can select one or more of these by clicking on the name of the desired assemblies and pressing the **Ctrl** or **Shift** keys as you would in the Windows File Explorer.


Add Custom Assembly References

☐ Show GAC assemblies ☑ Show latest versions only

Name	Location
Accessibility.dll	C:\Windows\Microsoft.NET\Framework\v4.0.30319
AdoNetDiag.dll	C:\Windows\Microsoft.NET\Framework\v4.0.30319
agcore.dll	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framewo
agcore.dll	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framewo
agcore.dll	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framewo
agcore.dll	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framewo
agcore.debug.dll	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framewo
agcore.debug.dll	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framewo
agcore.debug.dll	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framewo
agcore.debug.dll	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framewo
alink.dll	C:\Windows\Microsoft.NET\Framework\v4.0.30319
alinkui.dll	C:\Windows\Microsoft.NET\Framework\v4.0.30319\1033
alinkui.dll	C:\Windows\Microsoft.NET\Framework\v4.0.30319\3082
aspnet_filter.dll	C:\Windows\Microsoft.NET\Framework\v4.0.30319
aspnet_isapi.dll	C:\Windows\Microsoft.NET\Framework\v4.0.30319
Aspnet_perf.dll	C:\Windows\Microsoft.NET\Framework\v4.0.30319
aspnet_rc.dll	C:\Windows\Microsoft.NET\Framework\v4.0.30319
aspnet_rc.dll	C:\Windows\Microsoft.NET\Framework\v4.0.30319\es
Attunity.SqlServer.CDCControlTask.dll	C:\Program Files (x86)\Microsoft SQL Server\130\DTS\Tasks
Attunity.SqlServer.CDCSplit.dll	C:\Program Files (x86)\Microsoft SQL Server\130\DTS\PipelineCor
Attunity.SqlServer.CDCSrc.dll	C:\Program Files (x86)\Microsoft SQL Server\130\DTS\PipelineCor
clr.dll	C:\Windows\Microsoft.NET\Framework\v4.0.30319

Figure 47: Add Custom Assembly References dialog box

As noted in Figure 47, the dialog box also provides a search bar, which allows you to filter the assemblies listing by their full or partial names.

There are two check boxes to the right of the search bar. The first one tells LINQPad if the assemblies installed in the Global Assembly Cache (GAC) will be included in the list (checked) or not (unchecked). The other one restricts the assemblies listing only to their latest versions (checked), or tells LINQPad to include all installed versions for each one of them (unchecked).

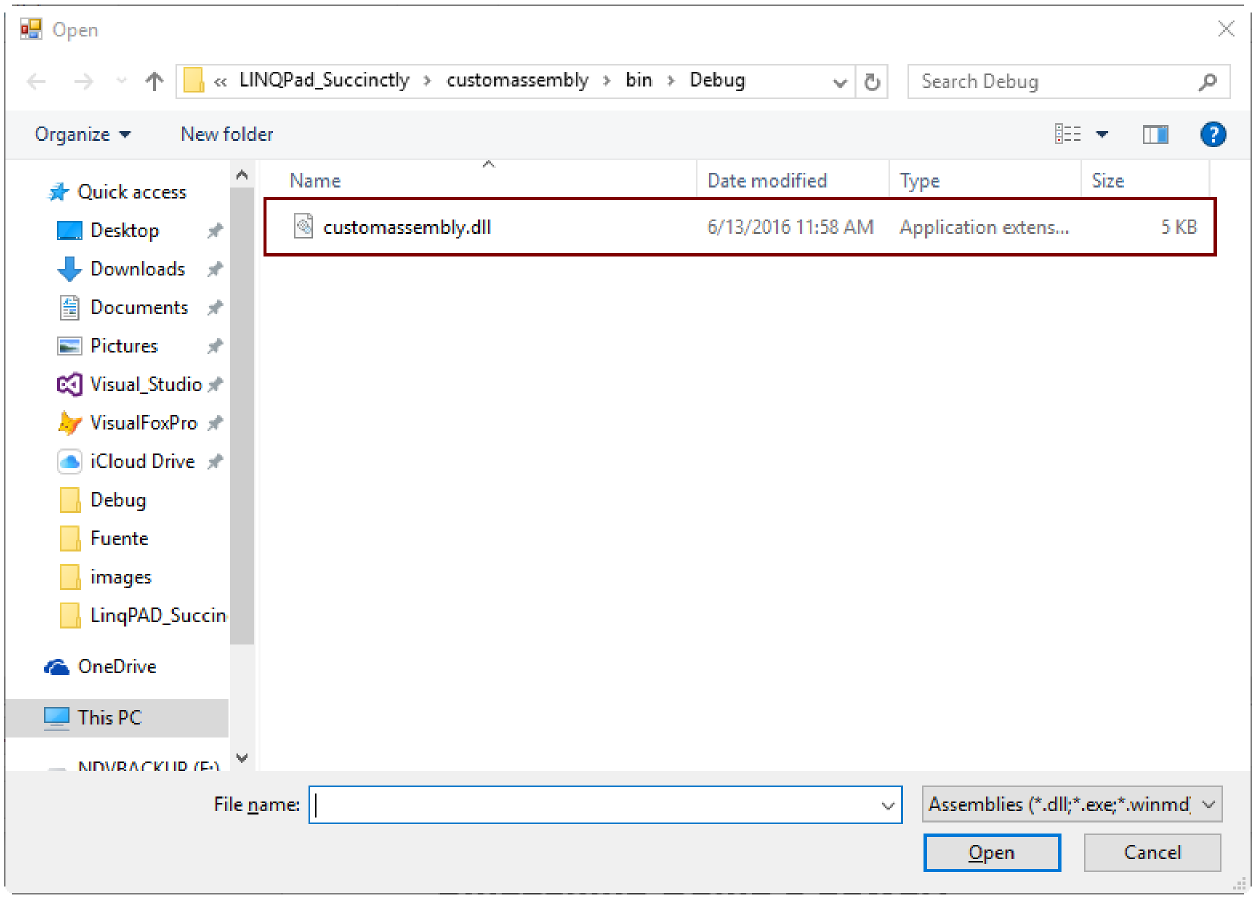
Note: The Global Assembly Cache (GAC) is the location of the computer that will store all assemblies designated to be shared by several applications. The GAC is typically located at C:\Windows\Microsoft.NET\assembly.

The Browse button, located at the bottom of the dialog box, lets allows you to search an assembly within the computer's file system.

The Refresh button, also located at the bottom of the dialog box, lets you refresh the assemblies listing to include recently installed assemblies.

Choosing an assembly directly from the computer's file system

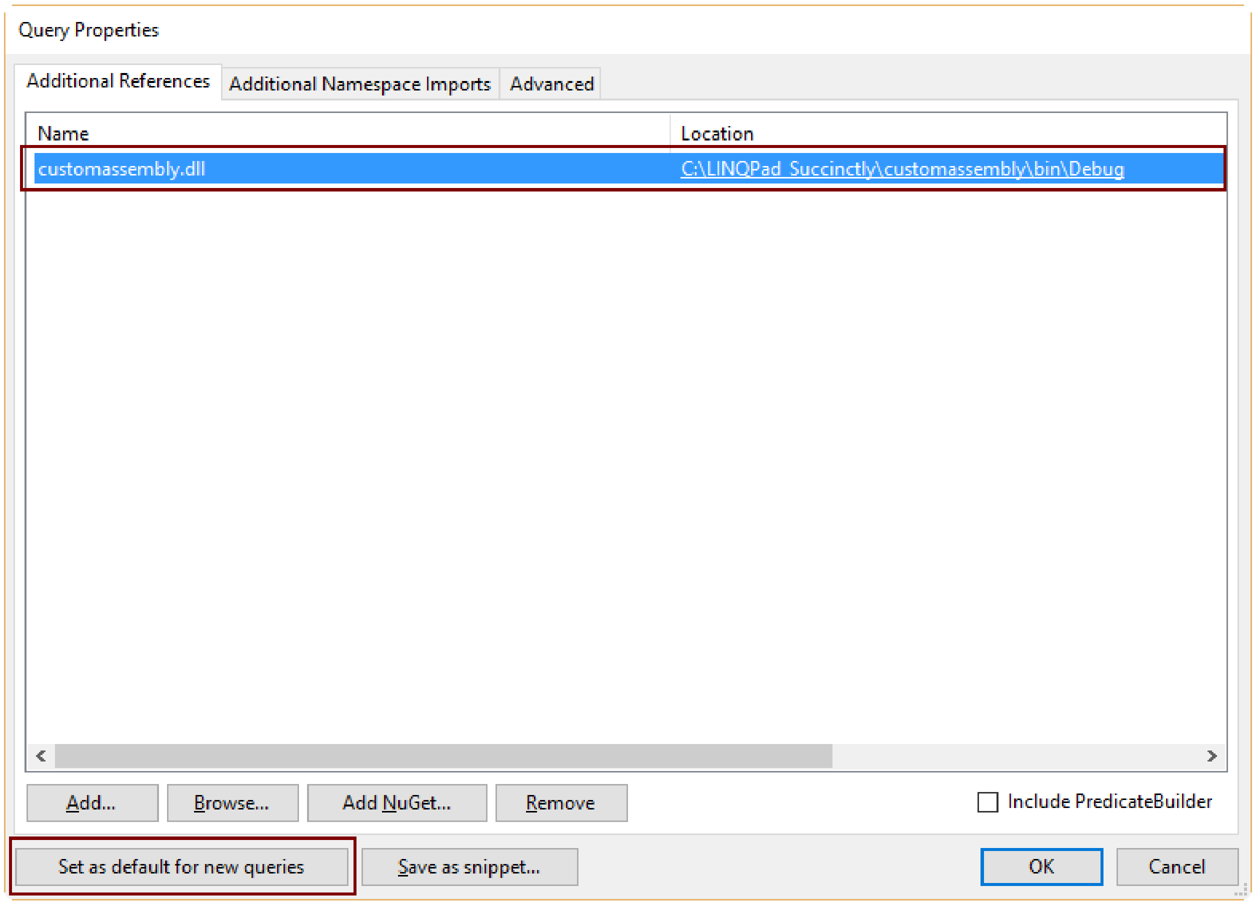
You can choose an assembly from the computer's file system without displaying the Add Custom Assembly References dialog box. You can do this by clicking the **Browse** button in the **Query Properties** dialog box. Now, the use can locate the desired assembly and click **Open** to pick it up. For the purposes of this book, a class library project named **customassembly** was created in the **C:\LINQPad_Succinctly** folder.



Name	Date modified	Type	Size
customassembly.dll	6/13/2016 11:58 AM	Application extens...	5 KB

Figure 48: Picking up the custom assembly library

Once the assembly is opened, its name appears in the Query Properties dialog box. You can accept the selection by clicking **OK**, or you can tell LINQPad that the assembly should be used by the current query and all those created after this one. To do this, click **Set as default for new queries**.



Name	Location
customassembly.dll	C:\LINQPad_Succinctly\customassembly\bin\Debug

Figure 49: The Query Properties dialog box after selecting the assembly

Importing namespaces into queries

When a developer writes a C# program and needs to employ a class or method located in another namespace, this program usually starts with the following statements.

Code Listing 38

```
using System.Drawing;
using System.Net;
using System.Windows.Forms;
```

The action performed by the **`using`** statements is called *importing namespaces* and allows the developer to employ all classes exposed in the namespaces declared by each one. Of course, the assemblies that contain these namespaces must be referenced in the project in order to make each **`using`** statement work properly.

In all previous samples where an assembly reference was added, when a class or method contained in that assembly was needed, it had to be written in a fully qualified form. That is, the name of the namespace that contains the class or method is written before the name of the method itself, followed by a period.

Code Listing 39

```
var image = Util.Image(new
System.Net.WebClient().DownloadData("https://www.linux.com/sites/all/themes/custom/
lcom/logo.png")); //The WebClient method is used in the form of
System.Net.WebClient(), where System.Net is the namespace which contains the method
image.Dump();
```

In the previous sample, the **`WebClient`** method is coded using its fully qualified name. In this case, the namespace that contains the method is **`System.Net`**. The namespace declaration cannot be omitted when the method is invoked. If this happens, LINQPad will trigger an error message upon the query's execution because it won't know where exactly the **`WebClient`** method is located. This will occur even when the reference to the assembly containing the namespace is added. This issue can be solved by importing the namespace into the query. However, the **`using`** statement cannot be employed in LINQPad queries. Instead, the Query Properties dialog box and the Additional Namespace imports tab should be used.

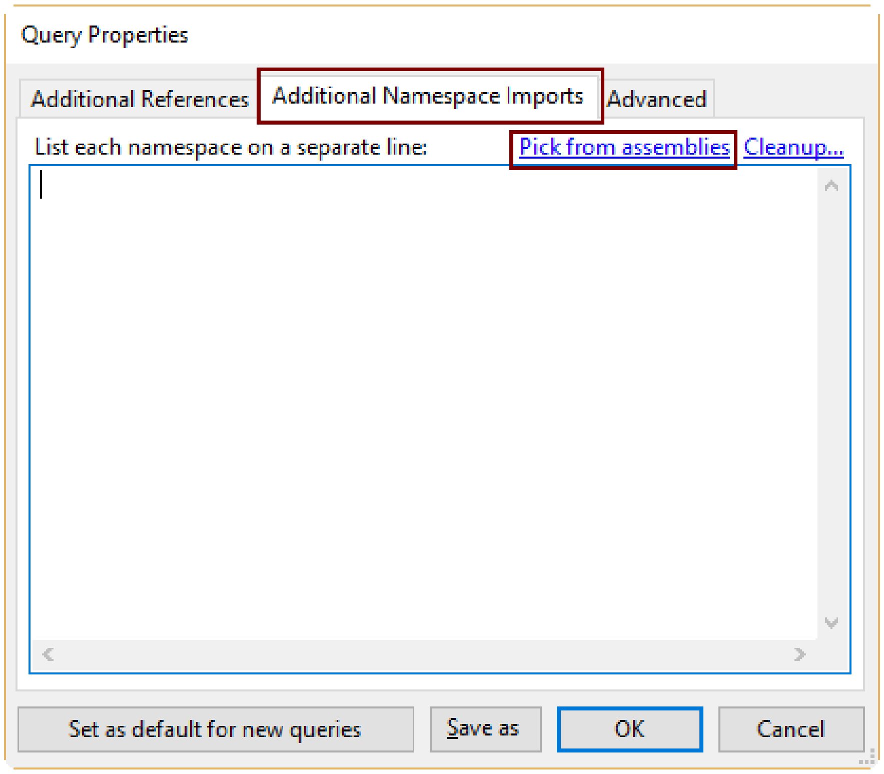

Query Properties
Additional References
Additional Namespace Imports
Advanced
List each namespace on a separate line:
Pick from assemblies
Cleanup...
Set as default for new queries
Save as
OK
Cancel

Figure 50: The Additional Namespace Imports tab

Figure 50 shows the Additional Namespace Imports tab within the Query Properties dialog box. A multiline text box is available in order to write the name for each namespace to be imported on a separate line. A hyperlink named **Pick from assemblies** is also available. When this hyperlink is clicked, a list with all referenced assemblies in the query appears on the right.

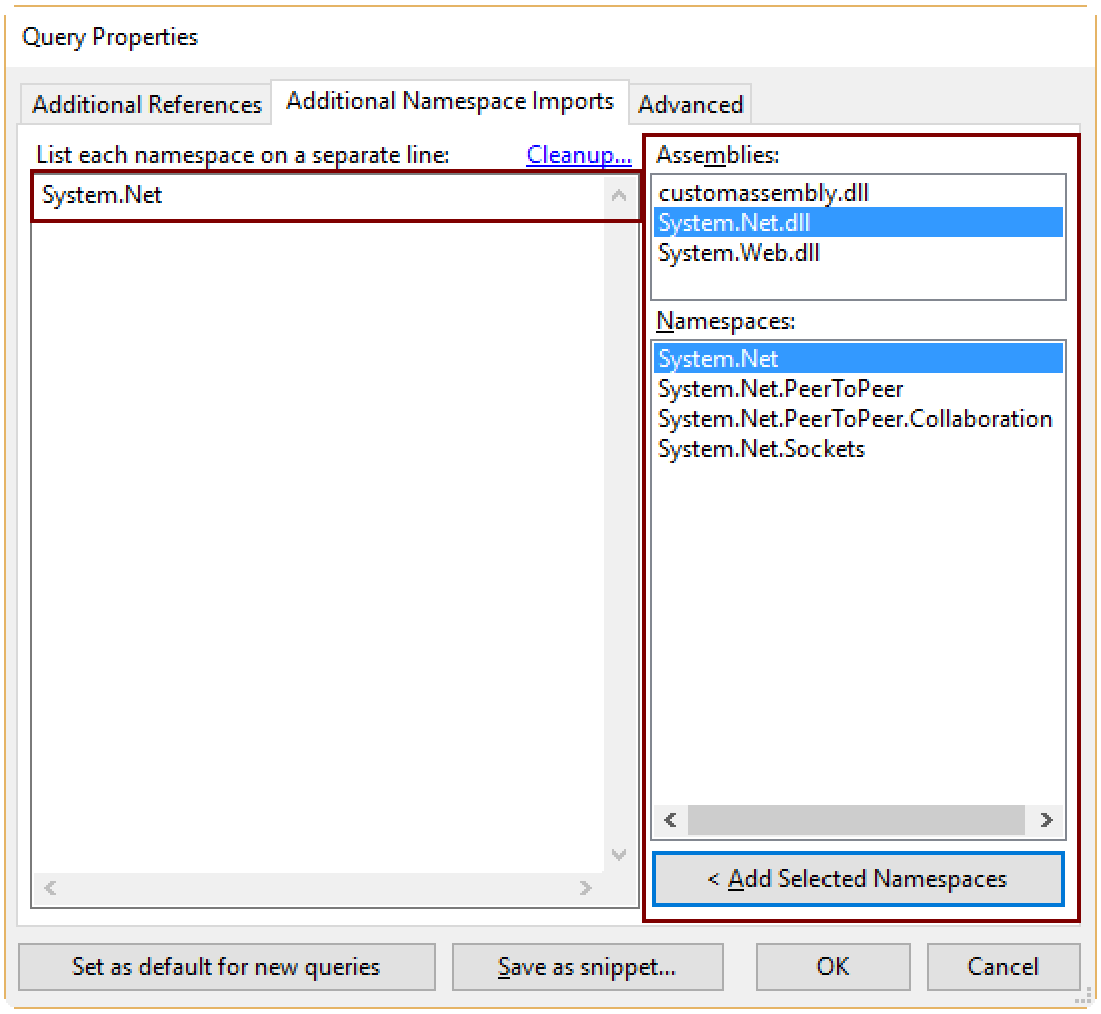


*Figure 51: The Query Properties dialog box after **Pick from assemblies** is clicked*

As displayed in Figure 51, the Namespaces list, which appears under the Assemblies list, shows all available namespaces within the assembly selected. This is a multi-select list, so several namespaces can be chosen at the same time. This multi selection can be done by using the Ctrl + click or the Shift + click combinations, as in the Windows File Explorer.

Clicking **OK** will import the selected namespaces into the query. If you want this selection to be available for all new queries, click **Set as default for new queries** before importing the namespaces.

Now, the code for the query should look like the following sample.

Code Listing 40

```
var image = Util.Image(new
WebClient().DownloadData("https://www.linux.com/sites/all/themes/custom/lcom/logo.p
ng"));
image.Dump();
```

As noted in the previous sample, the namespace declaration was omitted. There won't be an error message when the query is executed because the namespace for the **`WebClient`** method was previously imported into the query.

Unleashing Dump's power

As explained in Chapter 4, the **`Dump`** extension method is capable of rendering almost any type of object to generate output that will be displayed on the screen. This section will explore some powerful ways to use the **`Dump`** method.

Testing the appearance of a Windows form control

Sometimes, when a developer is building a custom controls assembly, it is necessary to see how those controls will look before releasing the project. The developer might write another project for testing purposes, which consumes time and effort.

With LINQPad, this time and effort can be downsized significantly, because LINQPad allows us to write a series of statements along with the **`Dump`** method, in order to prototype a control. In this case, the **`Dump`** method will render the control and displays it in the Results panel. The developer can run the statements over and over, until the desired results are presented. Then, the code can be added to the project.

The following sample prototypes a **`CustomButton`** class, and displays that custom button in the Results panel.

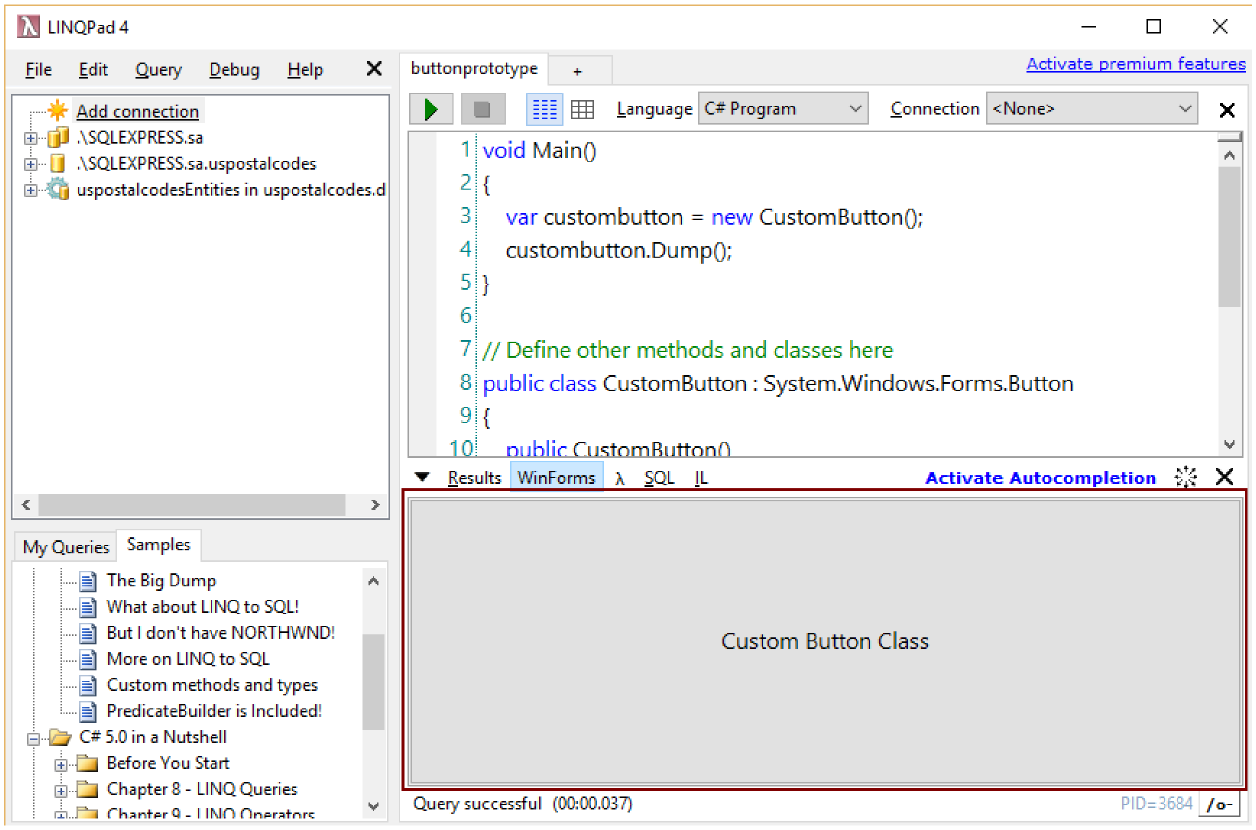
Code Listing 41

```
void Main()
{
	var custombutton = new CustomButton();
	custombutton.Dump();
}

// Define other methods and classes here
public class CustomButton : System.Windows.Forms.Button
{
    public CustomButton()
	{
	   Text = "Custom Button Class";
	   Font = new System.Drawing.Font("Segoe UI",12);
	}
}
```

This is a C# Program query, which defines a **`CustomButton`** class under its **`Main`** method. An instance for this custom class is created in the **`Main`** method, and the variable that stores the instance uses the **`Dump`** method to display the button.

Note: The previous example needs the System.Windows.Forms and System.Drawing assemblies to be referenced, as explained in the "Referencing custom and default assemblies" section of this chapter.

The output produced by the previous query is displayed in Figure 52.



```
void Main()
{
  var custombutton = new CustomButton();
  custombutton.Dump();
}

// Define other methods and classes here
public class CustomButton : System.Windows.Forms.Button
{
  public CustomButton()
```

Figure 52: The custom button displayed in the Results panel

As exposed in Figure 52, the **`Dump`** method renders the button, which is displayed in the Results panel.

The CustomButton class with a different approach

The following code prototypes the **`CustomButton`** class with a few changes.

Code Listing 42

```
void Main()
{
        var custombutton = new CustomButton();
        custombutton.Dump();
}
```

```
// Define other methods and classes here
public class CustomButton : System.Windows.Forms.Button
{
    public CustomButton()
      {
           FlatStyle = System.Windows.Forms.FlatStyle.Flat;
              BackColor = System.Drawing.Color.FromArgb(244,152,31);
              ForeColor = System.Drawing.Color.Black;
         Font = new System.Drawing.Font("Segoe UI",12);
         Text = "Custom Button Class Flat Version";
      }
}
```

In this case, the class will create a flat button with an orange background and a black colored text.

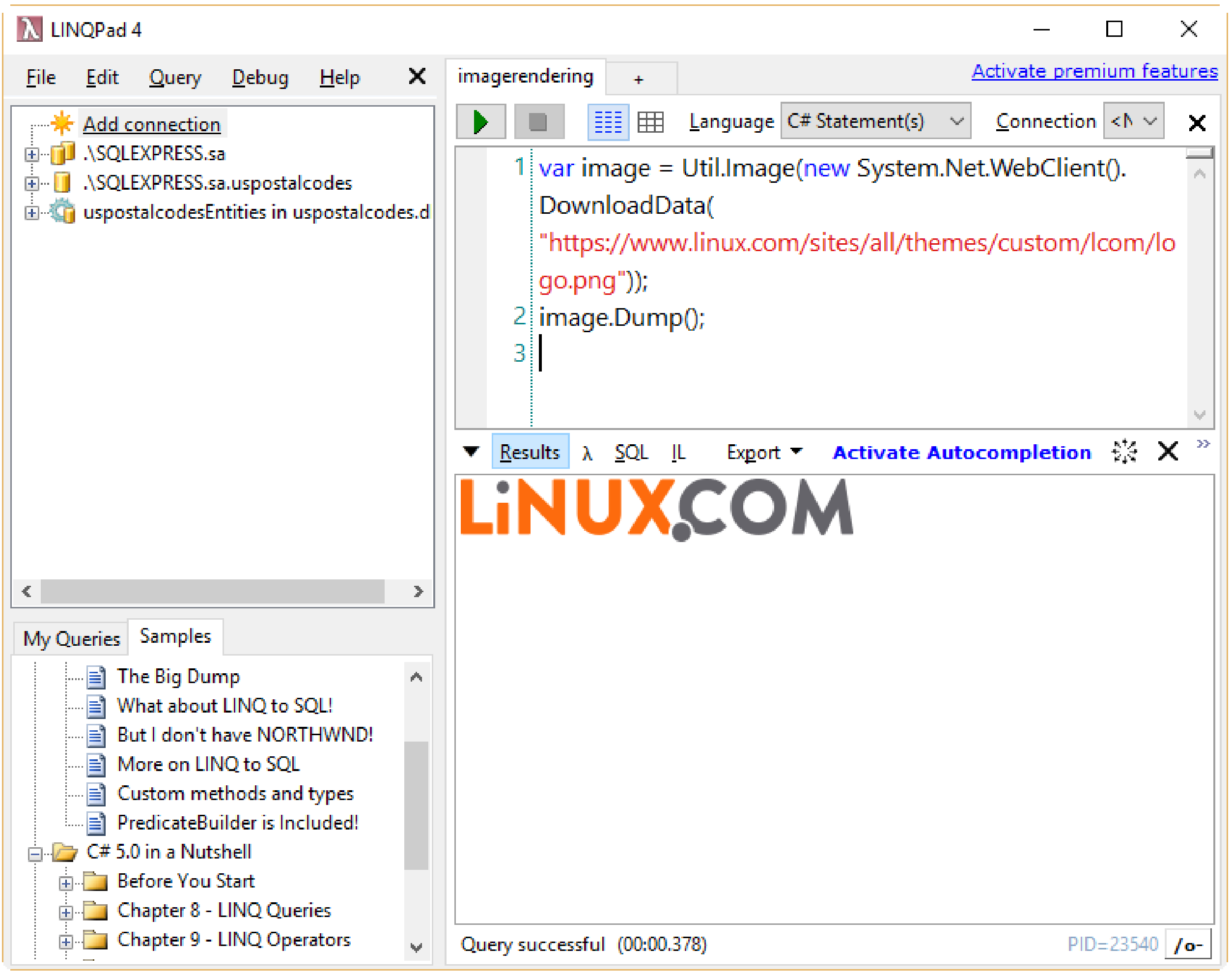
Downloading an image from a web location

The **Dump** method can render almost any kind of objects; images are not the exception. The following code demonstrates how to render an image that is downloaded from a web location.

Code Listing 43

```
var image = Util.Image(new
System.Net.WebClient().DownloadData("https://www.linux.com/sites/all/themes/custom/
lcom/logo.png"));
image.Dump();
```

The output displayed by this code is shown in Figure 53.



```
var image = Util.Image(new System.Net.WebClient().
DownloadData(
"https://www.linux.com/sites/all/themes/custom/lcom/lo
go.png"));
image.Dump();
```

Figure 53: An image rendered by the Dump method

Note: The previous example needs the System.Net assembly to be referenced, as explained in the "Referencing custom and default assemblies" section of this chapter.

The Util class

LINQPad exposes a class called **`Util`**. This class has many useful methods and properties that can be used to interact with the queries. This section will demonstrate some of the methods and properties available in the **`Util`** class.

ClearResults method

The **`ClearResults`** method wipes out the Results panel.

Code Listing 44

```
var image = Util.Image(new
WebClient().DownloadData("https://www.linux.com/sites/all/themes/custom/lcom/logo.p
ng"));
image.Dump();
Thread.Sleep(2000);  //Stops query execution for two seconds
Util.ClearResults();
```

This code sample renders an image downloaded from a web location. Then, the query execution is stopped for two seconds. After that, the **`ClearResults`** method clears the image from the Results panel.

Cmd method

The **`Cmd`** method executes a system command and displays the result. The following sample shows the contents of the C:\LINQPad_Succinctly\Queries folder.

Code Listing 45

```
Util.Cmd("dir C:\\LINQPad_Succinctly\\Queries");
```

The output displayed should look like the following snippet.

Code Listing 46

```
Volume in drive C has no label.
 Volume Serial Number is AA4F-79E0

 Directory of C:\LINQPad_Succinctly\Queries

06/26/2016  08:45 AM    <DIR>          .
06/26/2016  08:45 AM    <DIR>          ..
05/31/2016  12:28 PM               416 A simple query expression.linq
06/16/2016  10:19 AM               654 buttonprototype.linq
06/16/2016  11:25 AM               826 buttonprototypechanged.linq
06/24/2016  10:39 PM             1,025 catchingwpfelementevent.linq
06/21/2016  12:18 PM             1,046 cmdlineoptions.txt
06/17/2016  10:17 AM               239 CmdMethod.linq
06/22/2016  01:50 PM            43,681 cmdsymbolsample.html
06/22/2016  01:49 PM               950 cmdsymbolsample.linq
06/11/2016  02:36 PM               646 countiesortedentityframework.linq
06/10/2016  06:33 PM               989 CountiesPredicateKeywords.linq
06/10/2016  09:40 AM               909 CountiesPredicateSample.linq
06/10/2016  11:37 AM               979 CountiesPredicateSample_2.linq
06/22/2016  12:15 PM               473 custombutton.txt
              13 File(s)        135,723 bytes
               2 Dir(s)  65,482,702,848 bytes free
```

GetMyQueries method

The **GetMyQueries** method returns all the queries located in the default queries folder and displayed under the My Queries tree view node. The method returns an **IEnumerable<Query>** object. Each element in the enumeration is a reference to a **Query** object, which has the following properties.

- **OpenLink**: Contains a hyperlink with the name of the query. When this hyperlink is clicked, the query is opened in a separate tab.
- **Location**: Contains the query file's relative path, starting from the default queries folder. The default queries folder is considered the root folder (\).
- **Name**: The name of the query in plain text.
- **Text**: The query's code.
- **Language**: The kind of programming language, according to those explained in the "How C# code can be executed in LINQPad" section of this chapter.
- **FileReferences**: An **IEnumerable<string>** object containing the full filename for all references added to the query, as explained in the "Referencing custom and default assemblies" section of this chapter.
- **GacReferences**: An **IEnumerable<string>** object containing the full filename for all references from the Global Assemblies Cache (GAC) added to the query, as explained in the "Referencing custom and default assemblies" section of this chapter.
- **NamespaceImports**: An **IEnumerable<string>** object containing the name of all namespaces imported into the query, as explained in the "Importing namespaces into queries" section of this chapter.
- **LastModified**: The date and time when the query was last updated.

The following code displays all queries available.

Code Listing 47

```
Util.GetMyQueries().Dump("My Queries listing");
```

WriteCsv method

The **WriteCsv** method stores the contents of an object into a CSV file. The following sample creates a CSV file from a **Where** query made to the **Counties** table.

Code Listing 48

```
var csvFile = Util.CurrentQueryPath.Replace(".linq",".csv");
Util.WriteCsv(Counties.Where(row => row.State_id == 29),csvFile);
```

The first line of the previous code uses the **CurrentQueryPath** property of the **Util** class (which returns the filename of the query currently running, including its absolute path) along with the **Replace** method, in order to build the filename for saving the CSV contents. This filename is stored in the **csvFile** variable. The **WriteCsv** method receives the result of querying the **Counties** table that is applying the **Where** method, and then writes this result to the file specified into the **csvFile** variable. Now, if you look into the queries default folder, you will find a CSV file named like the query (except for the .csv extension).

Note: Since there's no Dump call in the previous sample, the Results panel displays nothing. Also, in order to retrieve the filename for the query with the CurrentQueryPath property, this must be saved before being executed.

Run method

The **`Run`** method executes a LINQPad query from within another query, .NET program, or Windows Service. In case of a .NET program or Windows Service, the LINQPad.exe file should be referenced within them.

The **`Run`** method executes the query just as the command line tool lprun.exe (which will be explained in the following chapter) would do it.

The following sample uses the **`Run`** method to execute the **`getmyqueries.linq`** query located in the default queries folder.

Code Listing 49

```
const string queriesPath = @"C:\LINQPad_Succinctly\Queries\";
var dummyFormat = new LINQPad.QueryResultFormat();
Util.Run(queriesPath+"getmyqueries.linq",dummyFormat).Dump();
```

Note: Unlike the formatted output shown when a query is executed from LINQPad interface, the output generated by the Run method is in plain text format.

ProgressBar class

The **`ProgressBar`** class allows you to track the progress for a query's execution, displaying a progress bar in the Results panel.

Code Listing 50

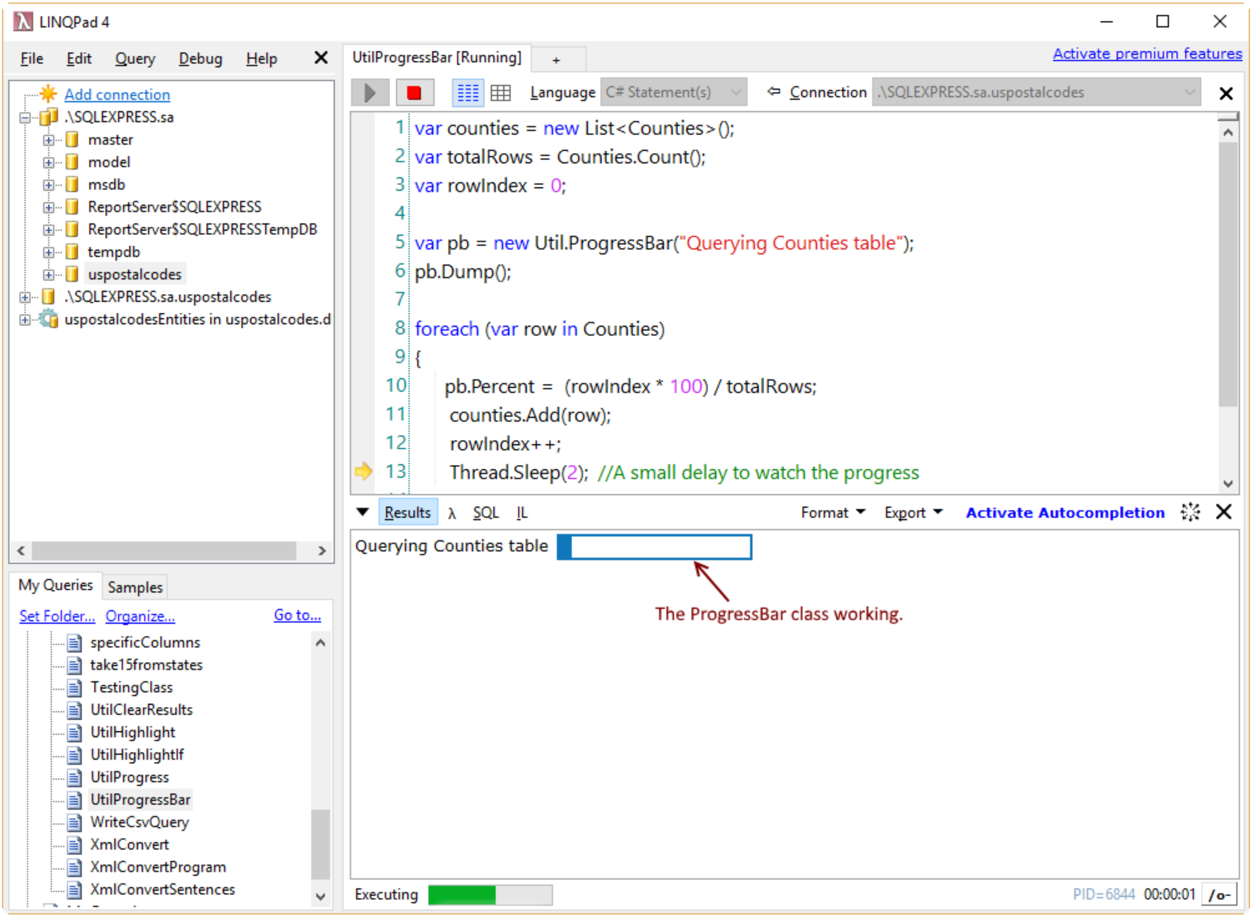
```
var counties = new List<Counties>();
var totalRows = Counties.Count();
var rowIndex = 0;

var pb = new Util.ProgressBar("Querying Counties table");
pb.Dump();

foreach (var row in Counties)
{
      pb.Percent =  (rowIndex * 100) / totalRows;
       counties.Add(row);
          rowIndex++;
          Thread.Sleep(2);  //A small delay to watch the progress
}
pb.Percent = 100;
counties.Dump();
```

This code stores the rows from the **`Counties`** table into a **`List<Counties>`** variable named **`counties`**, using a **`foreach`** loop. The **`ProgressBar`** class is used to display the progress of the operation within the loop. As displayed in the code, a message can be assigned to the progress bar when the class is instantiated.

Note that the **`Dump`** method for the instance of the **`ProgressBar`** class is called before entering the loop. This makes the progress bar visible prior to iterate the table.

The **`Thread.Sleep`** method adds a delay of two milliseconds after adding each row to the list, in order to make the user watch the progress of the operation.

The work performed by the **`ProgressBar`** class is shown in the following figure.



```
var counties = new List<Counties>();
var totalRows = Counties.Count();
var rowIndex = 0;

var pb = new Util.ProgressBar("Querying Counties table");
pb.Dump();

foreach (var row in Counties)
{
    pb.Percent =  (rowIndex * 100) / totalRows;
     counties.Add(row);
     rowIndex++;
     Thread.Sleep(2);  //A small delay to watch the progress
```

```
Querying Counties table
```

Figure 54: The progress bar displaying the advance

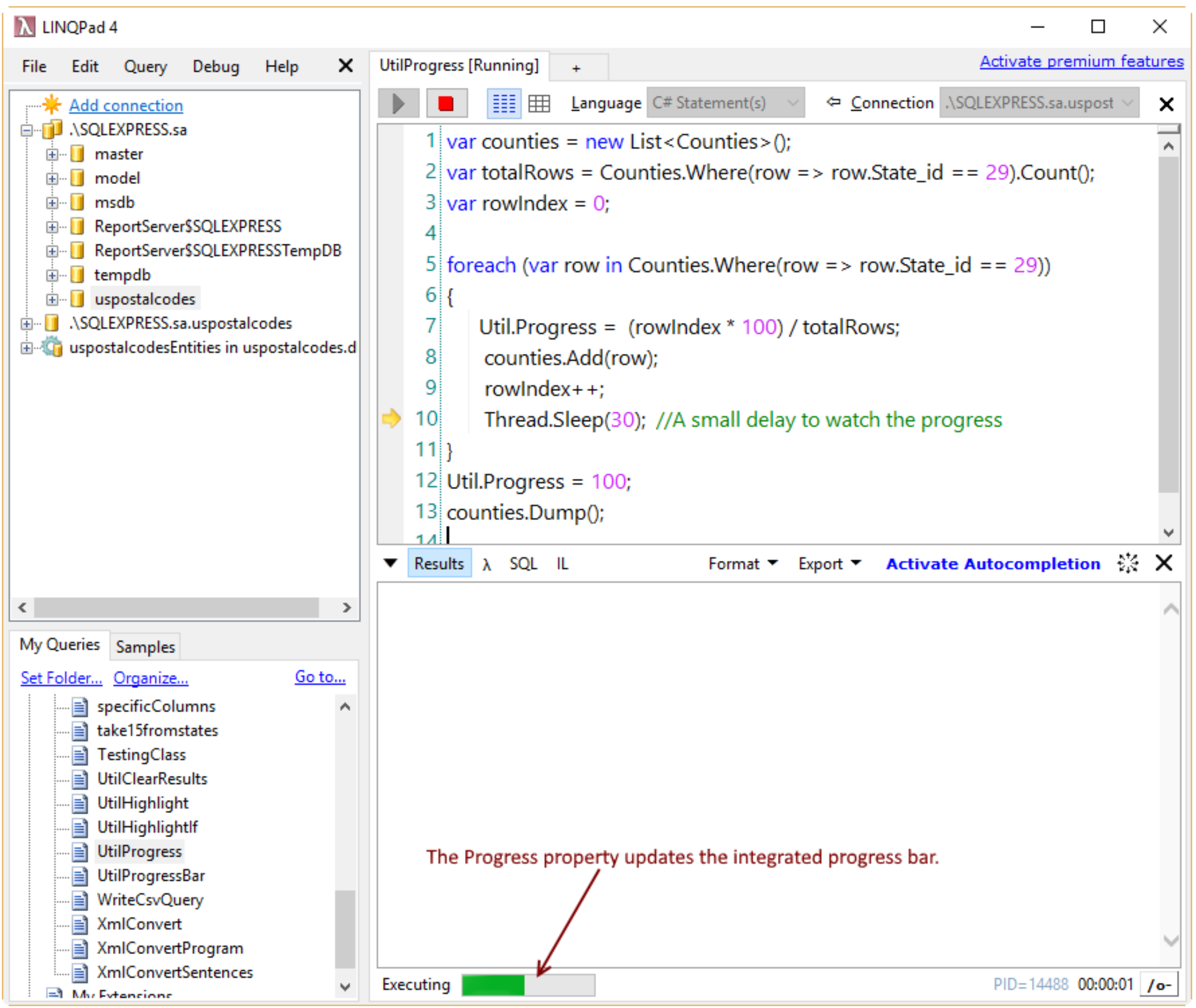
Progress property

The **`Progress`** property allows you to update the progress bar integrated in the status bar of LINQPad. In this case, the progress bar is not displayed in the Results panel, and the user cannot assign a message to it.

Code Listing 51

```
var counties = new List<Counties>();
var totalRows = Counties.Where(row => row.State_id == 29).Count();
var rowIndex = 0;

foreach (var row in Counties.Where(row => row.State_id == 29))
{
       Util.Progress =  (rowIndex * 100) / totalRows;
        counties.Add(row);
           rowIndex++;
           Thread.Sleep(30);  //A small delay to watch the progress
}
Util.Progress = 100;
counties.Dump();
```

The previous code is similar to the code used to explain the **`ProgressBar`** class, except for the use of the **`Progress`** property. The following figure shows how the **`Progress`** property works.



```
var counties = new List<Counties>();
var totalRows = Counties.Where(row => row.State_id == 29).Count();
var rowIndex = 0;

foreach (var row in Counties.Where(row => row.State_id == 29))
{
    Util.Progress =  (rowIndex * 100) / totalRows;
     counties.Add(row);
     rowIndex++;
     Thread.Sleep(30);  //A small delay to watch the progress
}
Util.Progress = 100;
counties.Dump();
```

The Progress property updates the integrated progress bar.

Figure 55: Updating the integrated progress bar

Chapter summary

LINQPad has the ability to be used as a test and prototyping tool, and can run code in the C#, F#, and VB languages.

C# code can be executed in LINQPad as a single line expression, as several C# sentences, and as a C# program with a **`Main`** method.

LINQPad gives the user the ability to measure a query's performance by displaying the time consumed by query's execution in the status bar.

Every time LINQPad executes a query, the results are displayed in a window known as the Results panel. This window offers some useful features by way of six graphic elements: the Options Menu drop-down button, Results View Mode options, Format Menu, Export Menu, Unlock Results button, and Close button.

LINQPad also allows the use of custom assemblies in queries. These assemblies can be referenced in a query by selecting the **Query Properties** command from the **Query** menu, or by pressing the **F4** key. The user can pick up an assembly from those installed in the system, browse for a custom assembly file into the local filesystem, or add references for both. The users of the LINQPad Premium edition can also reference NuGet packages. In the Free edition, only those packages used in LINQPad examples can be referenced.

When custom or system assemblies are referenced in queries, the use should declare the classes or methods belonging to those assemblies by an explicitly qualified name. That is, to use the **`Button`** control located in the **`System.Windows.Forms`** namespace, the user should write **`System.Windows.Forms.Button`** in order to create an instance for the class. To simplify the way the code is written, LINQPad allows you to import namespaces into queries using the Additional Namespace Imports tab located in the Query Properties dialog box.

The **`Dump`** method is used to display the output produced by a query. This method also has tremendous capabilities that can help the user to render a Windows Form control and download and render an image from a web location. In LINQPad5, the **`Dump`** method has additional parameters to conditionally display the output in the screen.

As a final point, LINQPad exposes a static class called **`Util`**. This class has many useful methods and properties that can be used to interact with the queries in order to run system commands, obtain all available queries in a list, highlight the results for a query, export results to a CSV file, or display progress bars.

Chapter 7 General Summary

LINQPad is a software utility intended to be used as a part of .NET applications development. It is used for interactively querying SQL databases using LINQ, as well as interactively writing C# code without the need for an IDE, such as Visual Studio. LINQPad supports the VB.NET, SQL, and F# languages. LINQPad also supports Entity Framework, LINQ to Objects, LINQ to SQL, and LINQ to XML interoperability.

There are three versions of the product: LINQPad 2.x (for .NET Framework 3.5), LINQPad 4 (for .NET Framework 4.0/4.5), and LINQPad 5 (for .NET Framework 4.6). There are four editions for each of these versions: Free, Pro, Developer, and Premium.

LINQPad enables extra features for the paid license versions. Some of these features include full C#/F#/VB auto completion and tooltips, code outlining and formatting for C#/VB, built-in and custom code snippets, and full NuGet integration.

LINQPad is a client/server application that works as one client and many servers. In this case, the LINQPad user interface is the client, and for each query to be executed, LINQPad creates a separate server. For communications between client and servers, LINQPad uses a customized communications layer that runs using Windows shared memory.

LINQPad uses third-party libraries from several software publishers within the UI. These includes Actipro's SyntaxEditor, the SharpDevelop project, Microsoft Roslyn, and NRefactory. The libraries from these publishers are included as embedded resources in LINQPad, so it ships as a single executable. All updates for the product are downloaded into the application's data folder. Every time it's started, LINQPad always checks to see if there's a later signed version of itself in the application's data folder.

LINQPad software can be downloaded from the following locations:

- LINQPad 5
- LINQPad 4

After you download LINQPad, double-click the **LINQPad4Setup.exe** (or **LINQPad5Setup.exe**) file to begin the installation.

The installation process is guided by a series of dialog boxes that allow you to select the program's location folder, files associations, icons creation, and environment variables setting for command line support. When LINQPad installation is complete, a set of folders and files are created that include queries, plugins, code snippets, data context drivers, connections, default references, and user preferences.

LINQPad can also be deployed as a portable application. To accomplish this task, a special directory structure should be created in a USB memory stick, including the linqpad.exe file within it.

To start working with LINQPad, double-click on the icon placed on the Windows desktop. This will launch the LINQPad user interface, which is the client application that will do most of the work.

For the purposes of this book, we identified 14 graphic elements in the LINQPad user interface: the LINQPad main menu, Connection's tree view area, Queries and Samples tree view area, Query code tab window area, Query code tab caption, Add New Tab button, Execute Query button, Stop Execution button, Results View Type button, Language Selector combo box, Connection Selector combo box, Close Tab button, Premium Features activation link, and Status Bar.

The editor window is one of the user interface elements that can be customized. To do so, select the **Preferences** command from the **Edit** menu, and the settings for the window can be changed in the **Editor** tab of the **Preferences** dialog box. For the purposes of this book, the Segoe UI font was used, and also the code line numbers were displayed, too. Font size customization is not available because LINQPad allows the user to increase or decrease the font size using Ctrl+Plus (increase) or Ctrl+Minus (decrease), or by using the mouse's scroll wheel while pressing the Ctrl key.

When working with LINQPad, a basic concept is the term *query*. A query is a piece of code that can be executed immediately. All written queries are displayed beneath a tree view node called My Queries. Physically, LINQPad stores queries in a folder named LINQPad Queries, located into the user's Documents folder. The location of this folder can be changed by clicking the **Set Folder** hyperlink in the user interface, and browsing for a new folder in the **Preferences** dialog box. All queries discussed in this book were saved in a folder named **C:\LINQPad_Succinctly\Queries**.

LINQPad allows you to search queries by using a term. This can be done by clicking on the **Go to** hyperlink located on the **My Queries** tree view node in the user interface. Then, you can type the term in the **Navigate to Query** dialog box, and LINQPad will show all queries that match that term.

LINQPad queries can be written in 10 different programming modes: C# Expression, C# Statements, C# Program, VB Expression, VB Statement, VB Statements, VB Program, SQL, ESQL, F# Expression, and F# Program. Each of these languages can be selected in the **Language** selector combo box, located at the top of the editor window.

All query filenames are associated with the .linq file extension. Internally, a header in XML format is added at the beginning of every query file. This header tells LINQPad which kind of programming language is used to compile the query, the connection used by the query to retrieve data, and which assemblies and namespaces are referenced in the query. This XML header is not displayed in the editor window.

You can manage connections to SQL servers in the Connection's tree view area by clicking on the **Add Connection** hyperlink. Then, the Choose Data Context dialog displayed on the screen allows you to choose which LINQPad driver will be used. For purposes of this book, the **LINQ to SQL driver** (which is the default) option was chosen for all queries.

After the LINQPad driver is chosen, the LINQPad Connection dialog box is shown to set the connection's parameters. A Microsoft SQL Server connection was created, assuming that a SQLEXPRESS instance configured with SQL Authentication security mode was installed in the local computer.

LINQPad names a connection using the server name followed by a period and the name of the authenticated user when SQL Authentication is chosen. If Windows Authentication is selected, the name for the connection contains the server name only. Also, a context menu is available for all connections displayed in LINQPad user interface. This menu appears when the user right-clicks on a connection's name. The menu contains a set of commands for managing the connection, including connection properties or connection refreshing.

LINQPad can be used to create database queries employing the default LINQ-to-SQL driver. We used a database named **uspostalcodes** in the samples discussed in this book. The database was created using the Choose Data Context dialog box, along with the LINQ to SQL Connection dialog box. Because there's no way to create database entities with LINQ-to-SQL, Microsoft SQL Server Management Studio was used to create the database tables and to populate them with data.

The LINQ-to-SQL methods discussed in this book include the **Take** method, which returns a row set with a specific rows number, and the **OrderBy** method, which sorts the rows returned by a method. Some of these methods require a lambda expression in order to work properly. A lambda expression is composed by a parameter and an expression, which takes the parameter and uses it to return a value. The term *lambda* expression comes from lambda calculus, which is a mathematical notation applied for defining functions. The methods discussed in this book that use a lambda expression are: **`OrderBy`** (to sort rows), **`Where`** (to filter rows based on a conditional expression), **`Sum`** (to summarize the value of a given column), **`Average`** (to get an average from a series of values of a given column), and **`GroupBy`** (to group rows based on the value of a given column).

LINQPad provides a special class named **`PredicateBuilder`**. This class is intended to simplify queries writing when a complex filter condition is needed. The class solves this issue by providing a set of special methods intended to act as logical comparison operators. Because **`PredicateBuilder`** is a generic class, it can receive any entity reference of the database used by a connection. The value proposition for the **`PredicateBuilder`** class resides in its ability to build any lambda expression dynamically when the instance of the class is passed to any LINQ-to-SQL method. The expression is created using the values passed to the class instance using the **`And`** or the **`Or`** methods. Both methods should be called repeatedly before using the instance. For using **`PredicateBuilder`** class in queries, you must add a reference to the class into any query which employs it. This can be done by pressing **F4** and checking the **Include PredicateBuilder** check box that appears in the **Query Properties** dialog box.

One of the most valuable features of LINQPad is the ability to use it as a test and prototyping tool. LINQPad can run code in the C#, F#, and VB languages for that purpose. Each one of these languages can be executed as a single-line expression, as several sentences, and as a program. In all cases, the performance for a given code can be measured by reading the time consumed by the execution from the status bar. This feature allows developers to improve code before a product's release. LINQPad also allows the use of custom assemblies in queries. These assemblies can be referenced by selecting the **Query Properties** command from the **Query** menu, or by pressing the **F4** key. The user can pick up an assembly from those installed in the system, or browse for a custom assembly file into the local filesystem, or both. Users of the LINQPad Premium edition also can reference NuGet packages, while the users of the Free edition can reference only those packages used in LINQPad examples.

When custom or system assemblies are referenced in queries, the user should declare the classes or methods belonging to those assemblies by an explicitly qualified name. That is, to use the **`Button`** control located in the **`System.Windows.Forms`** namespace, the user should write **`System.Windows.Forms.Button`** in order to create an instance for the class. To simplify the way that the code is written, LINQPad allows you to import namespaces into queries using the **Additional Namespace Imports** tab located in the **Query Properties** dialog box.

LINQPad uses a method called **`Dump`** to display the output produced by a query. This method renders any kind of object, such as Windows Form control, an image from a web location, or the results of a LINQ-to-SQL query.

Another useful class shipped with LINQPad is the **`Util`** class. This class has many useful methods and properties that can be used to interact with queries in order to run system commands, obtaining all available queries in a list, highlighting the results for a query, exporting results to a CSV file, or displaying progress bars.

General Conclusions about LINQPad

Since LINQPad was added to our toolbox, the development process for any of our products has improved a lot. LINQPad has helped us in various stages of the development lifecycle, such as testing, or products prototyping. LINQPad has become a valuable tool for LINQ training, allowing our developers to take advantage of LINQ's features in order to create smaller and more efficient pieces of code.

Before LINQPad, our main repository space was almost 20 percent consumed by single-use Visual Studio projects. Most of those projects were created for testing purposes. Now, all those projects have been transformed into LINQPad queries that can be used more than once. Therefore, repository management has become easier, and the space consumed in our storage devices has decreased.

Another useful thing we've taken advantage of is LINQPad's ability to import data sources and easily interact with them. Because the program allows us to manage several connections with the power of a click, many data models can be taken from the diagrams tool to the real world, and they can be validated in less time than those employed with traditional tools. Also, our database designers have the power to retrieve data using LINQ-to-SQL queries, and to turn these queries into SQL Views or Stored Procedures by simply selecting the SQL commands automatically generated by LINQPad.

The samples that come with LINQPad have been a tremendous knowledge source, and when these samples were not enough, downloading more samples from the web was quite easy.

In the general opinion of our development team, LINQPad is an awesome product.

Appendix

LINQPad keyboard shortcuts

Execution

F5 or Alt+X or Ctrl+G	Execute a query
Shift+F5	Cancel an executing query

Query properties

Alt+C	Change Connection
Ctrl+D	Use Connection/Database selected in Schema Explorer (toggle)
Ctrl+1 through Ctrl+9	Change query language: Ctrl+1 — C# Expression Ctrl+2 — C# Statements Ctrl+3 — C# Program Ctrl+4 — VB Expression Ctrl+5 — VB Statements Ctrl+6 — VB Program Ctrl+7 — SQL

Ctrl+8	ESQL
Ctrl+9	F# Expression
Ctrl+0	F# Program

Window management

Ctrl+Tab	Most recently activated query
Shift+Ctrl+Tab	Least recently activated query
Ctrl+PageDown	Next query
Ctrl+PageUp	Previous query
Alt+W	Show all open queries in Window
F7	View Code
Alt+Left / Alt+Right	Switch between output panels
Alt+Up / Alt+Down Alt+PageUp / Alt+PageDown Alt+Home / Alt+End	Scroll output panels
Alt+R	View/focus results
Alt+A	View lambda translation
Alt+S	View SQL translation
Alt+I	View IL translation

Shift+Alt+S	View Schema Explorer
F6	View My Queries / Samples
Shift+F8	Show/Hide Explorer panels

Editor

Ctrl+.	Open smart tag
Ctrl+E, C or Ctrl+K, C	Comment selected lines
Ctrl+E, U or Ctrl+K, U	Uncomment selected lines
Ctrl+Plus	Increase editor font size
Ctrl+Minus	Decrease editor font size
Ctrl+F3	Find next selected
Shift+Ctrl+F3	Find previous selected

Mouse shortcuts

Middle-click query in **My Queries** or **Sample Queries**	Open the query in a new window
Middle-click query tab	Close the query
Drag database from Schema Explorer to query	Set the query's database
Ctrl + drag database from Schema Explorer to query	Attach additional database to query (LINQPad Premium users)