O'REILLY®

Covers Version 9.0

C# Cookbook

Modern Recipes for Professional Developers

Early Release

RAW & UNEDITED

Joe Mayo

C# Cookbook

Modern Recipes for Professional Developers

Joe Mayo

C# Cookbook

by Joe Mayo

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

- Editors: Angela Rufino and Amanda Quinn
- Production Editor: Katherine Tozer
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- October 2021: First Edition

Revision History for the Early Release

- 2020-11-11: First Release
- 2021-02-03: Second Release
- 2021-03-16: Third Release
- 2021-05-11: Fourth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492093695 for release details.

978-1-492-09363-3

Chapter 1. Constructing Types and Apps

1.1 Overview

One of the first things we do as developers is to design, organize, and create new types. This chapter helps with these tasks by offering several useful recipes for initial project setup, managing object lifetime, and establishing patterns.

Establishing Architecture

When you’re first setting up a project, you have to think about the overall architecture. There’s a concept called separation of concerns where each part of an application has a specific purpose. e.g. The UI layer interacts with users, a business logic layer manages rules, and a data layer interacts with a data source. Each layer has a purpose or responsibilities and contains the code to perform its operations.

In addition to promoting more loosely coupled code, separation of concerns makes it easier for developers to work with that code because it’s easier to find where a certain operation occurs. This makes it easier to add new features and maintain existing code. The benefits of this are higher quality applications and more productive work. It pays to get started right, which is why we have a section on Designing Application Layers later in this chapter.

Applying Patterns

A lot of the code we write is Transaction Script, where the user interacts with a

UI and the code performs some time of Create, Read, Update, or Delete (CRUD) operation in the database and returns the result. Other times, we have complex interactions between objects that are difficult to organize. In that case, we need other patterns to solve these hard problems.

This chapter presents a few useful patterns in a rather informal manner. The purpose is so you can have some code to rename and adapt to your purposes and rationale on when a given pattern would be useful. As you read through each pattern, try to think about other code you've written or other situations where that pattern would have simplified the code.

First, there's Inversion of Control (IoC), which helps decouple code and promotes testability. The section on Removing Explicit Dependencies explains how this works. When we look at Ensuring Quality, in Chapter 3, you'll see how IoC fits in to unit testing.

If you run into the problem of having different APIs to different systems and needing to switch between them, you'll be interested in reading the Making Classes Adapt to your Interface section. This shows how to build a single interface to solve this problem.

Managing Object Lifetime

Other important tasks we perform are creating objects and managing their lifetime. The Delegating Object Creation sections show a couple nice factory patterns that let you decouple object creation from code. This fits in with the IoC concepts, just discussed.

Another way to manage object creation is through a fluid interface, where you can include optional settings, via methods, and validate before object construction.

Another important object lifetime consideration is disposal. Think about the drawbacks to excessive resource consumption; whether excessive memory use, file locks, or any other object that holds operating system resources. These problems often result in application crashes and are difficult to detect and fix. Performing proper resource cleanup is so important that it's the first section of this chapter.

1.2 1.1 Managing Object End-of-Lifetime

Problem

Your application is crashing because of excessive resource usage.

Solution

Here’s the object with the original problem:

```
using System;
using System.IO;

public class DeploymentProcess
{
    StreamWriter report = new StreamWriter("DeploymentReport.txt");

    public bool CheckStatus()
    {
        report.WriteLine($"{DateTime.Now} Application Deployed.");

        return true;
    }
}
```

And here’s how to fix the problem:

```
using System;
using System.IO;

public class DeploymentProcess : IDisposable
{
    private bool disposedValue;

    StreamWriter report = new StreamWriter("DeploymentReport.txt");

    public bool CheckStatus()
    {
        report.WriteLine($"{DateTime.Now} Application Deployed.");

        return true;
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposedValue)
        {
            if (disposing)
            {
                report?.Close();
            }

            disposedValue = true;
        }
    }

    // ~DeploymentProcess()
    // {
    //     Dispose(disposing: false);
    // }

    public void Dispose()
    {
        Dispose(disposing: true);
        GC.SuppressFinalize(this);
    }
}
```

Discussion

The problem in this code was with the `StreamWriter`, report. Whenever you're using some type of resource, such as the report file reference, you need to release (or dispose) that resource. The specific problem here occurs because the Windows OS adds locks to files, supposedly to protect the application that has ownership. However, what happens all too often is that the OS doesn't release that file lock on its own. That's the responsibility of your application - to tell Windows to release that file lock. So, the next time you run this application, it might not be able to write to the file. Further, if another application in a processing pipeline wanted to read that file, it wouldn't be able to. In the worst case, everything crashes in a hard-to-find scenario that involved multiple people over a number of hours debugging a critical production problem.

The solution is to implement the Dispose pattern, which involves adding code that makes it easy to release resources. The solution code implements the `IDisposable` interface. `IDisposable` only specifies the `Dispose()` method, without parameters and there's more to be done than just adding that method, including another `Dispose` method overload that keeps track of what type if disposal to do and an optional finalizer.

Complicating the implementation is a field and parameter that control disposal logic: `disposed` and `disposing`. The `disposed` field ensures that this object gets disposed only one time. Inside the `Dispose(bool)` method, there's an `if` statement, ensuring that if `disposed` is `true` (the object has been disposed) then it won't execute any disposal logic. The first time through `Dispose(bool)`, disposed will be `false` and the code in the `if` block will execute. Make sure that you also set `disposed` to `true` to ensure this code doesn't run any more - the consequences of not doing so bring exposure to unpredictable errors like `NullReferenceException` or `ObjectDisposedException`.

The disposing parameter tells `Dispose(bool)` how it's being called. Notice that `Dispose()`, without parameters, and the finalizer call `Dispose(bool)`. When `Dispose()` calls `Dispose(bool)`, `disposing` is `true`. This means that calling code is acting properly by instantiating `DeploymentProcess` in a `using` statement or wrapping it in the finally block of a `try/finally`.

The finalizer calls `Dispose(bool)` with `disposing` set to `false`, meaning that it isn't being run by calling application code. The `Dispose(bool)` method uses the `disposing` value to determine whether it should release managed resources. If you have unmanaged resources, you can release them any time.

Let's clarify what is happening with the finalizer. The .NET CLR Garbage Collector (GC) executes an object's finalizer when it cleans that object from memory. The GC can make multiple passes over objects and calling finalizers is one of the last things it does. Managed objects are instantiated and managed by the .NET CLR and you don't have control over when they're released, which could potentially be out-of-order. That's why you have to check the disposing value, to prevent an `ObjectDisposedException` in case the dependent object was disposed by the GC first.

What the finalizer gives you is a way to clean up unmanaged resources. An unmanaged resource doesn't belong to the .NET CLR, it belongs to the Windows OS. There are situations where developers might need to call into a Win32/64 DLL to get a handle to an OS or 3rd party device. The reason you need the finalizer is because if your object doesn't get disposed properly, there's no other way to release that handle, which could crash your system for the same reason we need to release managed objects. So, the finalizer is a just-in-case mechanism to ensure the code that needs to release the unmanaged resource will execute.

A lot of applications don't have objects that use unmanaged resources. In that case, don't even add the finalizer. Having the finalizer adds overhead to the object because the GC has to do accounting to recognize objects that do have finalizers and call them in a multi-pass collection. Omitting the finalizer avoids this.

On a related note, remember to call `GC.SuppressFinalize` in the `Dispose()` method. This is another optimization telling the GC to not call the finalizer for this object, because all resources, managed and unmanaged, are released when the application calls `IDisposable.Dispose()`.

1.3 1.2 Removing Explicit Dependencies

Problem

Your application is tightly coupled and difficult to maintain.

Solution

Define the types you need:

```
public class DeploymentArtifacts
{
    public void Validate()
    {
        System.Console.WriteLine("Validating...");
    }
}

public class DeploymentRepository
{
    public void SaveStatus(string status)
    {
        System.Console.WriteLine("Saving status...");
    }
}
```

```
interface IDeploymentService
{
    void PerformValidation();
}

public class DeploymentService : IDeploymentService
{
    readonly DeploymentArtifacts artifacts;
    readonly DeploymentRepository repository;

    public DeploymentService(
        DeploymentArtifacts artifacts,
        DeploymentRepository repository)
    {
        this.artifacts = artifacts;
        this.repository = repository;
    }

    public void PerformValidation()
    {
        artifacts.Validate();
        repository.SaveStatus("status");
    }
}
```

And start the application like this:

```
using Microsoft.Extensions.DependencyInjection;
using System;

class Program
{
    public readonly IDeploymentService service;

    public Program(IDeploymentService service)
    {
        this.service = service;
    }

    static void Main()
    {
        var services = new ServiceCollection();

        services.AddTransient<DeploymentArtifacts>();
        services.AddTransient<DeploymentRepository>();
        services.AddTransient<IDeploymentService, DeploymentService>
();
```

```
            ServiceProvider serviceProvider =
    services.BuildServiceProvider();

            IDeploymentService deploymentService =
                serviceProvider.GetRequiredService<IDeploymentService>();

            var program = new Program(deploymentService);

            program.StartDeployment();
        }

        public void StartDeployment()
        {
            service.PerformValidation();
            Console.WriteLine("Validation complete - continuing...");
        }
    }
```

Discussion

The term, Tightly Coupled, often means that one piece of code, is overburdened with the responsibility of instantiating the types (dependencies) it uses. This requires the code to know how to construct, manage lifetime, and contain logic for dependencies. This distracts from the purpose of the code in solving the problem it exists for. It duplicates instantiation of dependencies in different classes. This makes the code brittle because changes in dependency interfaces affects all other code that needs to instantiate that dependency. Additionally, code that instantiates its dependencies makes it difficult, if not impossible, to perform proper unit testing.

The solution is Dependency Injection, which is a technique to define dependency type instantiation in one place and expose a service that other types can use to obtain instances of those dependencies. There are a couple ways to perform dependency injection: Service Locator and Inversion of Control (IoC). Which to use and when is an active debate and we avoid venturing into theoretical territory. To keep things simple, this solution uses IoC, which is a common and straight-forward approach.

The specific solution requires that you have types that rely on other dependency types, configure type constructors to accept dependencies, reference a library to help manage IoC container, and use the container to declare how to instantiate types. The following paragraphs explain how this works.

The solution is a utility to help manage a deployment process, validating whether the deployment process is configured properly. It has a `DeploymentService` class that runs the process. Notice that the `DeploymentService` constructor accepts `DeploymentArtifacts` and `DeploymentRepository` classes. `DeploymentService` does not instantiate these classes - rather, they are injected.

To inject these classes, you can use an IoC container, which is a library that helps to automatically instantiate types and to automatically instantiate and provide instances of dependency types. The IoC container in the solution, as shown in the using declaration, is `Microsoft.Extensions.DependencyInjection`, which you can reference as the NuGet package by the same name.

While we want to inject all dependencies for every type in the application, you must still instantiate the IoC container directly, which is why the `Main` method instantiates `ServiceCollection` as services. Then use the `services` instance to add all of the dependencies, including `DeploymentService`.

The IoC container can help manage the lifetime of objects. This solution uses `AddTransient`, which means that the container should create a new instance every time it's type is requested. A couple other examples of managing object lifetime are `AddSingleton`, which instantiates an object only one time and passes that one instance to all objects, and `AddScoped`, which gives more control over the lifetime of the object. In ASP.NET, `AddScoped` is set to the current request. Over time, you'll want to think more about what the lifetime of your objects should be and investigate these options in more depth. For now, it's simple to get started with `AddTransient`.

The call to `BuildServiceProvider` converts `services`, a `ServiceCollection`, into a `ServiceProvider`. The term, IoC Container refers to this `ServiceProvider` instance — it instantiates and locates types to be injected.

You can see the container in action, calling `GetRequiredService` to return an instance that implements `IDeploymentService`. Going back to the `ServiceCollection`, notice that there's an `AddTransient` associating the `DeploymentService` class with the `IDeploymentService` interface.

This means that `GetRequiredService` will return an instance of `DeploymentService`.

Finally, `Main` instantiates `Program`, with the new `DeploymentService` instance.

Going back to the constructor for `DeploymentService`, you can see that it expects to be instantiated with instances for `DeploymentArtifacts` and `DeploymentRepository`. Because we used the IoC container to instantiate `DeploymentService`, the IoC Container also knows how to instantiate it's dependencies, which were also added to the `ServiceCollection`, with calls to `AddTransient`. This solution only used three types and you can build object dependency graphs much deeper than this.

Also, notice how the `DeploymentService` constructor saves the injected instances in `readonly` fields, making them available for use by `DependencyService` members.

The beauty of IoC is that instantiation only happens in one place and you don't have to code all of that in your constructors or in members that need a new instance of a dependency. This makes your code more loosely coupled and maintainable. It also opens the opportunity for higher quality by making the type more unit testable.

See Also

3.1 Writing a Unit Test

1.4 1.3 Delegating Object Creation to a Class

Problem

You're using IoC, the type you're trying to instantiate doesn't have an interface, and you have complex construction requirements.

Solution

We want to instantiate this class:

```
using System;

public class ThirdPartyDeploymentService
{
    public void Validate()
    {
        Console.WriteLine("Validated");
    }
}
```

We’ll use this class for IoC:

```
public interface IValidatorFactory
{
    ThirdPartyDeploymentService CreateDeploymentService();
}
```

And here’s the `IValidatorFactory` implementation:

```
public class ValidatorFactory : IValidatorFactory
{
    public ThirdPartyDeploymentService CreateDeploymentService()
    {
        return new ThirdPartyDeploymentService();
    }
}
```

Then instantiate the factory like this:

```
public class Program
{
    ThirdPartyDeploymentService service;

    public Program(IValidatorFactory factory)
    {
        service = factory.CreateDeploymentService();
    }

    static void Main()
    {
        var factory = new ValidatorFactory();
        var program = new Program(factory);
        program.PerformValidation();
    }

    void PerformValidation()
```

```
        {
            service.Validate();
        }
    }
```

Discussion

As discussed in Section 1.2, IoC is a best-practice because it decouples dependencies, making code easier to maintain, more adaptable, and easier to test. The problem is that there are exceptions and situations that cause difficulties with the best of plans. One of these problems is when trying to use a 3rd party API without an interface.

The solution shows a `ThirdPartyDeploymentService` class. Obviously, you can see the code and what it does. In reality, even if you can read the code through reflection or disassembler, it doesn't help because you can't add your interface. Even if a `ThirdPartyDeploymentService` was open-source, you would have to weigh the decision to fork the library for your own modifications - the tradeoff being that your modifications are brittle in the face of new features and maintenance to the original open-source library. An example is the `System.Net.HttpClient` class in the .NET Framework, which doesn't have an interface. Ultimately, you'll need to evaluate the situation and make a decision that works for you, but the Factory Class described here can be an effective work-around.

To see how a Factory Class works, observe the `IValidatorFactory` interface. This is the interface we'll use for IoC.

Next, examine how the `ValidatorFactory` class implements the `IValidatorFactory` interface. It's `CreateDeploymentService` instantiates and returns the `ThirdPartyDeploymentService`. This is what a factory does - it creates objects for us.

To simplify this example, the code doesn't use an IoC container. Instead, the `Main` method instantiates `ValidatorFactory` and passes that instance to the `Program` constructor, which is the important part of this example.

Examine how the constructor takes the `IValidatorFactory` reference and calls `CreateDeploymentService`. Now we've been able to inject the dependency and maintain the loose coupling we sought.

Another benefit is since the `ThirdPartyDeploymentService` is instantiated in the factory class, you can make any future changes to class instantiation without affecting consuming code.

See Also

1.2 Removing Explicit Dependencies

1.5 1.4 Delegating Object Creation to a Method

Problem

You want a plug-in framework and need to structure object instantiation some place other than application logic.

Solution

Here’s the abstract base class with the object creation contract:

```
public abstract class DeploymentManagementBase
{
    private IDeploymenPlugin deploymentService;

    protected abstract IDeploymentPlugin CreateDeploymentService();

    public bool Validate()
    {
        if (deploymentService == null)
            deploymentService = CreateDeploymentService();

        return deploymentService.Validate();
    }
}
```

These are a couple derived classes that instantiate associated plug-in classes:

```
public class DeploymentManager1 : DeploymentManagementBase
{
    protected override IDeploymentPlugin CreateDeploymentService()
    {
        return new DeploymentPlugin1();
    }
}

public class DeploymentManager2 : DeploymentManagementBase
{
    protected override IDeploymentPlugin CreateDeploymentService()
    {
        return new DeploymentPlugin2();
    }
}
```

The plug-in classes implement the `IDeploymentPlugin` interface:

```
public interface IDeploymentPlugin
{
    bool Validate();
}
```

And here are the plug-in classes being instantiated:

```
public class DeploymentPlugin1 : IDeploymentPlugin
{
    public bool Validate()
    {
        Console.WriteLine("Validated Plugin 1");
        return true;
    }
}

public class DeploymentPlugin2 : IDeploymentPlugin
{
    public bool Validate()
    {
        Console.WriteLine("Validated Plugin 2");
        return true;
    }
}
```

Finally, here’s how it all fits together:

```
class Program
{
    DeploymentManagementBase[] deploymentManagers;

    public Program(DeploymentManagementBase[] deploymentManagers)
    {
        this.deploymentManagers = deploymentManagers;
    }

    static DeploymentManagementBase[] GetPlugins()
    {
        return new DeploymentManagementBase[]
            {
                new DeploymentManager1(),
                new DeploymentManager2()
            };
    }

    static void Main()
    {
        DeploymentManagementBase[] deploymentManagers = GetPlugins();

        var program = new Program(deploymentManagers);

        program.Run();
    }

    void Run()
    {
        foreach (var manager in deploymentManagers)
            manager.Validate();
    }
}
```

Discussion

Plug-In systems are all around us. Excel can consume and emit different document types, Adobe works with multiple image types, and Visual Studio Code has numerous extensions. These are all plug-in systems and whether the only plug-ins available are via vendor or 3rd party, they all leverage the same concept - the code must be able to adapt to handling a new abstract object type.

While the previous examples are ubiquitous in our daily lives, many developers won’t be building those types of systems. That said, the plug-in model is a powerful opportunity for making our applications extensible. Application integration is a frequent use case where your application needs to consume documents from customers, other departments, or other businesses. Sure, Web services and other types of APIs are popular, but needing to consume an Excel spreadsheet is normal. As soon as you do that, someone has data in a different format, like CSV, JSON, Tab Delimited, and more. Another side of the story is the frequent need to export data in a format that multiple users need to consume.

In this spirit, the solution demonstrates a situation where a plug-in system allows an application to add support for new deployment types. This is a typical situation where you've built the system to handle the deployment artifacts that you know about, but the system is so useful that everyone else wants to add their own deployment logic, which you never knew about when original requirements were written.

In the solution, each of the `DeploymentManagers` implement the abstract base class, `DeploymentManagementBase`. `DeploymentManagementBase` orchestrates the logic and the derived `DeploymentManager` classes are simply factories for their associated plugins. Notice that `DeploymentManagementBase` uses polymorphism to let derived classes instantiate their respective plug-in classes.

TIP

If this is getting a little complex, you might want to review Section 1.2 Removing Explicit Dependencies and 1.3 Delegating Object Creation to a Class. This is one level of abstraction above that.

The Solution shows two classes that implement the `IDeploymentPlugin` interface. The `DeploymentManagementBase` class consumes the `IDeploymentPlugin` interface, delegating calls to it's methods to the plug-in classes that implement that interface. Notice how `Validate` calls `Validate` on the `IDeploymentPlugin` instance.

The `Program` has no knowledge of the plug-in classes. It operates on instances of `DeploymentManagementBase`, as demonstrated where `Main` calls `GetPlugins` and receives an array of `DeploymentManagementBase` instances. `Program` doesn't care about the plug-ins. For demo simplicity, `GetPlugins` is a method in Program, but could be another class with a mechanism for selecting which plugins to use. Notice in the `Run` method, how it iterates through `DeploymentManagementBase` instances.

NOTE

Making `DeploymentManagementBase` implement an interface might make IoC more consistent if you're using interfaces everywhere else. That said, an abstract base class can often work for most IoC containers, mocking, and unit testing tools.

To re-cap, the `DeploymentManagementBase` encapsulates all functionality and delegates work to plug-in classes. The code that makes the plug-in are the deployment managers, plug-in interface, and plug-in classes. The consuming code only works with a collection of `DeploymentManagementBase` and is blissfully unaware of the specific plug-in implementations.

Here's where the power comes in. Whenever you, or any 3rd party you allow, wants to extend the system for a new type of deployment, they do this:

1. Create a new `DeploymentPlugin` class that implements your `IDeploymentPlugin` interface.
2. Create a new `DeploymentManagement` class that derives from `DeploymentManagementBase`.
3. Implement the `DeploymentManagement.CreateDeploymentService` method to instantiate and return the new `DeploymentPlugin`.

Finally, the `GetPlugins` method, or some other logic of your choosing, would add that new code to it's collections of plug-ins to operate on.

See Also

1.2 Removing Explicit Dependencies

1.3 Delegating Object Creation to a Class

1.6 1.5 Designing Application Layers

Problem

You're setting up a new application and are unsure of how to structure the project.

Solution

Here’s a data access layer class:

```
public class GreetingRepository
{
    public string GetNewGreeting() => "Welcome!";

    public string GetVisitGreeting() => "Welcome back!";
}
```

Here’s a business logic layer class:

```
public class Greeting
{
    GreetingRepository greetRep = new GreetingRepository();

    public string GetGreeting(bool isNew) =>
        isNew ? greetRep.GetNewGreeting() :
greetRep.GetVisitGreeting();
}
```

These two classes are part of the UI layer:

```
public class SignIn
{
    Greeting greeting = new Greeting();

    public void Greet()
    {
        Console.Write("Is this your first visit? (true/false): ");
        string newResponse = Console.ReadLine();

        bool.TryParse(newResponse, out bool isNew);

        string greetResponse = greeting.GetGreeting(isNew);

        Console.WriteLine($"\n*\n* {greetResponse} \n*\n");
    }
}

public class Menu
{
    public void Show()
    {
        Console.WriteLine(
            "*------*\n" +
            "* Menu *\n" +
            "*------*\n" +
            "\n" +
            "1. ...\n" +
            "2. ...\n" +
            "3. ...\n" +
            "\n" +
            "Choose: ");
    }
}
```

This is the application entry point (part of the UI layer):

```
class Program
{
    SignIn signIn = new SignIn();
    Menu menu = new Menu();

    static void Main()
    {
        new Program().Start();
    }

    void Start()
    {
        signIn.Greet();
        menu.Show();
    }
}
```

Discussion

There are endless ways to set up and plan the structure of new projects, with some approaches better than others. Rather than viewing this discussion as a definitive conclusion, it’s rather meant as a few options with trade-offs that help you think about your own approach.

The anti-pattern here is Ball of Mud (BoM) architecture. BoM is where a developer opens a single project and starts adding all the code at the same layer in the application. While this approach might help knock out a quick prototype, it has severe complications in the long run. Over time most apps need new features and maintenance to fix bugs. What happens is that the code begins to run together and there’s often much duplication - commonly referred to as spaghetti code. Seriously, no-one wants to maintain code like this and you should avoid it.

WARNING

When under time pressure, it's easy to think that creating a quick prototype might be an acceptable use of time. However, resist this urge. The cost of maintenance on a BoM prototype project is high. The time required to work with spaghetti code to add a new feature or fix a bug quickly wipes out any perceived up-front gains of a sloppy prototype. Because of duplication, fixing a bug in one place leaves the same bug in other parts of the application. This means that not only a developer has to do the bug fix multiple times, but the entire lifecycle of QA, deployment, customer discovery, helpdesk service, and management wastes time on what would be multiple unnecessary cycles. The content in this section helps you avoid this anti-pattern.

The primary concept to grasp here is separation of control. You'll often hear this simplified as a layered architecture where you have UI, business logic, and data layers, with each section named for the type of code placed in that layer. This section uses the layered approach with the goal of showing how to achieve separation of concerns and associated benefits.

NOTE

Sometimes the idea of a layered architecture makes people think they must route application communication through the layers or that certain operations are restricted to their layer. This isn't quite true or practical. For example, business logic can be found in different layers, such as rules for validating user input in the UI layer as well as logic for how to process a certain request. Another example of exceptions to communication patterns is when a user needs to select a set of operations on a form - there isn't any business logic involved and the UI layer can request the list of items from the data layer directly. What we want is separation of concerns to enhance the maintainability of the code and any dogmatic/idealistic restrictions that don't make sense runs counter to that goal.

The Solution starts with a data access layer, `GreetingRepository`. This simulates the Repository pattern, which is an abstraction so that calling code doesn't need to think about how to retrieve the data. Ideally, creating a separate data project promises an additional benefit of reusing that data access layer in another project that needs access to the same data. Sometimes you get reuse and other times you don't, though you always get the benefits of reducing duplication and knowing where the data access logic resides.

The business logic layer has a `Greeting` class. Notice how it uses the `isNew` parameter to determine which method of `GreetingRepository` to call. Any time you find yourself needing to write logic for how to handle a user request, consider putting that code in another class that is considered part of the business logic layer. If you already have code like this, refactor it out into a separate object named for the type of logic it needs to handle.

Finally, there's the UI layer, which is comprised of the `SignIn` and `Menu` classes. These classes handle the interaction with the user, yet they delegate any logic to the business logic layer. `Program` might be considered part of the UI layer, though it's only orchestrating interaction/navigation between other UI layer classes and doesn't perform UI operations itself.

There are a couple dimensions to separation of concerns in this code. `GreetingRepository` is only concerned with data and `Greeting` data in particular. For example, if the app needed data to show in a `Menu`, you would need another class called `MenuRepository` that did Create, Read, Update, and Delete (CRUD) operations on `Menu` data. `Greeting` only handles business logic for `Greeting` data. If a `Menu` had its own business logic, you might consider a separate business logic layer class for that, but only if it made sense. As you can see in the UI layer, `SignIn` only handles interaction with the user for signing into the app and `Menu` only handles interaction with the user for displaying and choosing what they want to do. The beauty is that now you or anyone else can easily go into the application and find the code concerning the subject you need to address.

Figures 1.1, 1.2, and 1.3 show how you might structure each layer into a Visual Studio solution. Figure 1.1 is for a very simple app, like a utility that is unlikely to have many features. In this case it's okay to keep the layers in the same project because there isn't a lot of code and anything extra doesn't have tangible benefit.

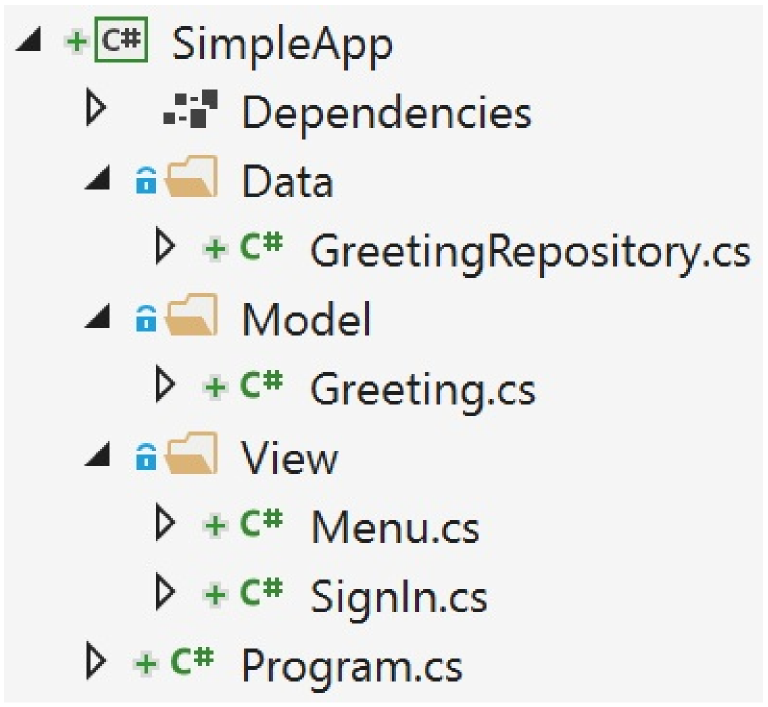
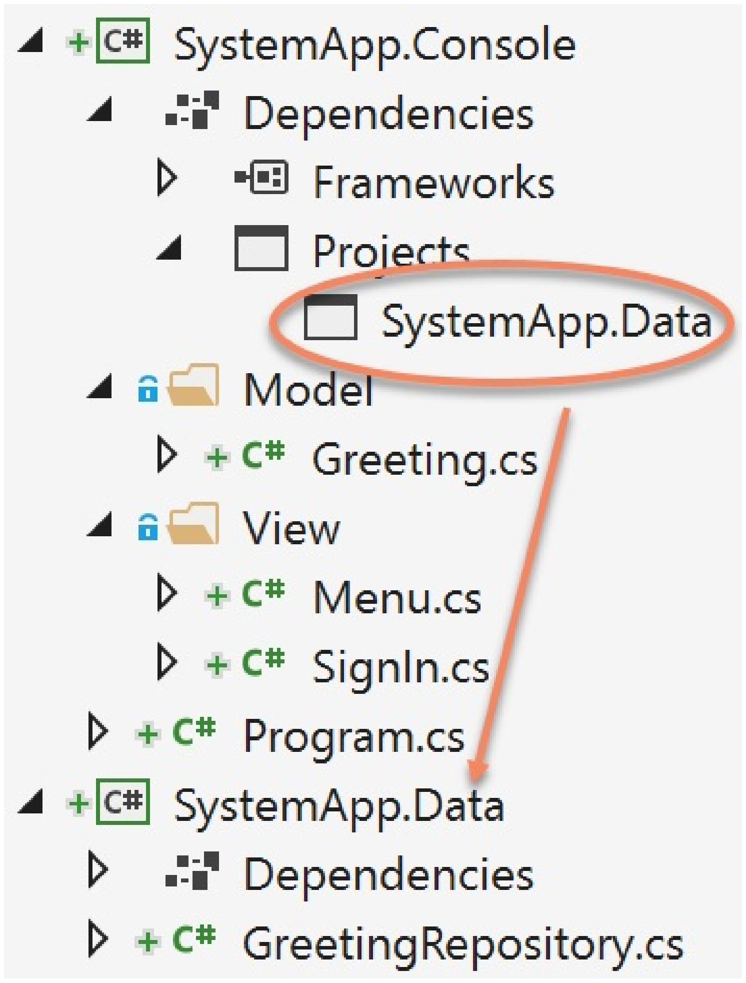
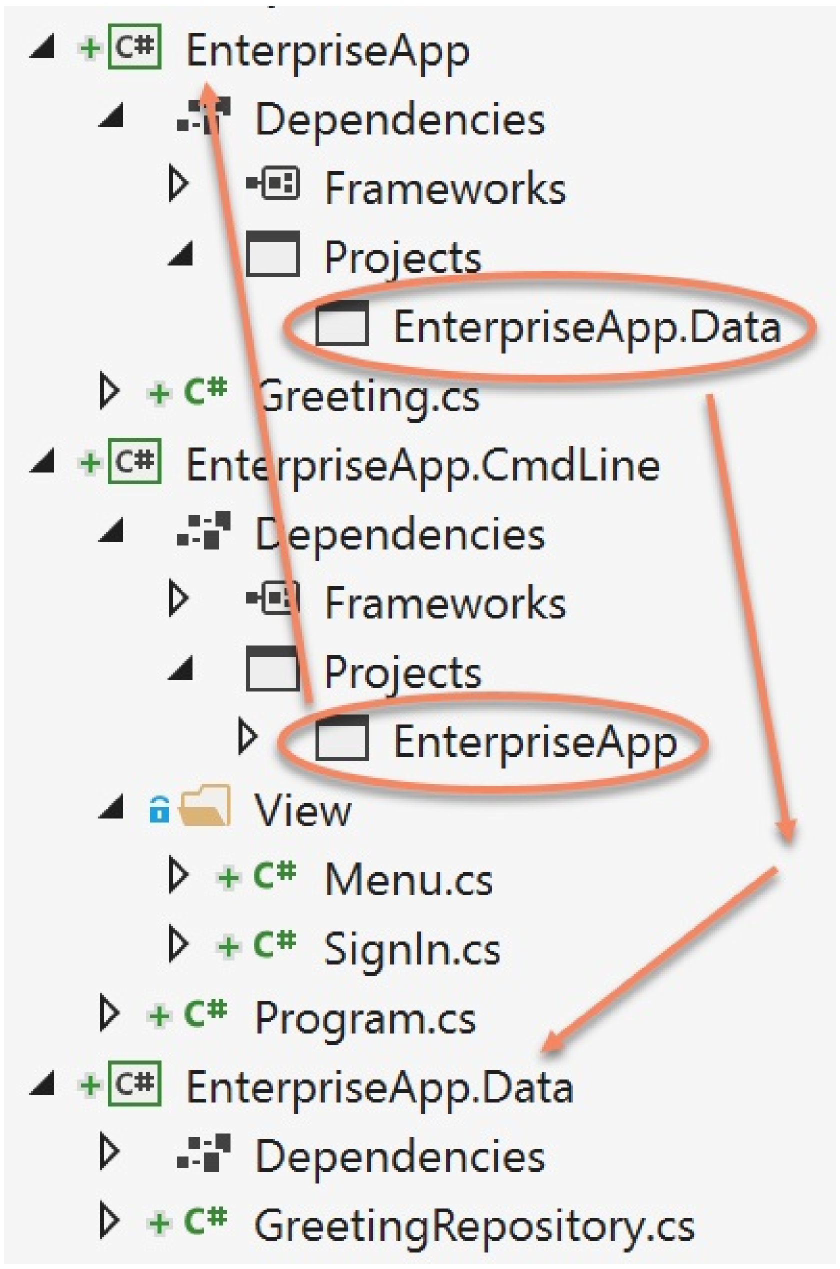


Figure 1.2 shows how you might structure a project that’s a little larger and will grow over time, which I’ll loosely call midsize for the sake of discussion. Notice that it has a separate data access layer. The purpose of that is potential reuse. Some projects offer different UIs for different customers. e.g. There might be a chatbot or mobile app that accesses the data for users but a web app for administrators. Having the data access layer as a separate project makes this possible. Notice how `SystemApp.Console` has an assembly reference to `SystemApp.Data`.



For larger enterprise apps, you’ll want to break the layers apart, as shown in Figure 1.3. The problem to solve here is that you want a cleaner break between sections of code to encourage loose coupling. Large applications often become complex and hard to manage unless you control the architecture in a way that encourages best practices.



For the enterprise scenario, this example is small. However, imagine the complexity of a growing application. As you add new business logic, you'll begin finding code that gets reused. Also, you'll naturally have some code that can stand on its own, like a service layer for accessing an external API. The opportunity here is to have a reusable library that might be useful in other applications. Therefore, you'll want to refactor anything reusable into its own project. On a growing project, you can rarely anticipate every aspect or feature that an app will support and watching for these changes and refactoring will help to keep your code, project, and architecture healthier.

1.7 1.6 Returning Multiple Values from a Method

Problem

You need to return multiple values from a method and using classic approaches, such as `out` parameters or returning a custom type, doesn't feel intuitive.

Solution

`ValidationStatus` has a deconstructor:

```
public class ValidationStatus
{
    public bool Deployment { get; set; }
    public bool SmokeTest { get; set; }
    public bool Artifacts { get; set; }

    public void Deconstruct(
        out bool isPreviousDeploymentComplete,
        out bool isSmokeTestComplete,
        out bool areArtifactsReady)
    {
        isPreviousDeploymentComplete = Deployment;
        isSmokeTestComplete = SmokeTest;
        areArtifactsReady = Artifacts;
    }
}
```

The `DeploymentService` shows how to return a tuple:

```
public class DeploymentService
{
    public
    (bool deployment, bool smokeTest, bool artifacts)
    PrepareDeployment()
    {
        ValidationStatus status = Validate();

        (bool deployment, bool smokeTest, bool artifacts) = status;
        return (deployment, smokeTest, artifacts);
    }

    ValidationStatus Validate()
    {
        return new ValidationStatus
        {
            Deployment = true,
            SmokeTest = true,
            Artifacts = true
        };
    }
}
```

And here's how to consume the returned tuple:

```
class Program
{
    DeploymentService deployment = new DeploymentService();
    static void Main(string[] args)
    {
        new Program().Start();
    }

    void Start()
    {
        (bool deployed, bool smokeTest, bool artifacts) =
            deployment.PrepareDeployment();

        Console.WriteLine(
            $"\nDeployment Status:\n\n" +
            $"Is Previous Deployment Complete? {deployed}\n" +
            $"Is Previous Smoke Test Complete? {smokeTest}\n" +
            $"Are artifacts for this deployment ready?
{artifacts}\n\n" +
            $"Can deploy: {deployed && smokeTest && artifacts}");
    }
}
```

Discussion

Historically, the typical way to return multiple values from a method was to create a custom type or add multiple `out` parameters. It always felt wasteful to create a custom type that would only be used one time for the purpose of returning values. The other option, to use multiple `out` parameters felt clunky too.

Using a tuple is more elegant. A tuple is a value type that lets you group data into a single object without declaring a separate type.

The solution shows a couple different aspects of Tuples, deconstruction and how to return a tuple from a method. The `ValidationStatus` class has a `Deconstruct` method and C# uses that to produce a tuple. This class wasn't strictly necessary for this example, but it does demonstrate an interesting way of converting a class to a tuple.

The `DeploymentService` class shows how to return a tuple. Notice that the return type of the `PrepareDeployment` method is a tuple. The property names in the tuple return type are optional, though meaningful variable names could make the code easier to read for some developers.

The code calls `Validate`, which returns an instance of `ValidationStatus`. The next line, assigning status to the tuple uses the deconstructor to return a tuple instance. `PrepareDeployment` uses those values to return a tuple to the caller.

The `Start` method, in `Program`, shows how to call `PrepareDeployment` and consume the tuple it returns.

1.8 1.7 Converting From Legacy to Strongly Typed Classes

Problem

You have a legacy type that operates on values of type `object` and need to modernize to a strongly typed implementation.

Solution

Here's a `Deployment` class that we'll be using:

```
public class Deployment
{
    string config;

    public Deployment(string config)
    {
        this.config = config;
    }

    public bool PerformHealthCheck()
    {
        Console.WriteLine($"Performed health check for config 
{config}.");
        return true;
    }
}
```

And here's a legacy `CircularQueue` collection:

```
public class CircularQueue
{
    int current = 0;
    int last = 0;
    object[] items;

    public CircularQueue(int size)
    {
        items = new object[size];
    }

    public void Add(object obj)
    {
        if (last >= items.Length)
            throw new IndexOutOfRangeException();

        items[last++] = obj;
    }

    public object Next()
    {
        current %= last;
        object item = items[current];
        current++;

        return item;
    }
}
```

This code shows how to use the legacy collection:

```
public class HealthChecksObjects
{
    public void PerformHealthChecks(int cycles)
    {
        CircularQueue checks = Configure();

        for (int i = 0; i < cycles; i++)
        {
            Deployment deployment = (Deployment)checks.Next();
            deployment.PerformHealthCheck();
        }
    }

    private CircularQueue Configure()
    {
        var queue = new CircularQueue(5);

        queue.Add(new Deployment("a"));
        queue.Add(new Deployment("b"));
        queue.Add(new Deployment("c"));

        return queue;
    }
}
```

Next, here's the legacy collection refactored as a generic collection:

```
    public class CircularQueue<T>
    {
        int current = 0;
        int last = 0;
        T[] items;

        public CircularQueue(int size)
        {
            items = new T[size];
        }

        public void Add(T obj)
        {
            if (last >= items.Length)
                throw new IndexOutOfRangeException();

            items[last++] = obj;
        }

        public T Next()
        {
            current %= last;
            T item = items[current];
            current++;

            return item;
        }
    }
```

With code that shows how to use the new generic collection:

```
    public class HealthChecksGeneric
    {
        public void PerformHealthChecks(int cycles)
        {
            CircularQueue<Deployment> checks = Configure();

            for (int i = 0; i < cycles; i++)
            {
                Deployment deployment = checks.Next();
                deployment.PerformHealthCheck();
            }
        }

        private CircularQueue<Deployment> Configure()
        {
            var queue = new CircularQueue<Deployment>(5);

            queue.Add(new Deployment("a"));
            queue.Add(new Deployment("b"));
            queue.Add(new Deployment("c"));

            return queue;
        }
    }
```

Here’s demo code to show both collections in action:

```
    class Program
    {
        static void Main(string[] args)
        {
            new HealthChecksObjects().PerformHealthChecks(5);
            new HealthChecksGeneric().PerformHealthChecks(5);
        }
    }
```

Discussion

The first version of C# didn’t have Generics. Instead, we had a `System.Collections` namespace with collections like `Dictionary`, `List`, and `Stack` that operated on instances of type `object`. If the instances in the collection were reference types, the conversion performance to/from object was negligible. However, if you wanted to manage a collection of value types, the boxing/unboxing performance penalty became more excruciating the larger the collection got or the more operations performed.

Microsoft had always intended to add Generics and they finally arrived in C# v2.0. However, in the meantime, there was a ton of non-generic code written. Imagine all of the new object-based collections that developers needed to write on their own for things like sets, priority queues, and tree data structures. Add to that types like Delegates, which were the primary means of method reference and async communication that operated on objects. There’s a long list of non-generic code that’s been written and chances are that you’ll encounter some of it as you progress through your career.

As C# developers, we appreciate the benefits of strongly typed code, making it easier for find and fix compile-time errors, making an application more maintainable, and improving quality. For this reason, you might have a strong desire to refactor a given piece of non-generic code so that it too is strongly typed with generics.

The process is basically this - whenever you see type `object`, convert it to generic.

The Solution shows a `Deployment` object that performs a health check on a deployed artifact. Since we have multiple artifacts, we also need to hold multiple `Deployment` instances in a collection. The collection is a circular queue and there’s a `HealthCheck` class that loops through the queue and periodically performs a health check with the next `Deployment` instance.

`HealthCheckObject` operates on old non-generic code and `HealthCheckGeneric` operates on new generic code. The difference between the two are that the `HealthCheckObject Configure` method instantiates a non-generic `CircularQueue` and the `HealthCheckGeneric Configure` method instantiates a generic `CircularQueue<T>`. Our primary task is to convert `CircularQueue` to `CircularQueue<T>`.

Since we’re working with a collection, the first task is to add the type parameter to the class, `CircularQueue<T>`. Then look for anywhere the code uses the `object` type and convert that to the class type parameter, `T`:

1. Convert the `object items[]` field to `T items[]`.
2. In the constructor, instantiate a new `T[]` instead of `object[]`.
3. Change the `Add` parameter from `object` to `T`.
4. Change the `Next` return type from `object` to `T`.
5. In `Next`, change the `object item` variable to `T item`.

After changing `object` types to `T`, you have a new strongly typed generic collection.

The `Program` class demonstrates how both of these collections work.

1.9 1.8 Making Classes Adapt to your Interface

Problem

You have a 3rd party library with similar functionality as your code, but it doesn’t have the same interface.

Solution

This is the interface we want to work with:

```
public interface IDeploymentService
{
    void Validate();
}
```

Here are a couple classes that implement that interface:

```
public class DeploymentService1 : IDeploymentService
{
    public void Validate()
    {
        Console.WriteLine("Deployment Service 1 Validated");
    }
}

public class DeploymentService2 : IDeploymentService
{
    public void Validate()
    {
        Console.WriteLine("Deployment Service 2 Validated");
    }
}
```

Here’s a 3rd party class that doesn’t implement `IDeploymentService`:

```
public class ThirdPartyDeploymentService
{
    public void PerformValidation()
    {
        Console.WriteLine("3rd Party Deployment Service 1 Validated");
    }
}
```

This is the adapter that implements `IDeploymentService`:

```
public class ThirdPartyDeploymentAdapter : IDeploymentService
{
    ThirdPartyDeploymentService service = new
ThirdPartyDeploymentService();

    public void Validate()
    {
        service.PerformValidation();
    }
}
```

This code shows how to include the 3rd party service by using the adapter:

```
class Program
{
    static void Main(string[] args)
    {
        new Program().Start();
    }

    void Start()
    {
        List<IDeploymentService> services = Configure();
```

```
            foreach (var svc in services)
                svc.Validate();
        }

        List<IDeploymentService> Configure()
        {
            return new List<IDeploymentService>
            {
                new DeploymentService1(),
                new DeploymentService2(),
                new ThirdPartyDeploymentAdapter()
            };
        }
    }
```

Discussion

An adapter is a class that wraps another class, but exposes the functionality of the wrapped class with the interface you need.

There are various situations where the need for an adapter class comes into play. What if you have a group of objects that implement an interface and want to use a 3rd party class that doesn't match the interface that your code works with? What if your code is written for a 3rd party API, like a payment service, and you know you want to eventually switch to a different provider with a different API? What if you need to use native code via P/Invoke or COM interop and didn't want the details of that interface to bleed into your code? These are all good candidates for considering an adapter.

The solution code has `DeploymentService` classes that implement `IDeploymentService`. You can see in the `Program` `Start` method that it only operates on instances that implement `IDeploymentService`.

Sometime later, you encounter the need to integrate `ThirdPartyDeploymentService` into the app. However, it doesn't implement `IDeploymentService` and you don't have the code for `ThirdPartyDeploymentService`.

The `ThirdPartyDeploymentServiceAdapter` class solves the problem. It implements `IDeploymentService`, instantiates its own copy of `ThirdPartyDeploymentService`, and the `Validate` method delegates the call to `ThirdPartyDeploymentService`. Notice that the `Program`

`Configure` method adds an instance of `ThirdPartyDeploymentServiceAdapter` to the collection that `Start` operates on.

This was a demo to show you how to design an adapter. In practice, the `PerformValidation` method of `ThirdPartyDeploymentService` likely has different parameters and a different return type. The `ThirdPartyServiceAdapter` `Validate` method will be responsible for preparing arguments and reshaping return values to ensure they conform to the proper `IDeploymentService` interface.

1.10 1.9 Designing a Custom Exception

Problem

The .NET Framework library doesn't have an Exception type that fits your requirements.

Solution

This is a custom exception:

```
[Serializable]
public class DeploymentValidationException : Exception
{
    public DeploymentValidationException() :
        this("Validation Failed!", null,
ValidationFailureReason.Unknown)
    {
    }

    public DeploymentValidationException(
        string message) :
        this(message, null, ValidationFailureReason.Unknown)
    {
    }

    public DeploymentValidationException(
        string message, Exception innerException) :
        this(message, innerException, ValidationFailureReason.Unknown)
    {
    }

    public DeploymentValidationException(
        string message, Exception innerException,
ValidationFailureReason reason) :
        base(message, innerException)
    {
        Reason = reason;
    }

    public ValidationFailureReason Reason { get; set; }

    public override string ToString()
    {
        return
            base.ToString() +
            $" - Reason: {Reason} ";
    }
}
```

And this is an enum type for a property on that exception:

```
public enum ValidationFailureReason
{
    Unknown,
    PreviousDeploymentFailed,
    SmokeTestFailed,
    MissingArtifacts
}
```

This code shows how to throw the custom exception:

```
public class DeploymentService
{
    public void Validate()
    {
        throw new DeploymentValidationException(
            "Smoke test failed - check with qa@example.com.",
            null,
            ValidationFailureReason.SmokeTestFailed);
    }
}
```

And this code catches the custom exception:

```
class Program
```

```
{
    static void Main()
    {
        try
        {
            new DeploymentService().Validate();
        }
        catch (DeploymentValidationException ex)
        {
            Console.WriteLine(
                $"Message: {ex.Message}\n" +
                $"Reason: {ex.Reason}\n" +
                $"Full Description: \n {ex}");
        }
    }
}
```

Discussion

The beautiful thing about C# exceptions are that they're strongly typed. When your code catches them, you can write specific handling logic for just that type of exception. The .NET Framework has a few exceptions, like `ArgumentNullException`, that get some reuse (you can throw yourself) in the average code base, but often you'll need to throw an exception with the semantics and data that gives a developer a fairer chance of figuring out why a method couldn't complete its intended purpose.

The exception in the solution is `DeploymentValidationException`, which indicates a problem related to the deployment process during the validation phase. It derives from `Exception`. Depending on how extensive your custom exception framework is, you could create your own base exception for a hierarchy and classify a derived exception tree from that. The benefit is that you would have flexibility in catch blocks to catch more general or specific exceptions as necessary. That said, if you only need a couple custom exceptions, the extra design work of an exception hierarchy might be overkill.

NOTE

In the past, there's been discussion of whether a custom exception should derive from `Exception` or `ApplicationException`, where `Exception` was for .NET type hierarchies and `ApplicationException` was for custom exception hierarchies. However, the distinction blurred over time with some .NET Framework types deriving from both with no apparent consistency or reason.

So, deriving from `Exception` seems to be fine these days.

The first three constructors mirror the `Exception` class options for message and inner exception. You'll also want custom constructors for instantiating with your custom data.

`DeploymentValidationException` has a property, of the enum type `ValidationFailedReason`. Besides having semantics unique to the reason for throwing an exception, another purpose of a custom exception is to include important information for exception handling and/or debugging.

Overriding `ToString` is also a good idea. Logging frameworks might just receive the `Exception` reference, resulting in a call to `ToString`. As in this example, you'll want to ensure your custom data gets included in the string output. This ensures people can read the full state of the exception, along with the stack trace.

The `Program` `Main` method demonstrates how nice it is to be able to handle the specific type, rather than another type that might not fit or the general `Exception` class.

1.11 1.10 Building a Fluid Interface

Problem

You need to build a new type with complex configuration options without an unnecessary expansion of constructors.

Solution

Here's the `DeploymentService` class we want to build:

```
public class DeploymentService
{
    public int StartDelay { get; set; } = 2000;
    public int ErrorRetries { get; set; } = 5;
    public string ReportFormat { get; set; } = "pdf";
    public void Start()
    {
        Console.WriteLine(
            $"Deployment started with:\n" +
            $"    Start Delay:   {StartDelay}\n" +
            $"    Error Retries: {ErrorRetries}\n" +
            $"    Report Format: {ReportFormat}");
    }
}
```

This is the class that builds the `DeploymentService` instance:

```
public class DeploymentBuilder
{
    DeploymentService service = new DeploymentService();

    public DeploymentBuilder SetStartDelay(int delay)
    {
        service.StartDelay = delay;
        return this;
    }

    public DeploymentBuilder SetErrorRetries(int retries)
    {
        service.ErrorRetries = retries;
        return this;
    }

    public DeploymentBuilder SetReportFormat(string format)
    {
        service.ReportFormat = format;
        return this;
    }

    public DeploymentService Build()
    {
        return service;
    }
}
```

Here’s how to use the `DeploymentBuilder` class:

```
class Program
{
    static void Main()
    {
        DeploymentService service =
            new DeploymentBuilder()
                .SetStartDelay(3000)
                .SetErrorRetries(3)
                .SetReportFormat("html")
                .Build();

        service.Start();
    }
}
```

Discussion

In Section 1.9, the `DeploymentValidationException` class has 4 constructors. This isn't a problem because convention with `Exceptions` means that everyone expects the first three constructors and the pattern of adding a new parameter for a new field in subsequent constructors is normal.

However, what if the class you were designing had a lot of options and there was a strong possibility that new features would require new options. Further, developers will want to pick and choose what options to configure the class with. Imagine the exponential explosion of new constructors for every new option added to the class. In such a scenario, constructors are practically useless. The Builder pattern can solve this problem.

An example of an object that implements the Builder Pattern is the ASP.NET `ConfigSettings`. Another is the `ServiceCollection` from Section 1.2 - the code isn't entirely written in a fluid manner, but it could be because it follows the Builder pattern.

The Solution has a `DeploymentService` class, which is what we want to build. Its properties have default values in case a developer doesn't configure a given value. In general terms, the class that the Builder creates will also have other methods and members for its intended purpose.

The `DeploymentBuilder` class implements the Builder pattern. Notice that all of the methods, except for `Build`, return the same instance (`this`) of the same type, `DeploymentBuilder`. They also use the parameter to configure the `DeploymentService` field that was instantiated with the `DeploymentBuilder` instance. The `Build` method returns the `DeploymentService` instance.

How the configuration and instantiation occur are implementation details of the `DeploymentBuilder` and you can vary them as needed. You can also accept any parameter type you need and perform the configuration. Additionally, you can collect configuration data and only instantiate the target class when the `Build` method runs. You have all the flexibility to design the internals of the builder for what makes sense to you.

Finally, notice how the `Main` method instantiates `DeploymentBuilder` and uses its fluent interface for configuration and finally calls `Build` to return the `DeploymentService` instance. This example used every method, but that wasn’t required because you have the option to use some, none, or all.

See Also

1.2 Removing Explicit Dependencies 1.9 Designing a Custom Exception

Chapter 2. Coding Algorithms

2.1 Overview

We code every day, thinking about the problem we’re solving and ensuring that our algorithms work correctly. This is how it should be and modern tools and SDKs increasingly free our time to do just that. Even so, there are features of C#, .NET, and coding in general that have significant effects on efficiency, performance, and maintainability.

Performance

A few subjects in this chapter discuss application performance, such as the efficient handling of strings, caching data, or delaying the instantiation of a type until you need it. In some simple scenarios, these things might not matter. However, in complex enterprise apps that need the performance and scale, keeping an eye on these techniques can help avoid expensive problems in production.

Maintainability

How you organize code can significantly affect its maintainability. Building on the discussions in Chapter 1, you’ll see a new pattern, Strategy, and how it can help simplify an algorithm and make an app more extensible. Another section discusses using recursion for naturally occurring hierarchical data. Collecting these techniques and thinking about the best way to approach an algorithm can make a significant difference in the maintainability and quality of code.

Mindset

A couple sections of this chapter might be interesting in specific contexts - different ways to think about solving problems. You might not use regular expressions every day, but they're very useful when you need them. Another section, on converting to/from Unix time, looks into the future of .NET as a cross-platform language; knowing that we need a certain mindset to think about designing algorithms in an environment we might not have ever considered in the past.

2.2 2.1 Processing strings Efficiently

Problem

A profiler indicates a problem in part of your code that builds a large string iteratively and you need to improve performance.

Solution

Here's an `InvoiceItem` class we'll be working with:

```
class InvoiceItem
{
    public decimal Cost { get; set; }
    public string Description { get; set; }
}
```

This method produces sample data for the demo:

```
static List<InvoiceItem> GetInvoiceItems()
{
    var items = new List<InvoiceItem>();
    var rand = new Random();
    for (int i = 0; i < 100; i++)
        items.Add(
            new InvoiceItem
            {
                Cost = rand.Next(i),
                Description = "Invoice Item #" + (i+1)
            });
    return items;
}
```

There are two methods for working with strings. First, the inefficient method:

```
    static string DoStringConcatenation(List<InvoiceItem> lineItems)
    {
        string report = "";

        foreach (var item in lineItems)
            report += $"{item.Cost:C} - {item.Description}";

        return report;
    }
```

Next is the more efficient method:

```
    static string DoStringBuilderConcatenation(List<InvoiceItem>
    lineItems)
    {
        var reportBuilder = new StringBuilder();

        foreach (var item in lineItems)
            reportBuilder.Append($"{item.Cost:C} - {item.Description}");

        return reportBuilder.ToString();
    }
```

The `Main` method ties all of this together:

```
    static void Main(string[] args)
    {
        List<InvoiceItem> lineItems = GetInvoiceItems();

        DoStringConcatenation(lineItems);

        DoStringBuilderConcatenation(lineItems);
    }
```

Discussion

There are different reasons why we need to gather data into a longer string. Reports, whether text based or formatted via HTML or other markup, require combining text strings. Sometimes we add items to an email or manually build

PDF content as an email attachment. Other times we might need to export data in a non-standard format for legacy systems. Too often, developers use string concatenation when `StringBuilder` is the superior choice.

String concatenation is intuitive and quick to code, which is why so many people do it. However, concatenating strings can also kill application performance. The problem occurs because each concatenation performs expensive memory allocations. Let’s examine both the wrong way to build strings and the right way.

The logic in the `DoStringConcatenation` method extracts `Cost` and `Description` from each `InvoiceItem` and concatenates that to a growing string. Concatenating just a few strings might go unnoticed. However, imagine if this was 25, 50, or 100 lines or more. Using string concatenation as an example, Section 3.10 shows how string concatenation is an exponentially time intensive operation that destroys application performance.

NOTE

When concatenating within the same expression, e.g. string1 + string2, the C# compiler can optimize the code. It’s the loop with concatenation that causes the huge performance hit.

The `DoStringBuilderConcatenation` method fixes this problem. It uses the `StringBuilder`, which is in the `System.Text` namespace. It uses the Builder pattern, described in section 1.10, where each `AppendText` adds the new string to the `StringBuilder` instance, `reportsBuilder`. Before returning, the method calls `ToString` to convert the `StringBuilder` contents to a string.

TIP

As a rule of thumb, once you’ve gone past 4 string concatenations, you’ll receive better performance by using StringBuilder.

Fortunately, the .NET ecosystem has many .NET Framework libraries and 3rd party libraries that help with forming strings of common format. You should use one of these libraries whenever possible because they're often optimized for performance and will save time and make the code easier to read. To give you an idea, here are a few libraries to consider for common formats:

Data Format | Library

JSON .NET 5 | System.Text.Json JSON ⇐ .NET 4.x | Json.NET XML | LINQ to XML CSV | LINQ to CSV HTML | System.Web.UI.HtmlTextWriter PDF | Various Commercial and Open Source Providers Excel | Various Commercial and Open Source Providers

One more thought - Custom search and filtering panels are common to give users a simple way to query corporate data. Too frequently, developers use string concatenation to build SQL queries. While string concatenation is easier, beyond performance, the problem with that is security. String concatenated SQL statements open the opportunity for SQL Injection attack. In this case, `StringBuilder` isn't a solution. Instead, you should use a data library that parameterizes user input to circumvent SQL injection. There's ADO.NET, LINQ Providers, and other 3rd party data libraries that do input value parameterization for you. For dynamic queries, using a data library might be harder, but it is possible. You might want to seriously consider using LINQ, which I discuss in Chapter 4.

See Also

Section 1.10 Building a Fluid Interface Section 3.10 Measuring Performance Chapter 4 Querying with LINQ

2.3 2.2 Simplifying Instance Cleanup

Problem

Old `using` statements cause unnecessary nesting and you want to clean up and simplify code.

Solution

This program has using statements for reading and writing to a text file:

```
class Program
{
    const string FileName = "Invoice.txt";

    static void Main(string[] args)
    {
        Console.WriteLine(
            "Invoice App\n" +
            "------------\n");

        WriteDetails();

        ReadDetails();
    }

    static void WriteDetails()
    {

        using var writer = new StreamWriter(FileName);

        Console.WriteLine("Type details and press [Enter] to end.\n");

        string detail = string.Empty;
        do
        {
            Console.Write("Detail: ");
            detail = Console.ReadLine();
            writer.WriteLine(detail);
        }
        while (!string.IsNullOrWhiteSpace(detail));
    }

    static void ReadDetails()
    {
        Console.WriteLine("\nInvoice Details:\n");

        using var reader = new StreamReader(FileName);

        string detail = string.Empty;
        do
        {
            detail = reader.ReadLine();
            Console.WriteLine(detail);
        }
        while (!string.IsNullOrWhiteSpace(detail));
    }
}
```

Discussion

Before C# 8, `using` statement syntax required parenthesis for `IDisposable` object instantiation and an enclosing block. During runtime, when the program reached the closing block, it would call `Dispose` on the instantiated object. If you needed multiple `using` statements to operate at the same time, developers would often nest them, resulting in extra space in addition to normal statement nesting. This pattern was enough of an annoyance to some developers that Microsoft added a feature to the language to simplify using statements.

In the solution, you can see a couple places where the new `using` statement syntax occurs: instantiating the `StreamWriter` in `WriteDetails` and instantiating the `StreamReader` in `ReadDetails`. In both cases, the `using` statement is on a single line. Gone are the parenthesis and curly braces and each statement terminates with a semi-colon.

The scope of the new `using` statement is its enclosing block, calling the `using` object's `Dispose` method when execution reaches the end of the enclosing block. In the solution, the enclosing block is the method, which causes each `using` object's `Dispose` method to be called at the end of the method.

What's different about the single line `using` statement is that it will work with both `IDisposable` objects and objects that implement a disposable pattern. In this context, a disposable pattern means that the object doesn't implement `IDisposable`, yet it has a parameterless `Dispose` method.

See Also

Section 1.1 Managing Object End-of-Lifetime

2.4 2.3 Keeping Logic Local

Problem

An algorithm has complex logic that is better refactored to another method, but the logic is really only used in one place.

Solution

The program uses the `CustomerType` and `InvoiceItem`:

```
enum CustomerType
{
    None,
    Bronze,
    Silver,
    Gold
}

class InvoiceItem
{
    public decimal Cost { get; set; }
    public string Description { get; set; }
}
```

This method generates and returns a demo set of invoices:

```
static List<InvoiceItem> GetInvoiceItems()
{
    var items = new List<InvoiceItem>();
    var rand = new Random();
    for (int i = 0; i < 100; i++)
        items.Add(
            new InvoiceItem
            {
                Cost = rand.Next(i),
                Description = "Invoice Item #" + (i + 1)
            });

    return items;
}
```

Finally, the `Main` method shows how to use a local function:

```
static void Main()
{
    List<InvoiceItem> lineItems = GetInvoiceItems();

    decimal total = 0;

    foreach (var item in lineItems)
        total += item.Cost;
    total = ApplyDiscount(total, CustomerType.Gold);

    Console.WriteLine($"Total Invoice Balance: {total:C}");

    decimal ApplyDiscount(decimal total, CustomerType customerType)
    {
        switch (customerType)
        {
            case CustomerType.Bronze:
                return total - total * .10m;
            case CustomerType.Silver:
                return total - total * .05m;
            case CustomerType.Gold:
                return total - total * .02m;
            case CustomerType.None:
            default:
                return total;
        }
    }
}
```

Discussion

Local methods are useful whenever code is only relevant to a single method and you want to separate that code. Reasons for separating code are to give meaning to a set of complex logic, re-use logic and simplify calling code (perhaps a loop), or allow an async method to throw an exception before awaiting the enclosing method.

The `Main` method in the solution has a local method, named `ApplyDiscount`. This example demonstrates how a local method can simplify code. If you examine the code in `ApplyDiscount`, it might not be immediately clear what its purpose is. However, by separating that logic into its own method, anyone can read the method name and know what the purpose of the logic is. This is a great way to make code more maintainable, by expressing intent, and making that logic local where another developer won’t need to hunt for a class method that might move around after future maintenance.

2.5 2.4 Operating on Multiple Classes the Same Way

Problem

An application must be extensible, for adding new plug-in capabilities, but you don’t want to re-write existing code for new classes.

Solution

This is a common interface for several classes to implement:

```
public interface IInvoice
{
    bool IsApproved();

    void PopulateLineItems();

    void CalculateBalance();

    void SetDueDate();
}
```

Here are a few classes that implement `IInvoice`:

```
public class BankInvoice : IInvoice
{
    public void CalculateBalance()
    {
        Console.WriteLine("Calculating balance for BankInvoice.");
    }

    public bool IsApproved()
    {
        Console.WriteLine("Checking approval for BankInvoice.");
        return true;
    }

    public void PopulateLineItems()
    {
        Console.WriteLine("Populating items for BankInvoice.");
    }

    public void SetDueDate()
    {
        Console.WriteLine("Setting due date for BankInvoice.");
    }
}
```

```
public class EnterpriseInvoice : IInvoice
{
    public void CalculateBalance()
    {
        Console.WriteLine("Calculating balance for
EnterpriseInvoice.");
    }

    public bool IsApproved()
    {
        Console.WriteLine("Checking approval for EnterpriseInvoice.");
        return true;
    }

    public void PopulateLineItems()
    {
        Console.WriteLine("Populating items for EnterpriseInvoice.");
    }

    public void SetDueDate()
    {
        Console.WriteLine("Setting due date for EnterpriseInvoice.");
    }
}

public class GovernmentInvoice : IInvoice
{
    public void CalculateBalance()
    {
        Console.WriteLine("Calculating balance for
GovernmentInvoice.");
    }

    public bool IsApproved()
    {
        Console.WriteLine("Checking approval for GovernmentInvoice.");
        return true;
    }

    public void PopulateLineItems()
    {
        Console.WriteLine("Populating items for GovernmentInvoice.");
    }

    public void SetDueDate()
    {
        Console.WriteLine("Setting due date for GovernmentInvoice.");
    }
}
```

This method populates a collection with classes that implement `IInvoice`:

```
static List<IInvoice> GetInvoices()
{
    return new List<IInvoice>
    {
        new BankInvoice(),
        new EnterpriseInvoice(),
        new GovernmentInvoice()
    };
}
```

The `Main` method has an algorithm that operates on the `IInvoice` interface:

```
static void Main(string[] args)
{
    List<IInvoice> invoices = GetInvoices();

    foreach (var invoice in invoices)
    {
        if (invoice.IsApproved())
        {
            invoice.CalculateBalance();
            invoice.PopulateLineItems();
            invoice.SetDueDate();
        }
    }
}
```

Discussion

As a developer's career progresses, chances are they'll encounter requirements that customers want an application to be "extensible". Although the exact meaning is anomalous to even the most seasoned architects, there's a general understanding that "extensibility" should be a theme in the application's design. We generally move in this direction by identifying areas of the application that can and will change over time. Patterns can help with this, such as the factory classes of Section 1.3, factory methods of Section 1.4, and builders in Section 1.10. In a similar light, the Strategy pattern described in this section helps organize code for extensibility.

The Strategy pattern is useful when there are multiple object types to work with at the same time and you want them to be interchangeable and write code one time that operates the same way for each object. The software we use every day are classic examples of where a Strategy could work. Office applications have different document types and allow developers to write their own add-ins. Browsers have add-ins that developers can write. The editors and Integrated Development Environments (IDEs) you use every day have plug-in capabilities.

The solution describes an application that operates on different types of invoices in the domains of Banking, Enterprise, and Government. Each of these domains have their own business rules related to legal or other requirements. What makes this extensible is the fact that, in the future, we can add another object to handle invoices in another domain.

The glue to making this work is the `IInvoice` interface. It contains the required methods (or contract) that each implementing object must define. You can see that the `BankInvoice`, `EnterpriseInvoice`, and `GovernmentInvoices` each implement `IInvoice`.

`GetInvoices` simulates the situation where you would write code to populate invoices from a data source. Whenever you need to extend the framework, by adding a new `IInvoice` derived type, this is the only code that changes. Because all classes are `IInvoice`, they can all be returned via the same `List<IInvoice>` collection.

Finally, examine the `Main` method. It iterates on each `IInvoice` object, calling each method. `Main` doesn’t care what the specific implementation is and so its code never needs to change to accommodate instance specific logic. You don’t need `if` or `switch` statements for special cases, which blows up into spaghetti code in maintenance. Any future changes will be on how `Main` works with the `IInvoice` interface. Any changes to business logic associated with invoices is limited to the invoice types themselves. This is easy to maintain and easy to figure out where logic is and should be. Further, it’s also easy to extend by adding a new Plug-In class that implements `IInvoice`.

See Also

1.3 Delegating Object Creation to a Class 1.4 Delegating Object Creation to a Method 1.10 Building a Fluid Interface

2.6 2.5 Checking for Type Equality

Problem

You need to search for objects in a collection and default equality won’t work.

Solution

The `Invoice` class implements `IEquatable<T>`:

```
public class Invoice : IEquatable<Invoice>
{
    public int CustomerID { get; set; }

    public DateTime Created { get; set; }

    public List<string> InvoiceItems { get; set; }

    public decimal Total { get; set; }

    public bool Equals(Invoice other)
    {
        if (ReferenceEquals(other, null))
            return false;

        if (ReferenceEquals(this, other))
            return true;

        if (GetType() != other.GetType())
            return false;

        return
            CustomerID == other.CustomerID &&
            Created.Date == other.Created.Date;
    }

    public override bool Equals(object other)
    {
        return Equals(other as Invoice);
    }

    public override int GetHashCode()
    {
        return (CustomerID + Created.Ticks).GetHashCode();
    }
```

```
        public static bool operator ==(Invoice left, Invoice right)
        {
            if (ReferenceEquals(left, null))
                return ReferenceEquals(right, null);

            return left.Equals(right);
        }

        public static bool operator !=(Invoice left, Invoice right)
        {
            return !(left == right);
        }
    }
```

This code returns a collection of `Invoice` classes:

```
    private static List<Invoice> GetAllInvoices()
    {
        return new List<Invoice>
        {
            new Invoice { CustomerID = 1, Created = DateTime.Now },
            new Invoice { CustomerID = 2, Created = DateTime.Now },
            new Invoice { CustomerID = 1, Created = DateTime.Now },
            new Invoice { CustomerID = 3, Created = DateTime.Now }
        };
    }
```

Here’s how to use the `Invoice` class:

```
    static void Main(string[] args)
    {
        List<Invoice> allInvoices = GetAllInvoices();

        Console.WriteLine($"# of All Invoices: {allInvoices.Count}");

        var invoicesToProcess = new List<Invoice>();

        foreach (var invoice in allInvoices)
        {
            if (!invoicesToProcess.Contains(invoice))
                invoicesToProcess.Add(invoice);
        }

        Console.WriteLine($"# of Invoices to Process:
    {invoicesToProcess.Count}");
    }
```

Discussion

The default equality semantics for reference types is reference equality and for value types is value equality. Reference equality means that when comparing objects, do their references refer to the same exact object instance. Value equality occurs when each member of an object is compared before two objects are considered equal. The problem with reference equality is that sometimes you have two copies of an object, referring to different object instances, but you really want to check their values to see if they are equal. Value equality might also pose a problem because you might only want to check part of the object to see if they're equal.

To solve the problem of inadequate default equality, the solution implements custom equality on `Invoice`. The `Invoice` class implements the `IEquatable<T>` interface, where `T` is `Invoice`. Although `IEquatable<T>` requires an `Equals(T other)` method, you should also implement `Equals(object other)`, `GetHashCode()`, and the `==` and `!=` operators, resulting in a consistent definition of equals for all scenarios.

There's a lot of science in picking a good hash code, which is out of scope for this book, so the solution implementation is minimal.

The equality implementation avoids repeating code. The `!=` operator invokes (and negates) the `==` operator. The `==` operator checks references and returns `true` if both references are `null` and `false` if only one reference is `null`. Both the `==` operator and the `Equals(object other)` method call the `Equals(Invoice other)` method.

The current instance is clearly not `null`, so `Equals(Invoice other)` only checks the `other` reference and returns `false` if it's `null`. Then it checks to see if `this` and `other` have reference equality, which would obviously mean they are equal. Then if the objects aren't the same type, they are not considered equal. Finally, return the results of the values to compare. In this example, the only thing that makes sense is the `CustomerID` and `Date`.

NOTE

One of the places you might change in the `Equals(Invoice other)` method is the type check. You could have a different opinion, based on the requirements of your application. e.g. What if you wanted to check equality even if `other` was a derived type? Then change the logic to accept derived types also.

The `Main` method processes invoices, ensuring we don't add duplicate invoices to a list. In the loop, it calls the collection `Contains` method, which checks the object's equality. If it doesn't find a matching object, it adds it to the `invoicesToProcess` list. When running the program, there are 4 invoices that exist in `allInvoices`, but only 3 are added to `invoicesToProcess` because there's one duplicate (based on `CustomerID` and `Date`) in `allInvoices`.

NOTE

C# 9.0 Records give you `IEquatable<T>` logic by default. However, Records give you value equality and you would want to implement `IEquatable<T>` yourself if you needed to be more specific. e.g. if your object has free-form text fields that don't contribute to the identity of the object, why waste resources doing the unnecessary field comparisons? Another problem (maybe more rare) could be that some parts of a record might be different for temporal reasons, e.g. temporary timestamps, status, or Globally Unique Identifiers (GUIDs), that will cause the objects to never be equal during processing.

2.7 2.6 Processing Data Hierarchies

Problem

An app needs to work with hierarchical data and an iterative approach is too complex and unnatural.

Solution

This is the format of data we're starting with:

```
class BillingCategory
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int? Parent { get; set; }
}
```

This method returns a collection of hierarchically related records:

```
static List<BillingCategory> GetBillingCategories()
{
    return new List<BillingCategory>
    {
        new BillingCategory { ID = 1, Name = "First 1",  Parent = null
},
        new BillingCategory { ID = 2, Name = "First 2",  Parent = null
},
        new BillingCategory { ID = 4, Name = "Second 1", Parent = 1 },
        new BillingCategory { ID = 3, Name = "First 3",  Parent = null
},
        new BillingCategory { ID = 5, Name = "Second 2", Parent = 2 },
        new BillingCategory { ID = 6, Name = "Second 3", Parent = 3 },
        new BillingCategory { ID = 8, Name = "Third 1",  Parent = 5 },
        new BillingCategory { ID = 8, Name = "Third 2",  Parent = 6 },
        new BillingCategory { ID = 7, Name = "Second 4", Parent = 3 },
        new BillingCategory { ID = 9, Name = "Second 5", Parent = 1 },
        new BillingCategory { ID = 8, Name = "Third 3",  Parent = 9 }
    };
}
```

This is a recursive algorithm that transforms the flat data into a hierarchical form:

```
static List<BillingCategory> BuildHierarchy(
     List<BillingCategory> categories, int? catID, int level)
{
    var found = new List<BillingCategory>();

    foreach (var cat in categories)
    {
        if (cat.Parent == catID)
        {
            cat.Name = new string('\t', level) + cat.Name;
            found.Add(cat);
            List<BillingCategory> subCategories =
                BuildHierarchy(categories, cat.ID, level + 1);
            found.AddRange(subCategories);
        }
    }

    return found;
}
```

The `Main` method runs the program and prints out the hierarchical data:

```
static void Main(string[] args)
{
    List<BillingCategory> categories = GetBillingCategories();

    List<BillingCategory> hierarchy =
        BuildHierarchy(categories, catID: null, level: 0);

    PrintHierarchy(hierarchy);
}

static void PrintHierarchy(List<BillingCategory> hierarchy)
{
    foreach (var cat in hierarchy)
        Console.WriteLine(cat.Name);
}
```

Discussion

It’s hard to tell how many times you have or will encounter iterative algorithms with complex logic and conditions on how the loop operates. Loops like `for`, `foreach`, and `while` are familiar and often used when more elegant solutions are available. I’m not suggesting there’s anything wrong with loops, which are integral parts of our language toolset. However, it’s useful to expand our minds to other techniques that might lend themselves to more elegant and maintainable code for given situations. Sometimes a declarative approach, like a simple lambda on a collection’s `ForEach` operator is simple and clear. LINQ is a nice solution for working with object collections in memory, which is the subject of Chapter 4. Another alternative is recursion - the subject of this section.

The main point I’m making here is that we need to write algorithms using the techniques that are most natural for a given situation. A lot of algorithms do use loops naturally, like iterating through a collection. Other tasks beckon for recursion. A class of algorithms that work on hierarchies might be excellent candidates for recursion.

The solution demonstrates one of the areas where recursion simplified processing and makes the code clear. It processes a list of categories based on billing. Notice that the `BillingCategory` class has both an `ID` and a `Parent`. These manage the hierarchy, where the `Parent` identifies the parent category. Any `BillingCategory` with a `null` `Parent` is a top level category. This is a single table relational DB representation of hierarchical data.

The `GetBillingCategories` represents how the `BillingCategories` arrive from a DB. It's a flat structure. Notice how the `Parent` properties reference the `BillingCategory` IDs that are their parents. Another important fact about the data is that there isn't a clean ordering between parents and children. In a real application, you'll start off with a given set of categories and add new categories later. Again, maintenance in code and data over time changes how we approach algorithm design and this would complicate an iterative solution.

The purpose of this solution is to take the flat category representation and transform it into another list that represents the hierarchical relationship between categories. This was a simple solution, but you might imagine an object based representation where parent categories contained a collection with child categories. The `BuildHierarchy` method is the recursive algorithm that does this.

The `BuildHierarchy` method accepts 3 parameters: `categories`, `catID`, and `level`. The `categories` parameter is the flat collection from the DB and every recursive call receives a reference to this same collection. A potential optimization might be to remove categories that have already been processed, though the demo avoids anything distracting from presented concepts. The `catID` parameter is the `ID` for the current `BillingCategory` and the code is searching for all sub-categories whose `Parent` matches `catID` - as demonstrated by the `if` statement inside the `foreach` loop. The `level` parameter helps manage the visual representation of each category. The first statement inside the `if` block uses `level` to determine how many tabs (`\t`) to prefix to the category name. Every time we make a recursive call to `BuildHierarchy`, we increment level so that subcategories are indented more than their parents.

The algorithm calls `BuildHierarchy` with the same categories collection. Also, it uses the `ID` of the current category, not the `catID` parameter. This means that it recursively calls `BuildHierarchy` until it reaches the bottom most categories. It will know it's at the bottom of the hierarchy because the `foreach` loop completes with no new categories because there aren't any subcategories for the current (bottom) category.

After reaching the bottom, `BuildHierarchy` returns and continues the `foreach` loop, collecting all of the categories under the `catID` - that is, their `Parent` is `catID`. Then it appends any matching sub-categories to the `found` collection to the calling `BuildHierchy`. This continues until the algorithm reaches the top level and all root categories are processed.

NOTE

The recursive algorithm in this solution is referred to as Depth First Search.

Having arrived at the top level, `BuildHierarchy` returns the entire collection to its original caller, which is `Main`. `Main` originally called `BuildHierarchy` with the entire flat `categories` collection. It set `catID` to `null`, indicating that `BuildHierarchy` should start at the root level. The `level` argument is `0`, indicating that we don't want any tab prefixes on root level category names. Here's the output:

First 1 Second 1 Second 5 Third 3 First 2 Second 2 Third 1 First 3 Second 3 Third 2 Second 4

Looking back at the `GetBillingCategories` method, you can see how the visual representation matches the data.

2.8 2.7 Converting From/To Unix Time

Problem

A service is sending date information in seconds or ticks since the Linux epoc that needs to be converted to a C#/.NET DateTime.

Solution

Here are some values we'll be using:

```
static readonly DateTime LinuxEpoch =   new DateTime(1970, 1, 1, 0, 0,
0, 0);
static readonly DateTime WindowsEpoch = new DateTime(0001, 1, 1, 0, 0,
0, 0);
static readonly double EpochMillisecondDifference =
    new TimeSpan(LinuxEpoch.Ticks -
WindowsEpoch.Ticks).TotalMilliseconds;
```

These methods convert from and to Linux epoch timestamps:

```
public static string ToLinuxTimestampFromDateTime(DateTime date)
{
    double dotnetMilliseconds =
TimeSpan.FromTicks(date.Ticks).TotalMilliseconds;

    double linuxMilliseconds = dotnetMilliseconds -
EpochMillisecondDifference;

    double timestamp = Math.Round(
        linuxMilliseconds, 0, MidpointRounding.AwayFromZero);

    return timestamp.ToString();
}

public static DateTime ToDateTimeFromLinuxTimestamp(string timestamp)
{
    ulong.TryParse(timestamp, out ulong epochMilliseconds);
    return LinuxEpoch + +TimeSpan.FromMilliseconds(epochMilliseconds);
}
```

The `Main` method demonstrates how to use those methods:

```
static void Main()
{
    Console.WriteLine(
        $"WindowsEpoch == DateTime.MinValue: " +
        $"{WindowsEpoch == DateTime.MinValue}");

    DateTime testDate = new DateTime(2021, 01, 01);

    Console.WriteLine($"testDate: {testDate}");

    string linuxTimestamp = ToLinuxTimestampFromDateTime(testDate);

    TimeSpan dotnetTimeSpan =
TimeSpan.FromMilliseconds(long.Parse(linuxTimestamp));
    DateTime problemDate = new DateTime(dotnetTimeSpan.Ticks);
```

```
        Console.WriteLine($"Accidentally based on .NET Epoch:
    {problemDate}");

        DateTime goodDate = ToDateTimeFromLinuxTimestamp(linuxTimestamp);

        Console.WriteLine($"Properly based on Linux Epoch: {goodDate}");
    }
```

Discussion

Sometimes developers represent date/time data as milliseconds or ticks in a database. Ticks are measured as 100 nanoseconds. Both milliseconds and Ticks represent time starting at a pre-defined epoch, which is some point in time that is the minimum date for a computing platform. For .NET, the epoch is 01/01/0001 00:00:00, corresponding to the `WindowsEpoch` field in the solution. This is the same as `DateTime.MinValue`, but defining this way makes the example more explicit. For MacOS, the epoch is 1 January 1904 and for Linux, the epoch is 1 January 1970, as shown by the `LinuxEpoch` field in the solution.

NOTE

There are various opinions on whether representing `DateTime` values as milliseconds or ticks is a proper design. However, I leave that debate to other people and venues. My habit is to use the `DateTime` format of the database I’m using. I also translate the `DateTime` to UTC because many apps need to exist beyond the local time zone and you need a consistent translatable representation.

Increasingly, developers are more likely to encounter situations where they need to build cross-platform solutions or integrate with a 3rd party system with milliseconds or ticks based on a different epoch. e.g. The Twitter API began using milliseconds based on the Linux epoch in their 2020 version 2.0 release. The solution example is inspired by code that works with milliseconds from Twitter API responses. The release of .NET Core gave us cross-platform capabilities for C# developers for Console and ASP.NET MVC Core applications. .NET 5 continues the cross-platform story and the roadmap for .NET 6 includes the first rich GUI interface, codenamed Maui. If you’ve been accustomed to working solely in the Microsoft and .NET platforms, this should indicate that things continue to change along the type of thinking required for future development.

The `ToLinuxTimestampFromDateTime` takes a .NET `DateTime` and converts it to a Linux timestamp. The Linux timestamp is the number of milliseconds from the Linux epoch. Since we’re working in milliseconds, the `TimeSpan` converts the `DateTime` ticks to milliseconds. To perform the conversion, we subtract the number of milliseconds between the .NET time and the equivalent Linux time, which we pre-calculated in `EpochMillisecondDifference` by subtracting the .NET (Windows) epoch from the Linux epoch. After the conversion, we need to round the value to eliminate excess precision. The default to `Math.Round` uses what’s called Bankers rounding, which is often not what we need, so the overload with `MidpointRounding.AwayFromZero` does the rounding we expect. The solution returns the final value as a string and you can change that for what makes sense for your implementation.

The `ToDateTimeFromLinuxTimestamp` method is remarkably simpler. After converting to a `ulong`, it creates a new timestamp from the milliseconds and adds that to the LinuxEpoch. Here’s the output from the `Main` method:

WindowsEpoch == DateTime.MinValue: True testDate: 1/1/2021 12:00:00 AM Accidentally based on .NET Epoch: 1/2/0052 12:00:00 AM Properly based on Linux Epoch: 1/1/2021 12:00:00 AM

As you can see, `DateTime.MinValue` is the same as the Windows epoch. Using 1/1/2021 as a good date (at least we hope so), `Main` starts by properly converting that date to a Linux timestamp. Then it shows the wrong way to process that date. Finally, it calls `ToDateTimeFromLinuxTimestamp` performing the proper translation.

2.9 2.8 Caching Frequently Requested Data

Problem

Network latency is causing an app to run slowly because static frequently used data is being fetched too often.

Solution

Here’s the type of data that will be cached:

```
public class InvoiceCategory
{
    public int ID { get; set; }

    public string Name { get; set; }
}
```

This is the interface for the repository that retrieves the data:

```
public interface IInvoiceRepository
{
    List<InvoiceCategory> GetInvoiceCategories();
}
```

This is the repository the retrieves and caches the data:

```
public class InvoiceRepository : IInvoiceRepository
{
    static List<InvoiceCategory> invoiceCategories;

    public List<InvoiceCategory> GetInvoiceCategories()
    {
        if (invoiceCategories == null)
            invoiceCategories = GetInvoiceCategoriesFromDB();

        return invoiceCategories;
    }

    List<InvoiceCategory> GetInvoiceCategoriesFromDB()
    {
        return new List<InvoiceCategory>
        {
            new InvoiceCategory { ID = 1, Name = "Government" },
            new InvoiceCategory { ID = 2, Name = "Financial" },
            new InvoiceCategory { ID = 3, Name = "Enterprise" },
        };
    }
}
```

Here’s the program that uses that repository:

```
class Program
{
    readonly IInvoiceRepository invoiceRep;

    public Program(IInvoiceRepository invoiceRep)
    {
        this.invoiceRep = invoiceRep;
    }

    static void Main()
    {
        new Program(new InvoiceRepository()).Run();
    }

    void Run()
    {
        List<InvoiceCategory> categories =
invoiceRep.GetInvoiceCategories();

        foreach (var category in categories)
            Console.WriteLine($"ID: {category.ID}, Name:
{category.Name}");
    }
}
```

Discussion

Depending on the technology you're using, there could be plenty of options for caching data through mechanisms like CDN, HTTP, and data source solutions. Each has a place and purpose and this section doesn't try to cover all of those options. Rather, it just has a quick and simple technique for caching data that might be helpful.

You might have experienced a scenario where there's a set of data used in a lot of different places. The nature of the data is typically lookup lists or business rule data. In the course of every day work, we build queries that includes this data either in direct select queries or in the form of database table joins. We forget about it until someone starts complaining about application performance. Analysis might reveal that there are a lot of queries that request that same data over and over again. If it's practical, you can cache that data in memory to avoid network latency exacerbated by excessive queries to the same set of data.

This isn't a blanket solution because you have to think about whether it's practical in your situation. e.g. it's impractical to hold too much data in memory, which will cause other scalability problems. Ideally, it’s a finite and relatively small set of data, like invoice categories. That data shouldn’t change too often because if you need real-time access to dynamic data, this won’t work. e.g. If the underlying data source changes, the cache is likely to be holding the old stale data.

The solution shows an `InvoiceCategory` class that we’re going to cache. It’s for a lookup list, just two values per object, a finite and relatively small set of values, and something that doesn’t change much. You can imagine that every query for invoices would require this data as well as admin or search screens with lookup lists. It might speed up invoice queries by removing the extra join and returning less data over the wire where you can join the cached data after the DB query.

The solution has an `InventoryRepository` that implements the `IInvoiceRepository` interface. This wasn’t strictly necessary for this example, though it does support demonstrating another example of IoC, as discussed in Section 1.2.

The `InvoiceRepository` class has a `invoiceCategories` field for holding a collection of `InvoiceCategory`. The `GetInvoiceCategories` method would normally make a DB query and return the results. However, this example only does the DB query if `invoiceCategories` is `null` and caches the result in `invoiceCategories`. This way, subsequent requests get the cached version and doesn’t require a DB query.

NOTE

The `invoiceCategories` field is static because you only want a single cache. In stateless web scenarios, as in ASP.NET, the IIS process recycles unpredictably and developers are advised not to rely on static variables. This situation is different because if the recycle clears out `invoiceCategories`, leaving it `null`, the next query will re-populate it.

The `Main` method uses IoC to instantiate `InvoiceRepository` and performs a query for the `InvoiceCategory` collection.

See Also

1.2 Removing Explicit Dependencies

2.10 2.9 Delaying Type Instantiation

Problem

A class has heavy instantiation requirements and you can save on resource usage by delaying the instantiation to only when necessary.

Solution

Here's the data we'll work with:

```
public class InvoiceCategory
{
    public int ID { get; set; }

    public string Name { get; set; }
}
```

This is the repository interface:

```
public interface IInvoiceRepository
{
    void AddInvoiceCategory(string category);
}
```

This is the repository that we delay instantiation of:

```
public class InvoiceRepository : IInvoiceRepository
{
    public InvoiceRepository()
    {
        Console.WriteLine("InvoiceRepository Instantiated.");
    }

    public void AddInvoiceCategory(string category)
    {
        Console.WriteLine($"for category: {category}");
    }
}
```

This program shows a few ways to perform lazy initialization of the repository:

```
class Program
{
    public static ServiceProvider Container;

    readonly Lazy<InvoiceRepository> InvoiceRep =
        new Lazy<InvoiceRepository>();

    readonly Lazy<IInvoiceRepository> InvoiceRepFactory =
        new Lazy<IInvoiceRepository>(CreateInvoiceRepositoryInstance);

    readonly Lazy<IInvoiceRepository> InvoiceRepIoC =
        new Lazy<IInvoiceRepository>(CreateInvoiceRepositoryFromIoC);

    static IInvoiceRepository CreateInvoiceRepositoryInstance()
    {
        return new InvoiceRepository();
    }

    static IInvoiceRepository CreateInvoiceRepositoryFromIoC()
    {
        return
Program.Container.GetRequiredService<IInvoiceRepository>();
    }

    static void Main()
    {
        Container =
            new ServiceCollection()
                .AddTransient<IInvoiceRepository, InvoiceRepository>()
                .BuildServiceProvider();

        new Program().Run();
    }

    void Run()
    {
        IInvoiceRepository viaLazyDefault = InvoiceRep.Value;
        viaLazyDefault.AddInvoiceCategory("Via Lazy Default \n");

        IInvoiceRepository viaLazyFactory = InvoiceRepFactory.Value;
        viaLazyFactory.AddInvoiceCategory("Via Lazy Factory \n");

        IInvoiceRepository viaLazyIoC = InvoiceRepIoC.Value;
        viaLazyIoC.AddInvoiceCategory("Via Lazy IoC \n");
        }
    }
```

Discussion

Sometimes you have objects with heavy startup overhead. They might need some initial calculation or have to wait on data that takes a while because of network latency or dependencies on poorly performing external systems. This can have serious negative consequences, especially on application startup. Imagine an app that is losing potential customers during trial because it starts too slow or even enterprise users whose work is impacted by wait times. Although you may or may not be able to fix the root cause of the performance bottleneck, another option might be to delay instantiation of that object until you need it. e.g. What if you really don't need that object immediately and can show a start screen right away?

The solution demonstrates how to use `Lazy<T>` to delay object instantiation. The object in question is the `InvoiceRepository` and we're assuming it has a problem in its constructor logic that causes a delay in instantiation.

`Program` has 3 fields whose type is `Lazy<InvoiceRepository>`, showing 3 different ways to instantiate. The first field, `InvoiceRep` instantiates a `Lazy<InvoiceRepository>` with no parameters. It assumes that `InvoiceRepository` has a default constructor (parameterless) and will create a new instance when called.

The `InvoiceRepFactory` property instance references the `CreateInvoiceRepositoryInstance` method. When code accesses this property, it calls the `CreateInvoiceRepositoryInstance` to construct the object. Since it's a method, you have a lot of flexibility in building the object.

In addition to the other two options, the `InvoiceRepIoC` property shows how you can use Lazy instantiation with IoC. Notice that the `Main` method builds an IoC container, as described in Section 1.2. The `CreateInvoiceRepositoryFromIoC` method uses that IoC container to request an instance of `InvoiceRepository`.

Finally, the `Run` method shows how to access the fields, through the `Lazy<T>.Value` property.

See Also

1.2 Removing Explicit Dependencies

2.11 2.10 Parsing Data Files

Problem

The application needs to extract data from a custom external format and string type operations lead to complex and less efficient code.

Solution

Here’s the data types we’ll be working with:

```
class InvoiceItem
{
    public decimal Cost { get; set; }
    public string Description { get; set; }
}

class Invoice
{
    public string Customer { get; set; }
    public DateTime Created { get; set; }
    public List<InvoiceItem> Items { get; set; }
    public decimal Total { get; set; }
}
```

This method returns the raw string data that we want to extract and convert to invoices:

```
static string GetInvoiceTransferFile()
{
    return
        "Creator 1::8/05/20::Item 1\t35.05\tItem 2\t25.18\tItem
3\t13.13::Customer 1::[NOTE] 1\n" +
        "Creator 2::8/10/20::Item 1\t45.05::Customer 2::[NOTE] 2\n" +
        "Creator 1::8/15/20::Item 1\t55.05\tItem 2\t65.18::Customer
3::[NOTE] 3\n";
}

These are utility methods for building and saving invoices:
```

```
static Invoice GetInvoice(string matchCustomer, string matchCreated,
string matchItems)
{
    List<InvoiceItem> lineItems = GetLineItems(matchItems);

    DateTime.TryParse(matchCreated, out DateTime created);

    var invoice =
        new Invoice
        {
            Customer = matchCustomer,
            Created = created,
            Items = lineItems
        };
    return invoice;
}

static List<InvoiceItem> GetLineItems(string matchItems)
{
    var lineItems = new List<InvoiceItem>();

    string[] itemStrings = matchItems.Split('\t');

    for (int i = 0; i < itemStrings.Length; i += 2)
    {
        decimal.TryParse(itemStrings[i + 1], out decimal cost);
        lineItems.Add(
            new InvoiceItem
            {
                Description = itemStrings[i],
                Cost = cost
            });
    }

    return lineItems;
}

static void SaveInvoices(List<Invoice> invoices)
{
    Console.WriteLine($"{invoices.Count} invoices saved.");
}
```

This method uses regular expressions to extract values from raw string data:

```
static List<Invoice> ParseInvoices(string invoiceFile)
{
    var invoices = new List<Invoice>();
    Regex invoiceRegEx = new Regex(
        @"^.+?::(?<created>.+?)::(?<items>.+?)::(?
<customer>.+?)::.+");

    foreach (var invoiceString in invoiceFile.Split('\n'))
    {
        Match match = invoiceRegEx.Match(invoiceString);

        if (match.Success)
        {
            string matchCustomer = match.Groups["customer"].Value;
            string matchCreated = match.Groups["created"].Value;
            string matchItems = match.Groups["items"].Value;

            Invoice invoice = GetInvoice(matchCustomer, matchCreated,
matchItems);
            invoices.Add(invoice);
        }
    }

    return invoices;
}
```

The `Main` method runs the demo:

```
static void Main(string[] args)
{
    string invoiceFile = GetInvoiceTransferFile();

    List<Invoice> invoices = ParseInvoices(invoiceFile);

    SaveInvoices(invoices);
}
```

Discussion

Sometimes, we’ll encounter textual data that doesn’t fit standard data formats. It might come from existing document files, log files, or external and legacy systems. Often, we need to ingest that data and process it for storage in a DB. This section explains how to do that with regular expressions.

The solution shows the data format we want to generate is an `Invoice` with a collection of `InvoiceItem`. The `GetInvoiceTransferFile` method shows the format of the data. The demo suggests that the data might come from a legacy system that already produced that format and it’s easier to write C# code to ingest that than to add code in that system for a better supported format. The specific data we’re interested in extracting are the `created` date, invoice `items`, and `customer` name. Notice that newlines (`\n`) separate records, double colons (`::`) separate invoice fields, and tabs (`\t`) separate invoice item fields.

The `GetInvoice` and `GetLineItems` methods construct the objects from extracted data and serve to separate object construction from the regular expression extraction logic.

The `ParseInvoices` method uses regular expressions to extract values from the input string. The `RegEx` constructor parameter contains the regular expression string, used to extract values. While an entire discussion of regular expressions is out of scope, here’s what this string does:

- `^` says to start at the beginning of the string
- `.`?::+ matches all characters, up to the next invoice field separator (`::`). That said, it ignores the contents that were matched.
- `(?<created>.`?)::+, `(?<items>.`?)::+, and `(?<customer>.`?)::+ are similar to `.`?)::+, but go a step further by extracting values into groups based on the given name. e.g. `(?<created>.`?)::+ means that it will extract all matched data and put the data in a group named “created”.
- `.`+ matches all remaining characters

The `foreach` loop relies on the `\n` separator in the string to work with each invoice. The `Match` method executes the regular expression match, extracting values. If the match was successful, the code extracts values from groups, calls `GetInvoice` and adds the new invoice to the `invoices` collection.

You might have noticed how we’re using `GetLineItems` to extract data from the `matchItems` parameter, from the regular expression `items` field. We could have used a more sophisticated regular expression to take care of that too. However, this was intentional for contrast in demonstrating how regular expression processing is a more elegant solution in this situation.

TIP

As an enhancement, you might log any situations where `match.Success` is `false` if you’re concerned about losing data and/or want to know if there’s a bug in the regular expression or original data formatting.

Example 2-1.

Finally, the application returns the new line items to the calling code, `Main`, so it can save them.

Chapter 3. Ensuring Quality

3.1 Overview

All the best practices, fancy algorithms, and patterns in the world mean nothing if the code doesn’t work properly. We all want to build the best app possible and minimize bugs. That’s why this entire chapter is about ways to build a quality product.

Maintainability

When working on a team, other developers must work with the code you write. They add new features and fix bugs. If you write code that’s easy to read, it will be more maintainable - that is, other developers will be able to read and understand it. Even if you’re the sole developer, coming back to code you’ve written in the past can be a new experience. Increased maintainability leads to less new bugs being introduced and quicker task turnaround. Fewer bugs mean less software life-cycle costs and more time for other value-added features. It is this spirit of maintainability motivating the content in this chapter.

Catching Problems

Users can and will use apps in a way that finds the one bug that we never thought would happen. The sections on Simplifying Parameter Validation and Protecting Against `NullReferenceException` give essential tools to help. Proper exception handling is an important skill and you’ll learn that too.

Correct Code

Although unit testing has been with us for a long time, it isn’t a solved problem. A lot of developers still don’t write unit tests. However, it’s such an important topic that the first section in this chapter shows you how to write a unit test.

3.2 3.1 Writing a Unit Test

Problem

Quality Assurance professionals are continually finding problems during integration testing and you want to reduce the number of bugs that are checked in.

Solution

Here’s the code to test:

```
public enum CustomerType
{
    Bronze,
    Silver,
    Gold
}

public class Order
{
    public decimal CalculateDiscount(CustomerType custType, decimal
amount)
    {
        decimal discount;

        switch (custType)
        {
            case CustomerType.Silver:
                discount = amount * 1.05m;
                break;
            case CustomerType.Gold:
                discount = amount * 1.10m;
                break;
            case CustomerType.Bronze:
            default:
                discount = amount;
                break;
        }
        return discount;
    }
}
```

A separate test project has unit tests:

```
public class OrderTests
{
    [Fact]
    public void CalculateDiscount_WithBronzeCustomer_GivesNoDiscount()
    {
        const decimal ExpectedDiscount = 5.00m;

        decimal actualDiscount =
            new Order().CalculateDiscount(CustomerType.Bronze, 5.00m);

        Assert.Equal(ExpectedDiscount, actualDiscount);
    }

    [Fact]
    public void
CalculateDiscount_WithSilverCustomer_GivesFivePercentDiscount()
    {
        const decimal ExpectedDiscount = 5.25m;

        decimal actualDiscount =
            new Order().CalculateDiscount(CustomerType.Silver, 5.00m);

        Assert.Equal(ExpectedDiscount, actualDiscount);
    }

    [Fact]
    public void
CalculateDiscount_WithGoldCustomer_GivesTenPercentDiscount()
    {
        const decimal ExpectedDiscount = 5.50m;

        decimal actualDiscount =
            new Order().CalculateDiscount(CustomerType.Gold, 5.00m);

        Assert.Equal(ExpectedDiscount, actualDiscount);
    }
}
```

Discussion

The code to test is the System Under Test (SUT) and the code that tests it is called a unit test. Unit tests are typically in a separate project, referencing the SUT, avoiding bloating the deliverable assembly by not shipping test code with production code. The size of the unit to test is often a type like a class, record, or struct. The solution has an `Order` class (SUT) with a `CalculateDiscount` method. The unit tests ensure `CalculateDiscount` operates correctly.

There are several well-known unit test frameworks and you can try a few and use the one you like best. These examples use XUnit. Most of the unit test frameworks integrate with Visual Studio and other IDE's.

Unit test frameworks help identify unit test code with attributes. Some have an attribute for the test class, but XUnit doesn't. With XUnit, you only need to add a `[Fact]` attribute to the unit test and it will work with the IDE or other tooling you're using.

The naming convention of the unit tests indicate their purpose, making it easy to read. The `OrderTests` class indicates that it's unit tests operate on the `Order` class. Unit test method names have the following pattern:

```
<MethodToTest>_<Condition>_<ExpectedOutcome>
```

The first unit test, `CalculateDiscount_WithBronzeCustomer_GivesNoDiscount`, follows this pattern where:

- `CalculateDiscount` is the method to test
- `WithBronzeCustomer` specifies what is unique about the input for this particular test
- `GivesNoDiscount` is the result to verify

The organization of the unit tests uses a format called Arrange, Act, and Assert (AAA). The Arrange section creates all the necessary types for the test to occur. In these unit tests, the arrange creates a `const ExpectedDiscount`.

In more complex scenarios the Arrange part will instantiate input parameters that establish the appropriate conditions for the test. In this example, the conditions were so simple that they are written as constant parameters in the Act part.

The Act part is a method call that takes parameters, if any, that create the conditions to be tested. In these examples, the Act part instantiates an `Order` instance and calls `CalculateDiscount`, assigning the response to `actualDiscount`.

The `Assert` class belongs to the XUnit testing framework. Appropriately named, you use `Assert` statements in the Assert part of the test.

Notice the naming convention I used for `actualDiscount` and `ExpectedDiscount`. The `Assert` class has several methods, with `Equal` being very popular because it allows you to compare what you expected to what you actually received during the Act part.

The benefits you get from unit tests potentially include better code design, verification that the code does what was intended, protection against regressions, deployment validation, and documentation. The key word here is “potential” because different people and/or teams choose the benefit they want from unit tests.

The better code design comes from writing tests before writing the code. You might have heard this technique discussed in agile or Behavior Driven Development (BDD) environments. In making the developer think about expected behavior ahead of time, a clearer design might evolve. On the other hand, you might want to write unit tests after the code is written. Developers write code and unit tests both ways and opinions differ on what is preferable. Ultimately, having the tests, regardless of how you arrived there, is more likely to improve code quality better than not having tests.

The second point of verifying that the code does what is intended is the biggest benefit. For simple methods that serve more as code documentation, it isn’t a big deal. However, for complex algorithms or something critical like ensuring customers receive the right discount, unit tests save the day.

Another important benefit is protecting against regressions. Not if, but when, the code changes, you or another developer could introduce bugs where the original intent of the code was accidentally changed. By running the unit tests after changing code, you can find and fix bugs at the source and not later by Quality Assurance professionals or (even worse) customers.

With modern DevOps, we have the ability to automate builds through continuous deployment. You can add unit test runs to a DevOps pipeline, which catches errors before they're merged with the rest of the code. The more unit tests you have, the more this technique reduces the possibility of any developers breaking the build.

Finally, you have another level of documentation. That's why the naming conventions for unit tests are important. If another developer, unfamiliar with an application, needs to understand the code, the unit tests can explain what the correct behavior of that code should be.

This discussion was to get you started with unit tests, if you aren't already using them. You can learn more by searching for XUnit and other unit testing frameworks to see how they work. If you haven't done so yet, please review Section 1.2 Removing Explicit Dependencies because it describes techniques that make code more testable.

See Also

1.2 Removing Explicit Dependencies

3.3 3.2 Versioning Interfaces Safely

Problem

You need to update an interface in one of your libraries without breaking deployed code.

Solution

Interface before update:

```
public interface IOrder
{
    string PrintOrder();
}
```

Interface after update:

```
public interface IOrder
{
    string PrintOrder();

    decimal GetRewards() => 0.00m;
}
```

`CompanyOrder` before update:

```
public class CompanyOrder : IOrder
{
    public string PrintOrder()
    {
        return "Company Order Details";
    }
}
```

`CompanyOrder` after update:

```
public class CompanyOrder : IOrder
{
    decimal total = 25.00m;

    public string PrintOrder()
    {
        return "Company Order Details";
    }

    public decimal GetRewards()
    {
        return total * 0.01m;
    }
}
```

`CustomerOrder` before and after update:

```
class CustomerOrder : IOrder
{
    public string PrintOrder()
    {
        return "Customer Order Details";
    }
}
```

Here’s how the types are used:

```
class Program
```

```
{
    static void Main()
    {
        var orders = new List<IOrder>
        {
            new CustomerOrder(),
            new CompanyOrder()
        };

        foreach (var order in orders)
        {
            Console.WriteLine(order.PrintOrder());
            Console.WriteLine($"Reward: {order.GetRewards()}");
        }
    }
}
```

Discussion

Prior to C# 8, we couldn't add new members to an existing interface without changing all the types that implement that interface. If those implementing types resided in the same code base, it was a recoverable change. However, for framework libraries where developers relied on an interface to work with that library, this would be a breaking change.

The solution describes how to update interfaces and the effects. The scenario is for a customer that might want to apply some reward points, earned previously, to a current order.

Looking at `IOrder`, you can see that the after update version adds a `GetRewards` method. Historically, interfaces were not allowed to have implementations. However, in the new version of `IOrder` The `GetRewards` method has a default implementation that returns `$0.00` as the reward.

The solution also has a before and after version of the `CompanyOrder` class, where the after version contains an implementation of `GetRewards`. Now, instead of the default implementation, any code invoking `GetRewards` through a `CompanyOrder` instance will execute the `CompanyOrder` implementation.

In contrast, the solution shows a `CustomerOrder` class that also implements `IOrder`. The difference here is that `CustomerOrder` didn't change. Any code invoking `GetRewards` through a `CompanyOrder` instance will execute the default `IOrder` implementation.

The `Program Main` method shows how this works. The `orders` is a list of `IOrder`, with run-time instances of `CustomerOrder` and `CompanyOrder`. The `foreach` loops through `orders`, calling `IOrder` methods. As described earlier, invoking `GetRewards` for the `CompanyOrder` instance uses that class' implementation, whereas `CustomerOrder` uses the default `IOrder` implementation.

Essentially, the change means that if a developer implements `IOrder` in their own class, such as `CustomerOrder`, their code doesn't break when updating the library to the latest version.

3.4 3.3 Simplifying Parameter Validation

Problem

You're always looking for ways to simplify code, including parameter validation.

Solution

Verbose parameter validation syntax:

```
static void ProcessOrderOld(
    string customer, List<string> lineItems)
{
    if (customer == null)
    {
        throw new ArgumentNullException(nameof(customer), $"
{nameof(customer)} is required.");
    }

    if (lineItems == null)
    {
        throw new ArgumentNullException(nameof(lineItems), $"
{nameof(lineItems)} is required.");
    }

    Console.WriteLine($"Processed {customer}");
}
```

Brief parameter validation syntax:

```
static void ProcessOrderNew(
    string customer, List<string> lineItems)
{
    _ = customer ?? throw new ArgumentNullException(nameof(customer),
$"{nameof(customer)} is required.");
    _ = lineItems ?? throw new
ArgumentNullException(nameof(lineItems), $"{nameof(lineItems)} is
required.");

    Console.WriteLine($"Processed {customer}");
}
```

Discussion

The first code of a public method is often concerned with parameter validation, which can sometimes be verbose. This section shows how to save a few lines of code so they don't obscure the code pertaining to the original purpose of the method.

The solution has two parameter validation techniques: verbose and brief. The verbose method is typical, where the code ensures that a parameter isn't `null` and throws otherwise. The parenthesis aren't required in this single-line throws statement, but some developers/teams prefer for them to be there anyway.

The brief method is an alternative that can save a few lines of code. They rely on newer features of C# - the variable discard, _ and coalescing operator, `??`.

On the line validating `customer`, the code starts with an assignment to the discard, because we need an expression. The coalescing operator is a guard that detects when the expression is `null`. When the expression is `null` the next statement executes, throwing an exception.

TIP

This example was for parameter evaluation. However, there are other scenarios where the code encounters a variable that was set to null and needs to throw for an invalid condition or a situation that never should have occurred. This technique lets you handle that quickly in a single line of code.

See Also

3.4 Protecting Code From `NullReferenceException`

3.5 3.4 Protecting Code From `NullReferenceException`

Problem

You’re building a reusable library and need to communicate nullable reference semantics.

Solution

This is old-style code that doesn’t handle null references:

```
class OrderLibraryNonNull
{
    // null property
    public string DealOfTheDay { get; set; }

    // method with null parameter
    public void AddItem(string item)
    {
        Console.Write(item.ToString());
    }


    // method with null return value
    public List<string> GetItems()
    {
        return null;
    }


    // method with null type parameter
    public void AddItems(List<string> items)
    {
        foreach (var item in items)
            Console.WriteLine(item.ToString());
    }
}
```

The following project file, turns on the new nullable reference feature:

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>netcoreapp3.1</TargetFramework>
        <RootNamespace>Section_03_04</RootNamespace>
        <Nullable>enable</Nullable>
    </PropertyGroup>
</Project>
```

Here’s the updated library code that communicates nullable references:

```
class OrderLibraryWithNull
{
    // null property
    public string? DealOfTheDay { get; set; }

    // method with null parameter
    public void AddItem(string? item)
    {
        _ = item ?? throw new ArgumentNullException(nameof(item), $"
{nameof(item)} must not be null");

        Console.Write(item.ToString());
    }


    // method with null return value
    public List<string>? GetItems()
    {
        return null;
    }


    // method with null type parameter
    public void AddItems(List<string?> items)
    {
        foreach (var item in items)
            Console.WriteLine(item?.ToString() ?? "None");
    }
}
```

This is an example of old-style consuming code that ignores nullable references:

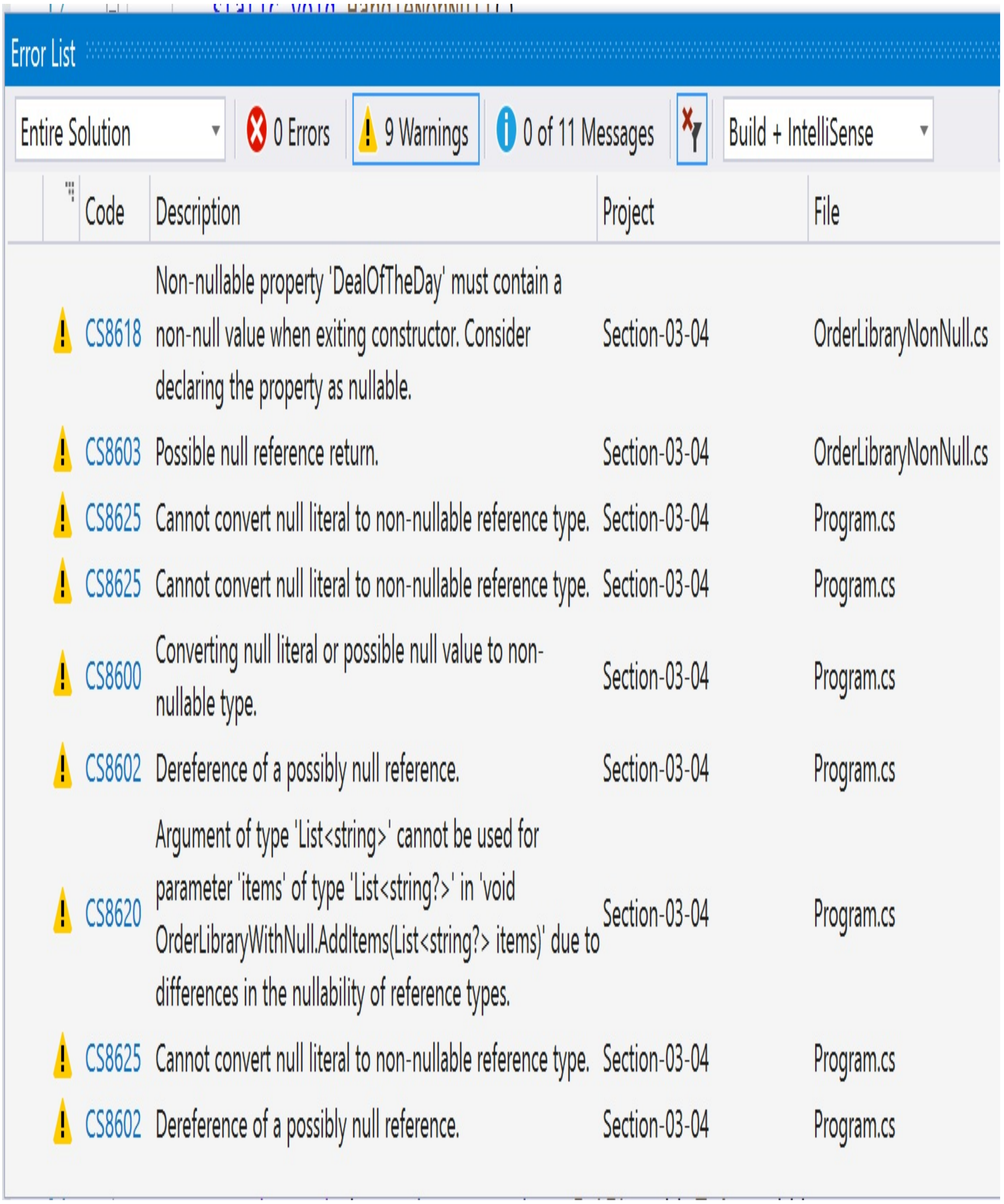
```
static void HandleWithNullNoHandling()
{
    var orders = new OrderLibraryWithNull();

    string deal = orders.DealOfTheDay;
    Console.WriteLine(deal.ToUpper());
```

```
        orders.AddItem(null);
        orders.AddItems(new List<string> { "one", null });

        foreach (var item in orders.GetItems().ToArray())
            Console.WriteLine(item.Trim());
    }
```

This table shows the warning wall the user sees from consuming code that ignores nullable references:



Error List

Entire Solution | 0 Errors | 9 Warnings | 0 of 11 Messages | Build + IntelliSense

Code	Description	Project	File
CS8618	Non-nullable property 'DealOfTheDay' must contain a non-null value when exiting constructor. Consider declaring the property as nullable.	Section-03-04	OrderLibraryNonNull.cs
CS8603	Possible null reference return.	Section-03-04	OrderLibraryNonNull.cs
CS8625	Cannot convert null literal to non-nullable reference type.	Section-03-04	Program.cs
CS8625	Cannot convert null literal to non-nullable reference type.	Section-03-04	Program.cs
CS8600	Converting null literal or possible null value to non-nullable type.	Section-03-04	Program.cs
CS8602	Dereference of a possibly null reference.	Section-03-04	Program.cs
CS8620	Argument of type 'List<string>' cannot be used for parameter 'items' of type 'List<string?>' in 'void OrderLibraryWithNull.AddItems(List<string?> items)' due to differences in the nullability of reference types.	Section-03-04	Program.cs
CS8625	Cannot convert null literal to non-nullable reference type.	Section-03-04	Program.cs
CS8602	Dereference of a possibly null reference.	Section-03-04	Program.cs

Finally, here’s how the consuming code can properly react to the reusable library with the proper checks and validation for nullable references:

```
static void HandleWithNullAndHandling()
{
    var orders = new OrderLibraryWithNull();

    string? deal = orders.DealOfTheDay;
    Console.WriteLine(deal?.ToUpper() ?? "Deals");

    orders.AddItem(null);
    orders.AddItems(new List<string?> { "one", null });

    List<string>? items = orders.GetItems();

    if (items != null)
        foreach (var item in items.ToArray())
            Console.WriteLine(item.Trim());
}
```

Discussion

If you've been programming C# for any length of time, it's likely that you've encountered `NullReferenceExceptions`. A `NullReferenceException` occurs when referencing a member of a variable that is still null, essentially trying to use an object that doesn't yet exist. Nullable references, first introduced in C# 8, help write higher quality code by reducing the number of `NullReferenceException` exceptions being thrown. The whole concept resolves around giving the developer compile-time notice of situations where variables are null and could potentially result in a thrown `NullReferenceException`. This scenario revolves around the need to write a reusable library, perhaps a separate class library or NuGet package, for other developers. Your goal is to let them know where a potential null reference occurs in the library so they can write code to protect against a `NullReferenceException`.

To demonstrate, the solution shows library code that doesn't communicate null references. Essentially, this is old-style code, representing what developers would have written before C# 8. You'll also see how to configure a project to support C# 8 Nullable References. Then you'll see how to change that library code so it communicates null references to a developer who might consume it. Finally, you'll see two examples of consuming code: one that doesn't handle null references and another that shows how to protect against null references.

In the first solution example, the `OrderLibraryNonNull` class has members with parameters or return types that are reference types, such as `string` and

`List<string>`, both of which could potentially be set to null. In both a nullable and non-nullable context, this code won't generate any warnings. Even in a nullable context, the reference types aren't marked as nullable and dangerously communicate to users that they'll never receive a `NullReferenceException`. However, because there could be potential `NullReferenceExceptions`, we don't want to write our code like this anymore.

The following XML listing is the project file with a `/Project/PropertyGroup/Nullable` element. Setting this to `true` puts the project in a nullable context. Putting a separate class library into a nullable context might provide warnings for the class library developer, but the consumer of that code won't ever see those warnings.

The next solution code snippet for `OrderLibraryWithNull` fixes this problem. Compare it with `OrderLibraryNonNull` to tell the differences. When evaluating null references, go member by member through a type to think about how parameters and return values affect a consumer of your library in regards to null references. There are a lot of different null scenarios, but this example captures three common ones: property type, method parameter type, and type parameter type.

NOTE

There are times when a method genuinely doesn't ever return a null reference. Then it makes sense to not use the nullable operator to communicate to the consumer that they don't need to check for null.

The `DealOfTheDay` getter property returns a `string`, which could potentially be a `null` value. Fix those with the nullable operator, `?` and return `string?`.

`AddItems` is similar, except it takes a `string` parameter. Since `string` could be `null`, changing it to `string?` lets the compiler know too. Notice how I used the Simplified Parameter Checking described in section 3.3.

The `GetItems` method returns a `List<string>` and `List<T>` is a reference type. Therefore, changing that to `List<string>?` fixes the problem.

Finally, here's one that's a little tricky. The items parameter in `AddItems` is a `List<string>`. It's easy enough to do a parameter check to test for a `null` parameter, but leaving the nullable operator off is also a good approach to let the user know that they shouldn't pass a `null` value.

That said, what if one of the values in the `List<string>` were `null?` In this case, it's a `List<string>`, but what about scenarios where the user were allowed to pass in a `Dictionary<string, string>` where the value could be `null`. Then annotate the type parameter, as the example does with `List<string?>` to say it's okay for a value to be `null`. Since you know that the parameter can be `null`, it's important to check before referencing its members - to avoid a `NullReferenceException`.

Now you have library code that's useful for a consumer. However, it will only be useful if the consumer puts their project into a nullable context too, as shown in the project file.

The `HandleWithNullNoHandling` method shows how a developer might have written code before C# 8. However, once they put the project into a nullable context, they will receive several warnings as illustrated in the warning wall, showing the Visual Studio Error List window. Comparing that with the `HandleWithNullAndHandling` method, the contrast is strong.

The whole process cascades, so start at the top of the method and work your way down.

1. Because the `DealOfTheDay` getter can return `null`, set deal type to `string?`.
2. Since `deal` can be `null`, use the null reference operator and a coalescing operator to ensure `Console.WriteLine` has something sensible to write.
3. The type passed to `AddItems` needs to be `List<string?>` to make the statement that you're aware that an item can be `null`.
4. Instead of in-lining `orders.GetItems` in the `foreach` loop, refactor it out into a new variable. This lets you check for `null` to avoid consuming a `null` iterator.

See Also

3.3 Simplified Parameter Checking

3.6 3.5 Avoiding Magic strings

Problem

A `const` string resides in multiple places in the app and you need a way to change it without breaking other code.

Solution

Here’s an Order object:

```
class Order
{
    public string DeliveryInstructions { get; set; }

    public List<string> Items { get; set; }
}
```

Here are some constants:

```
class Delivery
{
    public const string NextDay = "Next Day";
    public const string Standard = "Standard";
    public const string LowFare = "Low Fare";

    public const int StandardDays = 7;
}
```

This is the program that uses Order and constants to calculate number of days of delivery:

```
static void Main(string[] args)
{
    var orders = new List<Order>
    {
        new Order { DeliveryInstructions = Delivery.LowFare },
        new Order { DeliveryInstructions = Delivery.NextDay },
        new Order { DeliveryInstructions = Delivery.Standard },
    };

    foreach (var order in orders)
    {
        int days;

        switch (order.DeliveryInstructions)
        {
            case Delivery.LowFare:
                days = 15;
                break;
            case Delivery.NextDay:
                days = 1;
                break;
            case Delivery.Standard:
            default:
                days = Delivery.StandardDays;
                break;
        }

        Console.WriteLine(order.DeliveryInstructions);
        Console.WriteLine($"Expected Delivery Day(s): {days}");
    }
}
```

Discussion

After developing software for a while, most developers have seen their share of magic values, which are literal values, such as strings and numbers, written directly into an expression. From the perspective of the original developer, they might not be a huge problem. However, from the perspective of a maintenance developer, those literal values don't immediately make sense. It's as if they magically appeared out of nowhere or makes it feel like magic that the code even works because the meaning of literal value isn't obvious.

The goal is to write code that gives a future maintainer a chance to understand. Otherwise, project costs increase because of the time wasted trying to figure out what some seemingly random number is. The solution is often to replace the literal value with a variable whose name expresses the semantics of the value or why it's there. A commonly held belief is that readable code has a more maintainable lifetime than comments.

Going further, a local constant helps a method with readability, but constants are often reusable. The solution example demonstrates how some reusable constants can be placed in their own class for reuse by other parts of the code.

In addition to `items`, the `Order` class has a `DeliveryInstructions` property. Here, we make the assumption that there is a finite set of delivery instructions.

The `Delivery` class has `const string` values for `NextDay`, `Standard`, and `LowFare`, characterizing how an order should be delivered. Also, notice that this class has a `StandardDays` value, set to `7`. Which program would you rather read - the one that uses `7` or the one that uses a constant named `StandardDays`?

NOTE

You might first consider that the `const string` values in the `Delivery` class might be better candidates for an enum. However, notice that they have spaces. Also, they'll be written with the `order`. While there are techniques for using enums as `string`, this was simple. Also, sometimes you need a specific `string` value for lookup in some scenarios. It's a matter of opinion and what you think the right tool for the right job is. If you find a scenario where enums are more convenient, then do that.

The program that uses `Orders` and `Delivery` is trying to calculate the number of days for delivery, based on the order's `DeliveryInstructions`. There are three orders in a list, each with a different setting for `DeliveryInstructions`. The `foreach` loop iterates over those orders with a `switch` statement that sets the number of delivery days, depending on `DeliveryInstructions`.

Notice that both order list construction and the `switch` statement use constants from `Delivery`. Had that not been done, there would have been `strings` everywhere. Now, it's easy to code with Intellisense support, there is no duplication because the `string` is in one place, and the opportunity for mistyping is minimized. Further, if the `strings` need to change, that happens in one place. Additionally, you get IDE refactoring support to change the name everywhere that constant appears in the application.

3.7 3.6 Customizing Class String Representation

Problem

The class representation in the debugger, string parameters, and log files is illegible and you want to customize its appearance.

Solution

Here's a class with a custom `ToString` method:

```
using System;
using System.Text;

namespace Section_03_06
{
    class Order
    {
        public int ID { get; set; }

        public string CustomerName { get; set; }

        public DateTime Created { get; set; }

        public decimal Amount { get; set; }

        public override string ToString()
        {
            var stringBuilder = new StringBuilder();

            stringBuilder.Append(nameof(Order));
            stringBuilder.Append(" {\n");

            if (PrintMembers(stringBuilder))
                stringBuilder.Append(" ");

            stringBuilder.Append("\n}");

            return stringBuilder.ToString();
        }

                protected virtual bool PrintMembers(StringBuilder
builder)
                {
                        builder.Append("   " + nameof(ID));
                        builder.Append(" = ");
                        builder.Append(ID);
                        builder.Append(", \n");
                        builder.Append("  " + nameof(CustomerName));
                        builder.Append(" = ");
                        builder.Append(CustomerName);
                        builder.Append(", \n");
                        builder.Append("  " + nameof(Created));
                        builder.Append(" = ");
                        builder.Append(Created.ToString("d"));
                        builder.Append(", \n");
                        builder.Append("  " + nameof(Amount));
                        builder.Append(" = ");
                        builder.Append(Amount);

                        return true;
                }
        }
}
```

Here’s an example of how that’s used:

```
class Program
{
    static void Main(string[] args)
    {
        var order = new Order
        {
            ID = 7,
            CustomerName = "Acme",
            Created = DateTime.Now,
            Amount = 2_718_281.83m
        };

        Console.WriteLine(order);
    }
}
```

And here’s the output:

```
Order {
  ID = 7,
  CustomerName = Acme,
  Created = 1/23/2021,
  Amount = 2718281.83
}
```

Discussion

Some types are complex and viewing an instance in the debugger is cumbersome because you need to dig multiple levels to examine values. Modern IDEs make this easier, but sometimes it's nicer to have a more readable representation of the class.

That's where overriding the `ToString` method comes in. `ToString` is a method of the `Object` type, which all types derive from. The default implementation is the fully qualified name of the type, which is `Section_03_06.Order` for the `Order` class in the solution. Since it's a virtual method, you can override it.

In fact, the `Order` class overrides `ToString` with its own representation. As covered in Section 2.1 Processing strings Efficiently, the implementation uses `StringBuilder`. The format is using the name of the object with properties inside of curly braces, as shown in the output.

The demo code, in `Main` generates this output via the `Console.WriteLine`. This works because `Console.WriteLine` calls an object's `ToString` method if a parameter isn't already a `string`.

See Also

2.1 Processing strings Efficiently

3.8 3.7 Rethrowing Exceptions

Problem

An app is throwing exceptions, yet the messages are missing information and you need to ensure all relevant data is available during processing.

Solution

This object throws an exception:

```
class Orders
{
    public void Process()
    {
        throw new IndexOutOfRangeException(
            "Expected 10 orders, but found only 9.");
    }
}
```

Here are different ways to handle the exception:

```
    class OrderOrchestrator
    {
        public static void HandleOrdersWrong()
        {
            try
            {
                new Orders().Process();
            }
            catch (IndexOutOfRangeException ex)
            {
                throw new ArgumentException(ex.Message);
            }
        }

        public static void HandleOrdersBetter1()
        {
            try
            {
                new Orders().Process();
            }
            catch (IndexOutOfRangeException ex)
            {
                throw new ArgumentException("Error Processing Orders",
    ex);
            }
        }

        public static void HandleOrdersBetter2()
        {
            try
            {
                new Orders().Process();
            }
            catch (IndexOutOfRangeException)
            {
                throw;
            }
        }
    }
```

This program tests each exception handling method:

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            OrderOrchestrator.HandleOrdersWrong();
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("Handle Orders Wrong:\n" + ex);
        }

        try
        {
            OrderOrchestrator.HandleOrdersBetter1();
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("\n\nHandle Orders Better #1:\n" + ex);
        }

        try
        {
            OrderOrchestrator.HandleOrdersBetter2();
        }
        catch (IndexOutOfRangeException ex)
        {
            Console.WriteLine("\n\nHandle Orders Better #2:\n" + ex);
        }
    }
}
```

Here’s the output:

```
    Handle Orders Wrong:
    System.ArgumentException: Expected 10 orders, but found only 9.
       at Section_03_07.OrderOrchestrator.HandleOrdersWrong() in
C:\Projects\CSharp9Cookbook\CSharp9Cookbook\Chapter03\Section-03-
07\OrderOrchestrator.cs:line 15
       at Section_03_07.Program.Main(String[] args) in
C:\Projects\CSharp9Cookbook\CSharp9Cookbook\Chapter03\Section-03-
07\Program.cs:line 11


    Handle Orders Better #1:
```

```
    System.ArgumentException: Error Processing Orders
     ---> System.IndexOutOfRangeException: Expected 10 orders, but
found only 9.
       at Section_03_07.Orders.Process() in
C:\Projects\CSharp9Cookbook\CSharp9Cookbook\Chapter03\Section-03-
07\Orders.cs:line 9
       at Section_03_07.OrderOrchestrator.HandleOrdersBetter1() in
C:\Projects\CSharp9Cookbook\CSharp9Cookbook\Chapter03\Section-03-
07\OrderOrchestrator.cs:line 23
       --- End of inner exception stack trace ---
       at Section_03_07.OrderOrchestrator.HandleOrdersBetter1() in
C:\Projects\CSharp9Cookbook\CSharp9Cookbook\Chapter03\Section-03-
07\OrderOrchestrator.cs:line 27
       at Section_03_07.Program.Main(String[] args) in
C:\Projects\CSharp9Cookbook\CSharp9Cookbook\Chapter03\Section-03-
07\Program.cs:line 20


    Handle Orders Better #2:
    System.IndexOutOfRangeException: Expected 10 orders, but found
only 9.
       at Section_03_07.Orders.Process() in
C:\Projects\CSharp9Cookbook\CSharp9Cookbook\Chapter03\Section-03-
07\Orders.cs:line 9
       at Section_03_07.OrderOrchestrator.HandleOrdersBetter2() in
C:\Projects\CSharp9Cookbook\CSharp9Cookbook\Chapter03\Section-03-
07\OrderOrchestrator.cs:line 35
       at Section_03_07.Program.Main(String[] args) in
C:\Projects\CSharp9Cookbook\CSharp9Cookbook\Chapter03\Section-03-
07\Program.cs:line 29
```

Discussion

There are various ways to handle exceptions, with some being better than others. From a troubleshooting perspective, we generally want a log of exceptions with enough meaningful information to help solve the problem. That is the guide of this section in determining what the better solution should be.

The `Orders` class `Process` method throws an `IndexOutOfRangeException` and the `OrderOrchestrator` class handles that exception in a few different ways: one which you should avoid and two that are better, depending on what makes sense for your scenario.

The `HandleOrdersWrong` method takes the `Message` property of the original exception and throws a new `ArgumentException` with that message as its input. The scenario models a situation where the handling analyzes the situation and tries to throw an exception that makes more sense or provides more information than what the original exception offered. However, this causes another problem where we lose stack trace information that's critical to fixing the problem. This example has a relatively shallow hierarchy, but in practice the exception could have been thrown via multiple levels down and arrived via various paths. You can see this problem in the output where the stack trace shows that the exception originated in the `OrderOrchestrator.HandleOrdersWrong` method, rather than it's true source in `Orders.Process`.

The `HandleOrdersBetter1` method improves on this scenario by adding an additional argument, `ex`, to the `innerException` parameter. This provides the best of both worlds because you can now throw an exception with additional data, as well as preserving the entire stack trace. In the output, delimited by `--- End of inner exception stack trace ---`, you an see that the path of the exception originated in `Orders.Process`.

The `HandleOrdersBetter2` simply throws the original exception. The assumption here is that the logic wasn't able to do something intelligent with the exception or log and rethrow. As shown in the output, the stack trace also originates at `Orders.Process`.

There are a lot of strategies for handling exceptions and this covers one aspect of that - when you need to re-throw, what is the best way to handle that through the consideration of preserving stack trace for later debugging. As always, think about your scenario and what makes sense to you.

3.9 3.8 Managing Process Status

Problem

The user started a process, but after an exception the user interface status wasn't updated.

Solution

This method throws an exception:

```
static void ProcessOrders()
{
    throw new ArgumentException();
}
```

This is the code you should not write:

```
static void Main()
{
    Console.WriteLine("Processing Orders Started");

    ProcessOrders();

    Console.WriteLine("Processing Orders Complete");
}
```

Here’s the code you should write instead:

```
static void Main()
{
    try
    {
        Console.WriteLine("Processing Orders Started");

        ProcessOrders();
    }
    catch (ArgumentException ae)
    {
        Console.WriteLine('\n' + ae.ToString() + '\n');
    }
    finally
    {
        Console.WriteLine("Processing Orders Complete");
    }
}
```

Discussion

The problem statement mentions there was an exception that occurred, which is true. However, from a user perspective, they won’t receive a message or status explaining that a problem occurred and their job didn’t finish. That’s because in the first `Main` method, which you shouldn’t write, if an exception throws during `ProcessOrder`, the “Processing Orders Complete” message won’t appear to the user.

This is a good use case for a `try/finally` block, which the second `Main` method uses. Put all the code that should run in a `try` block and a final status in the `finally` block. If an exception throws, you can catch it, log, and let the user know that their job was unsuccessful.

Although this was an example in a Console application, this is a good technique for UI code too. When starting a process, you might have a wait notification like an hourglass or progress indicator. Turning the notification off is a task that the `finally` block can help with also.

See Also

3.9 Building Resilient Network Connections 3.10 Measuring Performance

3.10 3.9 Building Resilient Network Connections

Problem

The app communicates with an unreliable back-end service and you want to prevent it from failing.

Solution

This method throws an Exception:

```
static async Task GetOrdersAsync()
{
    throw await Task.FromResult(
        new HttpRequestException(
            "Timeout", null, HttpStatusCode.RequestTimeout));
}
```

Here’s a technique to handle network errors:

```
public static async Task Main()
{
    const int DelayMilliseconds = 500;
    const int RetryCount = 3;

    bool success = false;
    int tryCount = 0;

    try
    {
        do
        {
            try
            {
                Console.WriteLine("Getting Orders");
                await GetOrdersAsync();

                success = true;
            }
            catch (HttpRequestException hre)
                when (hre.StatusCode == HttpStatusCode.RequestTimeout)
            {
                tryCount++;

                int millisecondsToDelay = DelayMilliseconds *
tryCount;
                Console.WriteLine(
                    $"Exception during processing - " +
                    $"delaying for {millisecondsToDelay}
milliseconds");

                await Task.Delay(millisecondsToDelay);
            }

        } while (tryCount < RetryCount);
    }
    finally
    {
        if (success)
            Console.WriteLine("Operation Succeeded");
        else
            Console.WriteLine("Operation Failed");
    }
}
```

And here’s the output:

```
Getting Orders
Exception during processing - delaying for 500 milliseconds
Getting Orders
Exception during processing - delaying for 1000 milliseconds
Getting Orders
Exception during processing - delaying for 1500 milliseconds
Operation Failed
```

Discussion

Anytime you’re doing out-of-process work, there’s a possibility or errors or timeouts. Often you don’t have control of the application you’re interacting with and it pays to write defensive code. In particular, code that does networking is prone to errors unrelated to the quality of code at either end of the connection because, to name a few causes, of latency, timeouts, and hardware issues.

This solution simulates a network connection issue through `GetOrdersAsync`. It throws an `HttpRequestException` with a `RequestTimeout` status. This is typical and the `Main` method shows how to mitigate these types of problems. The goal is to retry the connection a certain number of times with delay between tries.

First, notice that `success` initializes to `false` and the `finally`, of the `try/finally`, lets the user know the result of the operation, based on `success`. Following the nesting of `try/do/try`, the last line of the `try` block sets `success` to `true` because all of the logic is complete - if an exception occurred earlier, the program would not have reached that line.

The `do/while` loop iterates `RetryCount` times. We initialize `tryCount` to `0` and increment it in the `catch` block. That’s because if there’s an error, we know we’ll retry, and want to ensure we don’t exceed a specified number of retries. `RetryCount` is a `const`, initialized to `3`. You can adjust `RetryCount` to as many times as makes sense to you. If the operation is time sensitive, you might want to limit retries and send a notification of a critical error. Another scenario might be that you know the other end of the connection will eventually come back on-line and can set `RetryCount` to a very high number.

Whenever there is an exception, you often don’t want to immediately make the request again. One of the reasons why the timeout occurred might be because the other endpoint might not scale well and overloading it with more requests can overwhelm the server. Also, some 3rd party APIs rate-limit clients and immediate back-to-back requests eat up the rate-limit count. Some API providers might even block your app for excessive connection requests.

The `DelayMilliseconds` helps your retry policy, initialized to `500` milliseconds. You might adjust this if you find that retries are still too fast. If a single delay time works, then you can use that. However, a lot of situations call for a linear or exponential back-off strategy. You can see that the solution uses a linear back-off, multiplying `DelayMilliseconds` by `tryCount`. Since `tryCount` initializes to `0`, we increment it first.

TIP

You might want to log retries as Warning, rather than Error. Administrators, QA, or anyone looking at the logs (or reports) might be unnecessarily alarmed. They see what looks like errors, whereas your application is reacting and repairing appropriately to typical network behavior.

Alternatively, you might need to use an exponential back-off strategy, such as taking `DelayMilliseconds` to the `tryCount` power - `Math.Pow(DelayMilliseconds, tryCount)`. You might experiment, e.g. log errors and review periodically, to see what works best for your situation.

3.11 3.10 Measuring Performance

Problem

You know a couple ways to write an algorithm and need to test which algorithm performs the best.

Solution

Here’s the object type we’ll operate on:

```
class OrderItem
{
    public decimal Cost { get; set; }
    public string Description { get; set; }
}
```

This is the code that creates a list of OrderItem:

```
static List<OrderItem> GetOrderItems()
{
    const int ItemCount = 10000;

    var items = new List<OrderItem>();
    var rand = new Random();

    for (int i = 0; i < ItemCount; i++)
        items.Add(
            new OrderItem
            {
                Cost = rand.Next(i),
                Description = "Order Item #" + (i + 1)
            });

    return items;
}
```

Here’s an inefficient string concatenation method:

```
    static string DoStringConcatenation(List<OrderItem> lineItems)
    {
        var stopwatch = new Stopwatch();

        try
        {
            stopwatch.Start();

            string report = "";

            foreach (var item in lineItems)
                report += $"{item.Cost:C} - {item.Description}";

            Console.WriteLine($"Time for String Concatenation: 
    {stopwatch.ElapsedMilliseconds}");

            return report;
        }
        finally
        {
            stopwatch.Stop();
        }
    }
```

Here’s the faster `StringBuilder` method:

```
    static string DoStringBuilderConcatenation(List<OrderItem> lineItems)
    {
        var stopwatch = new Stopwatch();
        try
        {
            stopwatch.Start();

            var reportBuilder = new StringBuilder();

            foreach (var item in lineItems)
                reportBuilder.Append($"{item.Cost:C} -
    {item.Description}");

            Console.WriteLine($"Time for String Builder Concatenation:
    {stopwatch.ElapsedMilliseconds}");

            return reportBuilder.ToString();
        }
        finally
        {
            stopwatch.Stop();
        }
    }
```

This code drives the demo:

```
    static void Main()
    {
        List<OrderItem> lineItems = GetOrderItems();

        DoStringConcatenation(lineItems);

        DoStringBuilderConcatenation(lineItems);
    }
```

And here’s the output:

```
        Time for String Concatenation: 1137
        Time for String Builder Concatenation: 2
```

Discussion

Section 2.1 discussed the benefits of `StringBuilder` over string concatenation, which stressed performance as the primary driver. However, it didn’t explain how to measure the performance of the code. This section builds on that and shows how to measure algorithmic performance through code.

TIP

As our computers become increasingly faster by the year (or less), the results of the `StringBuilder` method will move closer to `0`. To experience the real magnitude of time difference between the two methods, add another `0` to `ItemCount` in `GetOrderItems`.

In both the `StringConcatenation` and `StringBuilderConcatenation` methods, you’ll find an instance of `StopWatch`, which is part of the `System.Diagnostics` namespace.

Calling `Start`, starts the timer and `Stop` stops the timer. Notice that the algorithms use `try/finally`, as described in Section 3.8 Managing Process Status, to ensure the timer stops.

At the end of each algorithm, `Console.WriteLine` uses `stopwatch.ElapsedMilliseconds` to show how much time the algorithm used.

As shown in the output, the running time difference between `StringBuilder` and string concatenation is dramatic.

See Also

2.1 Processing strings Efficiently 3.8 Managing Process Status

Chapter 4. Querying with LINQ

Language Integrated Query (LINQ) has been around since C# v3.0. It gives developers a means to query data sources, using syntax with accents of Structured Query Language (SQL). Because LINQ is part of the language, you experience features like syntax highlighting and Intellisense in Integrated Development Environments (IDEs).

LINQ is popularly known as a tool for querying databases, with the goal of reducing what is called inpedence mismatch, which is the difference between database representation of data and C# objects. Really, we can build LINQ Providers for any data technology. In fact, the author wrote an open-source provider for the Twitter API named LINQ to Twitter (*https://github.com/JoeMayo/LinqToTwitter*).

The examples in this chapter take a different approach. Instead of an external data source, they use an in-memory data source referred to as LINQ to Objects. While any in-memory data manipulation can be performed with C# loops and imperative logic, using LINQ instead can often simplify the code because of it's declarative nature - specifying what to do rather than how to do it. Each section has a unique representation of one or more entities (objects to be queried) and an `InMemoryContext` that sets up the in-memory data to be queried.

A couple recipes in this chapter are simple, such as transforming object shape and simplifying queries. However, there are important points to be made that also clarify and simplify your code.

Pulling together code from different data sources can result in confusing code. The sections on joins, left joins, and grouping describe how you can simplify these scenarios. There's also a related section for operating on sets.

A huge security problem with search forms and queries appears when developers build their queries with concatenated strings. While that might sound like a quick and easy solution, the cost is often too high. This chapter has a couple sections that show how LINQ deferred execution lets you build queries dynamically. Another section explains an important technique for search queries and how they give you the ability to use expression trees for dynamic clause generation.

4.1 Transforming Object Shape

Problem

You want data in a custom shape that differs from the original data source.

Solution

Here’s the entity to re-shape:

```
public class SalesPerson
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Address { get; set; }

    public string City { get; set; }

    public string PostalCode { get; set; }

    public string Region { get; set; }

    public string ProductType { get; set; }
}
```

This is the data source:

```
public class InMemoryContext
{
    List<SalesPerson> salesPeople =
        new List<SalesPerson>
```

```
            {
                new SalesPerson
                {
                    ID = 1,
                    Address = "123 1st Street",
                    City = "First City",
                    Name = "First Person",
                    PostalCode = "45678",
                    Region = "Region #1"
                },
                new SalesPerson
                {
                    ID = 2,
                    Address = "234 2nd Street",
                    City = "Second City",
                    Name = "Second Person",
                    PostalCode = "56789",
                    Region = "Region #2"
                },
                new SalesPerson
                {
                    ID = 3,
                    Address = "345 3rd Street",
                    City = "Third City",
                    Name = "Third Person",
                    PostalCode = "67890",
                    Region = "Region #3"
                },
            };

        public List<SalesPerson> SalesPeople => salesPeople;
    }
```

This code performs the projection that re-shapes the data:

```
    class Program
    {
        static void Main()
        {
            var context = new InMemoryContext();

            var salesPersonLookup =
                (from person in context.SalesPeople
                    select (person.ID, person.Name))
                .ToList();

            Console.WriteLine("Sales People\n");
            salesPersonLookup.ForEach(person =>
                Console.WriteLine($"{person.ID}. {person.Name}"));
        }
    }
```

Discussion

Transforming object shape is referred to as a projection in LINQ. A few common reasons you might want to do this is to create lookup lists, create a view or view model object, or translate data transfer objects (DTOs) to something your app works with better.

When doing database queries, using LINQ to Entities, or consuming DTOs, data often arrives in a format representing the original data source. However, if you want to work with domain data or bind to user interfaces (UI), the pure data representation doesn't have the right shape. Moreover, data representation often has attributes and semantics of the object-relational model (ORM) or data access library. Some developers try to bind these data objects to their UI because they don't want to create a new object type. While that's understandable, because noone wants to do more work than is necessary, problems occur because UI code is often a different shape than the data and requires it's own validation and attributes. So, the problem here is that you're using one object for two different purposes. Ideally, an object should have a single responsibility and mixing it up like this often results in confusing code that's not as easy to maintain.

Another scenario, which the solution demonstrates is the case where you only want a lookup list, with an ID and displayable value. This is useful when populating UI elements such as checkbox lists, radio button groups, combo boxes, or dropdowns. Querying entire entities is wasteful and slow (in the case of an out-of-process or cross-network database connection) when you only need the ID and something to display to the user.

The `Main` method of the solution demonstrates this. It queries the `SalesPeople` property of `InMemoryContext`, which is a list of `SalesPerson`, and the `select` clause re-shapes the result into a tuble of `ID` and `Name`.

NOTE

The `select` clause in the solution uses a tuple. However, you could project (only the requested fields) into an anonymous type, a `SalesPerson` type, or a new custom type.

Although this was an in-memory operation, the benefit of this technique comes when querying a database with a library like LINQ to Entities. In that case, LINQ to Entities translates the LINQ query into a database query that only requests the fields specified in the select clause.

4.2 Joining Data

Problem

You need to pull data from different sources into one record.

Solution

Here are the entities to join:

```
public class Product
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Type { get; set; }

    public decimal Price { get; set; }

    public string Region { get; set; }
}

public class SalesPerson
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Address { get; set; }

    public string City { get; set; }

    public string PostalCode { get; set; }
```

```
        public string Region { get; set; }

        public string ProductType { get; set; }
    }
```

This is the data source:

```
    public class InMemoryContext
    {
        List<SalesPerson> salesPeople =
            new List<SalesPerson>
            {
                new SalesPerson
                {
                    ID = 1,
                    Address = "123 1st Street",
                    City = "First City",
                    Name = "First Person",
                    PostalCode = "45678",
                    Region = "Region #1",
                    ProductType = "Type 2"
                },
                new SalesPerson
                {
                    ID = 2,
                    Address = "234 2nd Street",
                    City = "Second City",
                    Name = "Second Person",
                    PostalCode = "56789",
                    Region = "Region #2",
                    ProductType = "Type 3"
                },
                new SalesPerson
                {
                    ID = 3,
                    Address = "345 3rd Street",
                    City = "Third City",
                    Name = "Third Person",
                    PostalCode = "67890",
                    Region = "Region #3",
                    ProductType = "Type 1"
                },
                new SalesPerson
                {
                    ID = 4,
                    Address = "678 9th Street",
                    City = "Fourth City",
                    Name = "Fourth Person",
                    PostalCode = "90123",
                    Region = "Region #1",
                    ProductType = "Type 2"
                },
            };

        List<Product> products =
            new List<Product>
            {
                new Product
                {
                    ID = 1,
                    Name = "Product 1",
                    Price = 123.45m,
                    Type = "Type 2",
                    Region = "Region #1",
                },
                new Product
                {
                    ID = 2,
                    Name = "Product 2",
                    Price = 456.78m,
                    Type = "Type 2",
                        Region = "Region #2",
                },
                new Product
                {
                    ID = 3,
                    Name = "Product 3",
                    Price = 789.10m,
                    Type = "Type 3",
                    Region = "Region #1",
                },
                new Product
                {
                    ID = 4,
                    Name = "Product 4",
                    Price = 234.56m,
                    Type = "Type 2",
                    Region = "Region #1",
                },
            };

        public List<SalesPerson> SalesPeople => salesPeople;

        public List<Product> Products => products;
    }
```

This is the code that joins the entities:

```
class Program
{
    static void Main()
    {
        var context = new InMemoryContext();

        var salesProducts =
            (from person in context.SalesPeople
                join product in context.Products on
                (person.Region, person.ProductType)
                equals
                (product.Region, product.Type)
                select new
                {
                    Person = person.Name,
                    Product = product.Name,
                    product.Region,
                    product.Type
                })
            .ToList();

        Console.WriteLine("Sales People\n");

        salesProducts.ForEach(salesProd =>
            Console.WriteLine(
                $"Person: {salesProd.Person}\n" +
                $"Product: {salesProd.Product}\n" +
                $"Region: {salesProd.Region}\n" +
                $"Type: {salesProd.Type}\n"));
    }
}
```

Discussion

LINQ joins are useful when data comes from more than one source. A company might have merged and you need to pull in data from each of their databases, you might be using a micro-service architecture where the data comes from different services, or some of the data was created in-memory and you need to correlate it with database records.

Often, you can't use an ID because if the data comes from different sources they'll never match anyway. The best you can hope for is if some of the fields line up. That said, if you have a single field that matches, that's great. The `Main` method of the solution uses a composite key of `Region` and `ProductType`.

NOTE

The `select` clause uses an anonymous type for a custom projection. Another example of shaping object data as discussed in Section 4.1.

Even though this example uses a tuple for the composite key, you could use an anonymous type for the same results. The tuple is a little less syntax.

See Also

4.1 Transforming Object Shape

4.3 Performing Left Joins

Problem

You need a join on two data sources, but one of those data sources doesn't have a matching record.

Solution

Here are the entities to perform a left join with:

```
public class Product
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Type { get; set; }

    public decimal Price { get; set; }

    public string Region { get; set; }
}

public class SalesPerson
```

```
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Address { get; set; }

    public string City { get; set; }

    public string PostalCode { get; set; }

    public string Region { get; set; }

    public string ProductType { get; set; }
}
```

This is the data source:

```
public class InMemoryContext
{
    List<SalesPerson> salesPeople =
        new List<SalesPerson>
        {
            new SalesPerson
            {
                ID = 1,
                Address = "123 1st Street",
                City = "First City",
                Name = "First Person",
                PostalCode = "45678",
                Region = "Region #1",
                ProductType = "Type 2"
            },
            new SalesPerson
            {
                ID = 2,
                Address = "234 2nd Street",
                City = "Second City",
                Name = "Second Person",
                PostalCode = "56789",
                Region = "Region #2",
                ProductType = "Type 3"
            },
            new SalesPerson
            {
                ID = 3,
                Address = "345 3rd Street",
                City = "Third City",
                Name = "Third Person",
                PostalCode = "67890",
                Region = "Region #3",
                ProductType = "Type 1"
            },
            new SalesPerson
            {
                ID = 3,
                Address = "678 9th Street",
                City = "Fourth City",
                Name = "Fourth Person",
                PostalCode = "90123",
                Region = "Region #1",
                ProductType = "Type 2"
            },
        };

    List<Product> products =
        new List<Product>
        {
            new Product
            {
                ID = 1,
                Name = "Product 1",
                Price = 123.45m,
                Type = "Type 2",
                Region = "Region #1",
            },
            new Product
            {
                ID = 2,
                Name = "Product 2",
                Price = 456.78m,
                Type = "Type 2",
                    Region = "Region #2",
            },
            new Product
            {
                ID = 3,
                Name = "Product 3",
                Price = 789.10m,
                Type = "Type 3",
                Region = "Region #1",
            },
            new Product
            {
                ID = 4,
                Name = "Product 4",
                Price = 234.56m,
                Type = "Type 2",
                Region = "Region #1",
            },
        };

    public List<SalesPerson> SalesPeople => salesPeople;

    public List<Product> Products => products;
}
```

The following code performs the left join operation:

```
    class Program
    {
        static void Main()
        {
            var context = new InMemoryContext();

            var salesProducts =
                (from product in context.Products
                    join person in context.SalesPeople on
                    (product.Region, product.Type)
                    equals
                    (person.Region, person.ProductType)
                    into prodPersonTemp
                    from prodPerson in prodPersonTemp.DefaultIfEmpty()
                    select new
                    {
                        Person = prodPerson?.Name ?? "(none)",
                        Product = product.Name,
                        product.Region,
                        product.Type
                    })
                .ToList();

            Console.WriteLine("Sales People\n");

            salesProducts.ForEach(salesProd =>
                Console.WriteLine(
                    $"Person: {salesProd.Person}\n" +
                    $"Product: {salesProd.Product}\n" +
                    $"Region: {salesProd.Region}\n" +
                    $"Type: {salesProd.Type}\n"));
        }
    }
```

Discussion

This solution is similar to the `join`, discussed in Section 4.3. The difference is in the LINQ query in the `Main` method. Notice the `into prodPersonTemp` clause. This is a temporary holder for the joined data. The from clause (below `into`) queries `prodPersonTemp.DefaultIfEmpty()`.

The `DefaultIfEmpty()` performs the left join, where the `prodPerson` range variable receives all of the product objects and only the matching person objects.

Order on the first `from` clause is important here because it specifies the left side of the query. the `join` clause specifies the right side of the query, which might not have matching values.

Notice how the `select` clause checks `prodPerson?.Name` for `null` and replaces it with `(none)`. This ensures the output indicates that there wasn't a match, rather than relying on later code to check for null.

4.4 Grouping Data

Problem

You need to aggregate data into custom groups.

Solution

Here's the entity to group:

```
public class SalesPerson
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Address { get; set; }

    public string City { get; set; }

    public string PostalCode { get; set; }

    public string Region { get; set; }
```

```
        public string ProductType { get; set; }
    }
```

This is the data source:

```
    public class InMemoryContext
    {
        List<SalesPerson> salesPeople =
            new List<SalesPerson>
            {
                new SalesPerson
                {
                    ID = 1,
                    Address = "123 1st Street",
                    City = "First City",
                    Name = "First Person",
                    PostalCode = "45678",
                    Region = "Region #1"
                },
                new SalesPerson
                {
                    ID = 2,
                    Address = "234 2nd Street",
                    City = "Second City",
                    Name = "Second Person",
                    PostalCode = "56789",
                    Region = "Region #2"
                },
                new SalesPerson
                {
                    ID = 3,
                    Address = "345 3rd Street",
                    City = "Third City",
                    Name = "Third Person",
                    PostalCode = "67890",
                    Region = "Region #3"
                },
                new SalesPerson
                {
                    ID = 4,
                    Address = "678 9th Street",
                    City = "Second City",
                    Name = "Fourth Person",
                    PostalCode = "56788",
                    Region = "Region #2"
                },
            };
        public List<SalesPerson> SalesPeople => salesPeople;
    }
```

The following code groups the data:

```
    class Program
    {
        static void Main()
        {
            var context = new InMemoryContext();

            var salesPeopleByRegion =
                (from person in context.SalesPeople
                    group person by person.Region into personGroup
                    select personGroup)
                .ToList();

            Console.WriteLine("Sales People by Region");

            foreach (var region in salesPeopleByRegion)
            {
                Console.WriteLine($"\nRegion: {region.Key}");

                foreach (var person in region)
                    Console.WriteLine($"  {person.Name}");
            }
        }
    }
```

Discussion

Grouping is useful when you need a hierarchy of data. It creates a parent/children relationship between data where the parent is the main category and the children are objects (representing data records) in that category.

In the solution, each `SalesPerson` has a `Region` property, which is duplicated in the `InMemoryContext` data source. This helps see how multiple `SalesPerson` entities can be grouped into a single region.

In the `Main` method query, there's a `group by` clause, specifying the range variable, `person`, to group and the key, `Region`, to group by. The `personGroup` holds the result. In this example, the `select` clause uses the entire `personGroup`, rather than doing a custom projection.

Inside of `salesPeopleByRegion` is a set of top-level objects, representing each group. Each of those groups has a collection of objects belonging to that group, like this:

```
Key (Region):
    Items (IEnumerable<SalesPerson>)
```

The `foreach` loop demonstrates this group structure and how it could be used. At the top level, each object has a `Key` property. Because the original query was by `Region`, that key will have the name of the `Region`.

The nested `foreach` loop iterates on the group, reading each `SalesPerson` instance in that group. You can see where it prints out the `Name` of each `SalesPerson` instance in that group.

4.5 Building Incremental Queries

Problem

You need to customize a query based on a user's search criteria, but don't want to concatenate strings.

Solution

This is the type to query:

```
public class SalesPerson
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Address { get; set; }

    public string City { get; set; }

    public string PostalCode { get; set; }

    public string Region { get; set; }

    public string ProductType { get; set; }
}
```

Here’s the data source:

```
public class InMemoryContext
{
    List<SalesPerson> salesPeople =
        new List<SalesPerson>
        {
            new SalesPerson
            {
                ID = 1,
                Address = "123 1st Street",
                City = "First City",
                Name = "First Person",
                PostalCode = "45678",
                Region = "Region #1",
                ProductType = "Type 2"
            },
            new SalesPerson
            {
                ID = 2,
                Address = "234 2nd Street",
                City = "Second City",
                Name = "Second Person",
                PostalCode = "56789",
                Region = "Region #2",
                ProductType = "Type 3"
            },
            new SalesPerson
            {
                ID = 3,
                Address = "345 3rd Street",
                City = "Third City",
                Name = "Third Person",
                PostalCode = "67890",
                Region = "Region #3",
                ProductType = "Type 1"
            },
            new SalesPerson
            {
                ID = 4,
                Address = "678 9th Street",
                City = "Fourth City",
                Name = "Fourth Person",
                PostalCode = "90123",
                Region = "Region #1",
                ProductType = "Type 2"
            },
        };
        public List<SalesPerson> SalesPeople => salesPeople;
    }
```

This code builds a dynamic query:

```
    class Program
    {
        static void Main()
        {
            SalesPerson searchCriteria = GetCriteriaFromUser();

            List<SalesPerson> salesPeople =
    QuerySalesPeople(searchCriteria);

            PrintResults(salesPeople);
        }

        static SalesPerson GetCriteriaFromUser()
        {
            var person = new SalesPerson();

            Console.WriteLine("Sales Person Search");
            Console.WriteLine("(press Enter to skip an entry)\n");

            Console.Write($"{nameof(SalesPerson.Address)}: ");
            person.Address = Console.ReadLine();

            Console.Write($"{nameof(SalesPerson.City)}: ");
            person.City = Console.ReadLine();

            Console.Write($"{nameof(SalesPerson.Name)}: ");
            person.Name = Console.ReadLine();

            Console.Write($"{nameof(SalesPerson.PostalCode)}: ");
            person.PostalCode = Console.ReadLine();

            Console.Write($"{nameof(SalesPerson.ProductType)}: ");
            person.ProductType = Console.ReadLine();

            Console.Write($"{nameof(SalesPerson.Region)}: ");
            person.Region = Console.ReadLine();

            return person;
        }

        static List<SalesPerson> QuerySalesPeople(SalesPerson criteria)
        {
            var ctx = new InMemoryContext();
            IEnumerable<SalesPerson> salesPeopleQuery =
                from people in ctx.SalesPeople
                select people;

            if (!string.IsNullOrWhiteSpace(criteria.Address))
                salesPeopleQuery = salesPeopleQuery.Where(
                    person => person.Address == criteria.Address);

            if (!string.IsNullOrWhiteSpace(criteria.City))
                salesPeopleQuery = salesPeopleQuery.Where(
                    person => person.City == criteria.City);

            if (!string.IsNullOrWhiteSpace(criteria.Name))
                salesPeopleQuery = salesPeopleQuery.Where(
                    person => person.Name == criteria.Name);

            if (!string.IsNullOrWhiteSpace(criteria.PostalCode))
                salesPeopleQuery = salesPeopleQuery.Where(
                    person => person.PostalCode == criteria.PostalCode);

            if (!string.IsNullOrWhiteSpace(criteria.ProductType))
                salesPeopleQuery = salesPeopleQuery.Where(
                    person => person.ProductType == criteria.ProductType);

            if (!string.IsNullOrWhiteSpace(criteria.Region))
                salesPeopleQuery = salesPeopleQuery.Where(
                    person => person.Region == criteria.Region);

            List<SalesPerson> salesPeople = salesPeopleQuery.ToList();

            return salesPeople;
        }

        static void PrintResults(List<SalesPerson> salesPeople)
        {
            Console.WriteLine("\nSales People\n");

            salesPeople.ForEach(person =>
                Console.WriteLine($"{person.ID}. {person.Name}"));
        }
    }
```

Discussion

One of the worst things a developer can do from a security perspective is to build a concatenated string from user input to send as a SQL statement to a database. The problem is that string concatenation allows the user's input to be interpreted as part of the query. In most cases, people just want to perform a search. However, there are malicious users who intentionally probe systems for this type of vulnerability. They don’t have to be professional hackers as there are plenty of novices (often referred to as script kiddies) who want to practice and have fun. In the worst case, hackers can access private or proprietary information or even take over a machine. Once into one machine on a network, the hacker is on the inside and can monkey bar into other computers and take over your network. This particular problem is called a SQL Injection attack and this section explains how to avoid it.

NOTE

From a security point of view, no computer is theoreticallly 100% secure because there’s always a level of effort, either physical or virtual, where a computer can be broken into. In practicality, security efforts can grow to a point that they become prohibitively expensive because of the cost involved. Your goal is to perform a threat assessment of a system (outside the scope of this book) that’s strong enough to deter potential hackers. In most cases, having not been able to perform the typical attacks, like SQL Injection, a hacker will assess their own costs of attacking your system and move on to a different system that is less time consuming or expensive. This section offers a low cost option to solve a high cost security disaster.

The scenario for this section imagines a situation where the the user can perform a search. They fill in the data and the application dynamically builds a query, based on the criteria the user entered.

In the solution, the `Program` class has a method named `GetCriteriaFromUser`. The purpose of this method is to ask a question for each field inside of `SalesPerson`. This becomes the criteria for building a dynamic query. Any fields left blank aren’t included in the final query.

The `QuerySalesPeople` method starts with a LINQ query for `ctx.SalesPeople`. However, notice that this isn’t in parenthesis or calling the `ToList` operator, like previous sections. Calling `ToList` would have materialized the query, causing it to execute. However, we aren’t doing that here - the code is just building a query. That’s why the `salesPersonQuery` has the `IEnumerable<SalesPerson>` type, indicating that it’s a LINQ to Objects result, rather than a `List<SalesPerson>` we would have gotten back via a call to `ToList`.

Example 4-1.

This recipe takes advantage of a feature of LINQ, known as deferred query execution, which allows you to build the query that won't execute until you tell it to. In addition to facilitating dynamic query construction, deferred execution is also efficient because there's only a single query sent to the database, rather than each time the algorithm calls a specific LINQ operator.

With the `salesPersonQuery` reference, the code checks each `SalesPerson` field for a value. If the user did enter a value for that field, the code uses a `Where` operator to check for equality with what the user entered.

> **NOTE**
>
> You've seen LINQ queries with language syntax in previous sections. However, this section takes advantage of another way to use LINQ, via fluent interface. This is much like the builder pattern you learned about in Section 1.10 Building a Fluid Interface.

So far the only thing that has happened is that we've dynamically built a LINQ query and, because of deferred execution, the query hasn't run yet. Finally, the code calls `ToList` on `salesPersonQuery`, materializing the query. As the return type of this method indicates, this returns a `List<SalesPerson>`.

Now, the algorithm has built and executed a dynamic query, protected from SQL Injection attack. This protection comes from then fact that the LINQ provider parameterized user input so it will be treated as parameter data, rather than as part of the query. As a side benefit, you also have a method with strongly typed code, where you don't have to worry about inadvertent and hard to find typos.

See Also

1.10 Building a Fluid Interface

4.6 Querying Distinct Objects

Problem

You have a list of objects with duplicates and need to transform that to a distinct list of unique objects.

Solution

Here’s an object that won’t support distinct queries:

```
public class SalesPerson
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Address { get; set; }

    public string City { get; set; }

    public string PostalCode { get; set; }

    public string Region { get; set; }

    public string ProductType { get; set; }
}
```

Here’s how to fix that object to support distinct queries:

```
public class SalesPerson : IEqualityComparer<SalesPerson>
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Address { get; set; }

    public string City { get; set; }

    public string PostalCode { get; set; }

    public string Region { get; set; }

    public string ProductType { get; set; }

    public bool Equals(SalesPerson x, SalesPerson y)
    {
        return x.ID == y.ID;
    }
```

```
        public int GetHashCode(SalesPerson obj)
        {
            return ID.GetHashCode();
        }
    }
```

Here’s the data source:

```
    public class InMemoryContext
    {
        List<SalesPerson> salesPeople =
            new List<SalesPerson>
            {
                new SalesPerson
                {
                    ID = 1,
                    Address = "123 1st Street",
                    City = "First City",
                    Name = "First Person",
                    PostalCode = "45678",
                    Region = "Region #1",
                    ProductType = "Type 2"
                },
                new SalesPerson
                {
                    ID = 2,
                    Address = "234 2nd Street",
                    City = "Second City",
                    Name = "Second Person",
                    PostalCode = "56789",
                    Region = "Region #2",
                    ProductType = "Type 3"
                },
                new SalesPerson
                {
                    ID = 3,
                    Address = "345 3rd Street",
                    City = "Third City",
                    Name = "Third Person",
                    PostalCode = "67890",
                    Region = "Region #3",
                    ProductType = "Type 1"
                },
                new SalesPerson
                {
                    ID = 4,
                    Address = "678 9th Street",
                    City = "Fourth City",
                    Name = "Fourth Person",
                    PostalCode = "90123",
                    Region = "Region #1",
                    ProductType = "Type 2"
                },
                new SalesPerson
                {
                    ID = 4,
                    Address = "678 9th Street",
                    City = "Fourth City",
                    Name = "Fourth Person",
                    PostalCode = "90123",
                    Region = "Region #1",
                    ProductType = "Type 2"
                },
            };

        public List<SalesPerson> SalesPeople => salesPeople;
    }
```

This code filters by distinct objects:

```
class Program
{
    static void Main(string[] args)
    {
        var salesPeopleWithoutComparer =
            (from person in new InMemoryContext().SalesPeople
                select person)
            .Distinct()
            .ToList();

        PrintResults(salesPeopleWithoutComparer, "Without Comparer");

        var salesPeopleWithComparer =
            (from person in new InMemoryContext().SalesPeople
                select person)
            .Distinct(new SalesPerson())
            .ToList();

        PrintResults(salesPeopleWithComparer, "With Comparer");
    }

    static void PrintResults(List<SalesPerson> salesPeople, string
title)
    {
        Console.WriteLine($"\n{title}\n");
```

```
            salesPeople.ForEach(person =>
                Console.WriteLine($"{person.ID}. {person.Name}"));
        }
    }
```

Discussion

Sometimes you have a list of entities with duplicates, either because of some application processing or the type of database query that results in duplicates. Often, you need a list of unique objects. e.g. you're materializing into a `Dictionary` collection that doesn't allow duplicates.

The LINQ Distinct operator helps get a list of unique objects. At first glance, this is easy as shown in the first query of the `Main` method that uses the `Distinct()` operator. Notice that it doesn't have parameters. However, an inspection of the results shows that you still have the same duplicates in the data that you started with.

The problem, and subsequent solution, might not be immediately obvious because it relies on combining a few different C# concepts. First, think about how `Distinct` should be able to tell the difference between objects - it has to perform a comparison. Next, consider that the type of `SalesPerson` is class. That's important because classes are reference types, which have reference equality. When `Distinct` does a reference comparison, no two object references (which refer to an object in memory) are the same because each object has a unique reference. Finally, you need to write code to compare `SalesPerson` instances to see if they're equal and tell `Distinct` about that code.

There are two implementations of `SalesPerson` in the solution. The first is a normal class and the second implements `IEqualityComparer<SalesPerson>`. The normal class doesn't work because it has reference equality. However the `SalesPerson` class that implements `IEqualityComparer<SalesPerson>` compares properly because it has an `Equals` method. In this case, checking `ID` is sufficient to determine that instances are equal, assuming that each entity comes from the same data source with unique `ID` fields.

Now, `SalesPerson` knows how to compare `SalesPerson` instances, but that isn't the end of the story. If you ran the first query in `Main` with `Distinct()` (no parameter), the results will still have duplicates. The problem is that `Distinct` doesn't know how to compare the objects so it defaults to the instance type, `class`, which, as explained earlier, is a reference type.

The solution is to use the second query in `Main` that uses the call to `Distinct(new SalesPerson())` (with parameter). This uses the `Distinct` operator's overload with the `IEqualityComparer<T>` overload. Since `SalesPerson` implements `IEqualityComparer<SalesPerson>` this works.

See Also

2.5 Checking for Type Equality

4.7 Simplifying Queries

Problem

A query has become too complex and you need to make it more readable.

Solution

Here's the entity to query:

```
public class SalesPerson
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Address { get; set; }

    public string City { get; set; }

    public string PostalCode { get; set; }

    public string Region { get; set; }
```

```
        public string ProductType { get; set; }

        public string TotalSales { get; set; }
    }
```

This is the data source:

```
    public class InMemoryContext
    {
        List<SalesPerson> salesPeople =
            new List<SalesPerson>
            {
                new SalesPerson
                {
                    ID = 1,
                    Address = "123 1st Street",
                    City = "First City",
                    Name = "First Person",
                    PostalCode = "45678",
                    Region = "Region #1",
                    ProductType = "Type 2",
                    TotalSales = "654.32"
                },
                new SalesPerson
                {
                    ID = 2,
                    Address = "234 2nd Street",
                    City = "Second City",
                    Name = "Second Person",
                    PostalCode = "56789",
                    Region = "Region #2",
                    ProductType = "Type 3",
                    TotalSales = "765.43"
                },
                new SalesPerson
                {
                    ID = 3,
                    Address = "345 3rd Street",
                    City = "Third City",
                    Name = "Third Person",
                    PostalCode = "67890",
                    Region = "Region #3",
                    ProductType = "Type 1",
                    TotalSales = "876.54"
                },
                new SalesPerson
                {
                    ID = 4,
                    Address = "678 9th Street",
                    City = "Fourth City",
                    Name = "Fourth Person",
                    PostalCode = "90123",
                    Region = "Region #1",
                    ProductType = "Type 2",
                    TotalSales = "987.65"
                },
                new SalesPerson
                {
                    ID = 4,
                    Address = "678 9th Street",
                    City = "Fourth City",
                    Name = "Fourth Person",
                    PostalCode = "90123",
                    Region = "Region #1",
                    ProductType = "Type 2",
                    TotalSales = "109.87"
                },
            };

        public List<SalesPerson> SalesPeople => salesPeople;
    }
```

The following shows how to simplify a query projection:

```
class Program
{
    static void Main(string[] args)
    {
        decimal TotalSales = 0;

        var salesPeopleWithAddresses =
            (from person in new InMemoryContext().SalesPeople
                let FullAddress =
                $"{person.Address}\n" +
                $"{person.City}, {person.PostalCode}"
                let salesOkay =
                    decimal.TryParse(person.TotalSales, out
TotalSales)
                select new
                {
                    person.ID,
                    person.Name,
                    FullAddress,
                    TotalSales
                })
            .ToList();
        Console.WriteLine($"\nSales People and Addresses\n");

        salesPeopleWithAddresses.ForEach(person =>
            Console.WriteLine(
                $"{person.ID}. {person.Name}: {person.TotalSales:C}\n"
+
                $"{person.FullAddress}\n"));
    }
}
```

Discussion

Sometimes LINQ queries get complex. If the code is still hard to read, it's also hard to maintain. One option is to go imperative and re-write the query as a loop. Another is to use the `let` clause for simplification.

In the solution, the `Main` method has a query with a custom projection into an anonymous type. Sometimes queries are complex because they have sub-queries, or other logic, inside of the projection. For example, look at `FullAddress`, being built in a `let` clause. Without that simplification, the code would have ended up inside the projection.

Another scenario you might face is when parsing object input from string. The example uses a `TryParse` in a `let` clause, which is impossible to put in the projection. This is a little tricky because the `out` parameter, `TotalSales`, is outside of the query. We ignore the results of `TryParse`, but can now assign `TotalSales` in the projection.

4.8 Operating on Sets

Problem

You want to combine two sets of objects without duplication.

Solution

Here's the entity to query:

```
    public class SalesPerson : IEqualityComparer<SalesPerson>
```

```
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Address { get; set; }

    public string City { get; set; }

    public string PostalCode { get; set; }

    public string Region { get; set; }

    public string ProductType { get; set; }

    public bool Equals(SalesPerson x, SalesPerson y)
    {
        return x.ID == y.ID;
    }

    public int GetHashCode(SalesPerson obj)
    {
        return ID.GetHashCode();
    }
}
```

Here’s the data source:

```
public class InMemoryContext
{
    List<SalesPerson> salesPeople =
        new List<SalesPerson>
        {
            new SalesPerson
            {
                ID = 1,
                Address = "123 1st Street",
                City = "First City",
                Name = "First Person",
                PostalCode = "45678",
                Region = "Region #1",
                ProductType = "Type 2"
            },
            new SalesPerson
            {
                ID = 2,
                Address = "234 2nd Street",
                City = "Second City",
                Name = "Second Person",
                PostalCode = "56789",
                Region = "Region #2",
                ProductType = "Type 3"
            },
            new SalesPerson
            {
                ID = 3,
                Address = "345 3rd Street",
                City = "Third City",
                Name = "Third Person",
                PostalCode = "67890",
                Region = "Region #3",
                ProductType = "Type 1"
            },
            new SalesPerson
            {
                ID = 4,
                Address = "678 9th Street",
                City = "Fourth City",
                Name = "Fourth Person",
                PostalCode = "90123",
                Region = "Region #1",
                ProductType = "Type 2"
            },
        };

    public List<SalesPerson> SalesPeople => salesPeople;
}
```

This code shows how to perform set operations:

```
    class Program
    {
        static InMemoryContext ctx = new InMemoryContext();

        static void Main()
        {
            System.Console.WriteLine("\nLINQ Set Operations");

            DoUnion();
            DoExcept();
            DoIntersection();

            System.Console.WriteLine("\nComplete.\n");
        }

        static void DoUnion()
```

```
{
    var dataSource1 =
        (from person in ctx.SalesPeople
            where person.ID < 3
            select person)
        .ToList();

    var dataSource2 =
        (from person in ctx.SalesPeople
            where person.ID > 2
            select person)
        .ToList();

    List<SalesPerson> union =
        dataSource1
            .Union(dataSource2, new SalesPerson())
            .ToList();

    PrintResults(union, "Union Results");
}

static void DoExcept()
{
    var dataSource1 =
        (from person in ctx.SalesPeople
            select person)
        .ToList();

    var dataSource2 =
        (from person in ctx.SalesPeople
            where person.ID == 4
            select person)
        .ToList();

    List<SalesPerson> union =
        dataSource1
            .Except(dataSource2, new SalesPerson())
            .ToList();

    PrintResults(union, "Except Results");
}

static void DoIntersection()
{
    var dataSource1 =
        (from person in ctx.SalesPeople
            where person.ID < 4
            select person)
        .ToList();
```

```
            var dataSource2 =
                (from person in ctx.SalesPeople
                    where person.ID > 2
                    select person)
                .ToList();

            List<SalesPerson> union =
                dataSource1
                    .Intersect(dataSource2, new SalesPerson())
                    .ToList();

            PrintResults(union, "Intersect Results");
        }

        static void PrintResults(List<SalesPerson> salesPeople, string
    title)
        {
            Console.WriteLine($"\n{title}\n");

            salesPeople.ForEach(person =>
                Console.WriteLine($"{person.ID}. {person.Name}"));
        }
    }
```

Discussion

In Section 4.2, we discussed the concept of joining data from two separate data sources. The examples operate in that same spirit and show different manipulations, based on sets.

The first method, `DoUnion`, gets two sets of data, intentionally filtering by `ID` to ensure overlap. From the reference of the first data source, the code calls the `Union` operator with the second data source as the parameter. This results in a set of data from both data sources, including duplicates.

The `DoExcept` method is similar to `DoUnion`, but uses the `Except` operator. This results in a set of all the objects in the first data source. However, any objects in the second data source, even if they were in the first, won’t appear in the results.

Finally, `DoIntersect` is similar in structure to `DoUnion` and `DoExcept`. However, it queries objects that are only in both data sources. If any object is in one data source, but not the other, it won’t appear in the result.

LINQ has many standard operators that, just like the set operators, are very powerful. Before performing any complex operation in a LINQ query, it’s good practice to review standard operators to see if there exists something that will simplify your task.

See Also

4.2 Joining Data 4.3 Performing Left Joins

4.9 Building a Query Filter with Expression Trees

Problem

The LINQ `where` clause combines via `AND` conditions, but you need a `where` that works as an `OR` condition.

Solution

Here’s the entity to query:

```
public class SalesPerson
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Address { get; set; }

    public string City { get; set; }

    public string PostalCode { get; set; }

    public string Region { get; set; }

    public string ProductType { get; set; }
}
```

This is the data source:

```
public class InMemoryContext
{
    List<SalesPerson> salesPeople =
        new List<SalesPerson>
        {
            new SalesPerson
            {
                ID = 1,
                Address = "123 1st Street",
                City = "First City",
                Name = "First Person",
                PostalCode = "45678",
                Region = "Region #1",
                ProductType = "Type 2"
            },
            new SalesPerson
            {
                ID = 2,
                Address = "234 2nd Street",
                City = "Second City",
                Name = "Second Person",
                PostalCode = "56789",
                Region = "Region #2",
                ProductType = "Type 3"
            },
            new SalesPerson
            {
                ID = 3,
                Address = "345 3rd Street",
                City = "Third City",
                Name = "Third Person",
                PostalCode = "67890",
                Region = "Region #3",
                ProductType = "Type 1"
            },
            new SalesPerson
            {
                ID = 4,
                Address = "678 9th Street",
                City = "Fourth City",
                Name = "Fourth Person",
                PostalCode = "90123",
                Region = "Region #1",
                ProductType = "Type 2"
            },
        };

    public List<SalesPerson> SalesPeople => salesPeople;
}
```

Here’s an extension method for a filtered `OR` operation:

```
public static class CookbookExtensions
{
    public static IEnumerable<TParameter> WhereOr<TParameter>(
        this IEnumerable<TParameter> query,
        Dictionary<string, string> criteria)
    {
        const string ParamName = "person";

        ParameterExpression paramExpr =
            Expression.Parameter(typeof(TParameter), ParamName);

        Expression accumulatorExpr = null;

        foreach (var criterion in criteria)
        {
            MemberExpression paramMbr =
                LambdaExpression.PropertyOrField(
                    paramExpr, criterion.Key);

            MemberExpression leftExpr =
                Expression.Property(
                    paramExpr,
                    typeof(TParameter).GetProperty(criterion.Key));
            Expression rightExpr =
                Expression.Constant(criterion.Value, typeof(string));
            Expression equalExpr =
                Expression.Equal(leftExpr, rightExpr);

            accumulatorExpr = accumulatorExpr == null
                ? equalExpr
                : Expression.Or(accumulatorExpr, equalExpr);
        }

        Expression<Func<TParameter, bool>> allClauses =
            Expression.Lambda<Func<TParameter, bool>>(
                accumulatorExpr, paramExpr);

        Func<TParameter, bool> compiledClause = allClauses.Compile();

        return query.Where(compiledClause);
    }
}
```

Here’s the code that consumes the new extension method:

```
class Program
{
    static void Main()
    {
        SalesPerson searchCriteria = GetCriteriaFromUser();

        List<SalesPerson> salesPeople =
QuerySalesPeople(searchCriteria);

        PrintResults(salesPeople);
    }

    static SalesPerson GetCriteriaFromUser()
    {
        var person = new SalesPerson();

        Console.WriteLine("Sales Person Search");
        Console.WriteLine("(press Enter to skip an entry)\n");

        Console.Write($"{nameof(SalesPerson.Address)}: ");
        person.Address = Console.ReadLine();

        Console.Write($"{nameof(SalesPerson.City)}: ");
        person.City = Console.ReadLine();

        Console.Write($"{nameof(SalesPerson.Name)}: ");
        person.Name = Console.ReadLine();

        Console.Write($"{nameof(SalesPerson.PostalCode)}: ");
        person.PostalCode = Console.ReadLine();

        Console.Write($"{nameof(SalesPerson.ProductType)}: ");
        person.ProductType = Console.ReadLine();

        Console.Write($"{nameof(SalesPerson.Region)}: ");
        person.Region = Console.ReadLine();

        return person;
    }

    static List<SalesPerson> QuerySalesPeople(SalesPerson criteria)
    {
        var ctx = new InMemoryContext();

        var filters = new Dictionary<string, string>();

        IEnumerable<SalesPerson> salesPeopleQuery =
            from people in ctx.SalesPeople
            select people;

        if (!string.IsNullOrWhiteSpace(criteria.Address))
            filters[nameof(criteria.Address)] = criteria.Address;
        if (!string.IsNullOrWhiteSpace(criteria.City))
            filters[nameof(criteria.City)] = criteria.City;

        if (!string.IsNullOrWhiteSpace(criteria.Name))
            filters[nameof(criteria.Name)] = criteria.Name;

        if (!string.IsNullOrWhiteSpace(criteria.PostalCode))
            filters[nameof(criteria.PostalCode)] =
criteria.PostalCode;

        if (!string.IsNullOrWhiteSpace(criteria.ProductType))
            filters[nameof(criteria.ProductType)] =
criteria.ProductType;

        if (!string.IsNullOrWhiteSpace(criteria.Region))
            filters[nameof(criteria.Region)] = criteria.Region;

        salesPeopleQuery =
            salesPeopleQuery.WhereOr<SalesPerson>(filters);

        List<SalesPerson> salesPeople = salesPeopleQuery.ToList();

        return salesPeople;
    }

    static void PrintResults(List<SalesPerson> salesPeople)
    {
        Console.WriteLine("\nSales People\n");

        salesPeople.ForEach(person =>
            Console.WriteLine($"{person.ID}. {person.Name}"));
    }
}
```

Discussion

Section 4.5 showed the power of dynamic queries in LINQ. However, that isn’t the end of what you can do. With Expression Trees, you can make LINQ to any type of query. If the standard operators don’t provide something you need, you can use Expression Trees. This section does just that, showing how to use Expression Trees to run a dynamic `WhereOr` operation.

The motivation for a `WhereOr` comes from the fact that the standard `Where` operator combines in an `AND` comparison. In Section 4.5, all of those `Where` operators had an `AND` relationship. This means that a given entity must have a value equal to each of the fields (that the user specified in the criteria) to get a match. With the `WhereOr` in this section, all of the fields have an `OR` relationship and a match on only one of the fields is neccessary for inclusion in results.

In the solution, the `GetCriteriaFromUser` method gets the values for each `SalesPerson` property. `QuerySalesPeople` starts a query for deferred execution, as explained in Section 4.5, and builds a `Dictionary<string, string>` of filters.

The `CookbookExtensions` class has the `WhereOr` extension method that accepts the filters. The high level description of what `WhereOr` is trying to accomplish comes from the fact that it needs to return an `IEnumerable<SalesPerson>` for the caller to complete a LINQ query.

First, go to the bottom of `WhereOr` and notice that it returns the query with the `Where` operator and has a parameter named `compiledQuery`. Remember that the LINQ `Where` operator takes a C# lambda expression with a parameter and a predicate. We want a filter that returns an object if any one field of an object matches, based on the input criteria. Therefore, `compiledQuery` must evaluate to a lambda of the following form:

```
person => person.Field1 == "val1" || ... || person.FieldN == "valN"
```

That’s a lambda with `OR` operators for each value in the `Dictionary<string, string> criteria` parameter. To get from the top of this algorithm to the bottom, we need to build an expression tree that evaluates to this form of lambda.

The first thing `WhereOr` does is create a `ParameterExpression`. This is the `person` parameter in the lambda. It’s the parameter to every comparison expression because is represents the `TParameter`, which is an instance of `SalesPerson` in this example.

NOTE

This example called the `ParameterExpression` `person`. However, if this is a generic reusable extension method, you might give it a more general name, like `parameterTerm` because `TParameter` could be any type. The choice of `person` in this example is there to clarify that the `ParameterExpression` represents a `SalesPerson` instance in this example.

The `Expression` `accumulatorExpr`, as it’s name suggests, gathers all of the clauses for the lambda body.

The `foreach` statement loops through the `Dictionary` collection, which returns `KeyValuePair` instances, which have `Key` and `Value` properties. As shown in the `QuerySalesPeople` method, the `Key` property is the name of the `SalesPerson` property and the `Value` property is what the user entered.

For each clause of the lambda, the left hand side is a reference to the property on the `SalesPerson` instance. e.g. `person.Name`. To create that, the code instantiates the `paramMbr` using the `paramExpr` (which is `person`). That becomes a parameter of `leftExpr`.

The `rightExpr` expression is a constant that holds the value to compare and its type.

Then we need to complete the expression with an `Equals` expression for the left and right expressions, `leftExpr` and `rightExpr`, respectively.

Finally, we need to `OR` that expression with any others. The first time through the `foreach` loop `accumulatorExpr` will be `null`, so we just assign the first expression. On subsequent expressions, we use an `OR` expression to append the new `Equals` expression to `accumulatorExpr`.

After iterating through each input field, we form the final `LambdaExpression` that adds the parameter that was used in the left side of each `Equals` expression. Notice that the result is an `Expression<Func<TParameter, bool>>` which has a parameter type matching the lambda delegate type for the original query, which is `Func<SalesPersion, bool>`.

Now, we have an expression tree that was build dynamically. However, we need to convert that expression tree into runnable code, which is a task for the `Expression.Compile` method. This gives us a compiled lambda that we can assign to the `Where` clause.

The calling code receives the `IEnumerable<SalesPerson>` from the `WhereOr` method and materializes the query with a call to `ToList`. This produces a list of any `SalesPerson` objects that match at least one of the user's specified criteria.

See Also

4.5 Building Incremental Queries

4.10 Querying in Parallel

Problem

You want to improve performance and your query could benefit from multi-threading.

Solution

Here's the entity to query:

```
public class SalesPerson
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string Address { get; set; }

    public string City { get; set; }

    public string PostalCode { get; set; }

    public string Region { get; set; }

    public string ProductType { get; set; }
}
```

This is the data source:

```
public class InMemoryContext
{
    List<SalesPerson> salesPeople =
        new List<SalesPerson>
        {
```

```
new SalesPerson
{
    ID = 1,
    Address = "123 1st Street",
    City = "First City",
    Name = "First Person",
    PostalCode = "45678",
    Region = "Region #1",
    ProductType = "Type 2"
},
new SalesPerson
{
    ID = 2,
    Address = "234 2nd Street",
    City = "Second City",
    Name = "Second Person",
    PostalCode = "56789",
    Region = "Region #2",
    ProductType = "Type 3"
},
new SalesPerson
{
    ID = 3,
    Address = "345 3rd Street",
    City = "Third City",
    Name = "Third Person",
    PostalCode = "67890",
    Region = "Region #3",
    ProductType = "Type 1"
},
new SalesPerson
{
    ID = 4,
    Address = "678 9th Street",
    City = "Fourth City",
    Name = "Fourth Person",
    PostalCode = "90123",
    Region = "Region #1",
    ProductType = "Type 2"
},
new SalesPerson
{
    ID = 5,
    Address = "678 9th Street",
    City = "Fifth City",
    Name = "Fifth Person",
    PostalCode = "90123",
    Region = "Region #1",
    ProductType = "Type 2"
},
        };

    public List<SalesPerson> SalesPeople => salesPeople;
}
```

This code shows how to perform a parallel query:

```
class Program
{
    static void Main()
    {
        List<SalesPerson> salesPeople = new
InMemoryContext().SalesPeople;
        var result =
            (from person in salesPeople.AsParallel()
                select ProcessPerson(person))
            .ToList();
    }

    static SalesPerson ProcessPerson(SalesPerson person)
    {
        Console.WriteLine(
            $"Starting sales person " +
            $"#{person.ID}. {person.Name}");

        // complex in-memory processing
        Thread.Sleep(500);

        Console.WriteLine(
            $"Completed sales person " +
            $"#{person.ID}. {person.Name}");

        return person;
    }
}
```

Discussion

This section considers queries that can benefit from concurrency. Imagine you have a LINQ to Objects query, where the data is in memory. Perhaps work on each instance requires intensive processing, the code runs on a multi-threaded/multi-core CPU, and/or takes a non-trivial amount of time. Running the query in parallel might be an option.

The `Main` method performs a query, similar to any other query, except for the

`AsParallel` operator on the data source. What this does is let LINQ figure out how to split up the work and operate on each range variable in parallel.

This example also demonstrates another type of projection that uses a method to return an object. The assumption here is that the intensive processing occurs in `ProcessPerson`, which has a `Thread.Sleep` to indicate non-trivial processing.

In practice, you would want to do some testing to see if you’re really benefitting from parallelism. Section 3.10 shows how to measure performance with the `System.Diagnostics.StopWatch` class. If successful, this could be an easy way to boost performance. 3.10 Measuring Performance

Chapter 5. Implementing Dynamic and Reflection

Reflection allows code to look inside of a type and examine it’s details and members. This is useful for libraries and tools that want to give the user maximum flexibility to submit objects to perform some automatic operation. A common example of code that does reflection are unit testing frameworks. As described in Section 3.1, unit tests take classes whose members have attributes to indicate which methods are tests. The unit testing framework uses reflection to find classes that are tests, locate the test methods, and execute the tests.

The example in this chapter is based on a dynamic report building application. It uses reflection to read attributes of a class, accesses type members, and executes methods. The first 4 sections of this chapter show how to do that.

In addition to reflection, another way to work flexibly with code is via dynamic code. In C#, much of the code we write is strongly typed and that’s a huge benefit for productivity and maintainability. That said, C# has a `dynamic` keyword that allows developers to assume that objects have a certain structure. This is much like dynamic programming langages, like JavaScript and Python, where developers access objects based on documentation that specifies what members an object has. So, they just write code that uses those members. Dynamic code allows C# to do the same thing.

When performing operations requiring COM Interop, dynamic is particularly useful and there’s a section explaining how that works. You’ll see how dynamic can be useful in significantly reducing and simplifying the code, as compared to the verbosity and complexity of reflection. There are also types that allow us to build an inherently dynamic type. Additionally, there’s a Dynamic Language

Runtime (DLR) that enables interop between C# and dynamic languages, such as Python, and you’ll see two sections on interoperability between C# and Python.

5.1 Reading Attributes with Reflection

Problem

You want consumers of your library to have maximum flexibility when passing objects, but they still need to communicate important details of the object.

Solution

Here’s an `Attribute` class, representing report column metadata:

```
namespace Section_05_01
{
    [AttributeUsage(
        AttributeTargets.Property | AttributeTargets.Method,
        AllowMultiple = false)]
    public class ColumnAttribute : Attribute
    {
        public ColumnAttribute(string name)
        {
            Name = name;
        }

        public string Name { get; set; }

        public string Format { get; set; }
    }
}
```

This class represents a record to display and uses the attribute:

```
public class InventoryItem
{
    [Column("Part #")]
    public string PartNumber { get; set; }

    [Column("Name")]
    public string Description { get; set; }

    [Column("Amount")]
```

```
        public int Count { get; set; }

        [Column("Price")]
        public decimal ItemPrice { get; set; }

        [Column("Total")]
        public decimal CalculateTotal()
        {
            return ItemPrice * Count;
        }
    }
```

The `Main` method shows how to instantiate and pass the data:

```
    static void Main()
    {
        var inventory = new List<object>
        {
            new InventoryItem
            {
                PartNumber = "1",
                Description = "Part #1",
                Count = 3,
                ItemPrice = 5.26m
            },
            new InventoryItem
            {
                PartNumber = "2",
                Description = "Part #2",
                Count = 1,
                ItemPrice = 7.95m
            },
            new InventoryItem
            {
                PartNumber = "3",
                Description = "Part #3",
                Count = 2,
                ItemPrice = 23.13m
            },
        };

        string report = new Report().Generate(inventory);

        Console.WriteLine(report);
    }
```

This method uses reflection to find all type members:

```
public string Generate(List<object> items)
{
    _ = items ??
        throw new ArgumentNullException(
            $"{items} is required");

    MemberInfo[] members =
        items.First().GetType().GetMembers();

    var report = new StringBuilder("# Report\n\n");

    report.Append(GetHeaders(members));

    return report.ToString();
}
```

This method uses reflection to read attributes of a type:

```
const string ColumnSeparator = " | ";

StringBuilder GetHeaders(MemberInfo[] members)
{
    var columnNames = new List<string>();
    var underscores = new List<string>();

    foreach (var member in members)
    {
        var attribute =
            member.GetCustomAttribute<ColumnAttribute>();

        if (attribute != null)
        {
            string columnTitle = attribute.Name;
            string dashes = "".PadLeft(columnTitle.Length, '-');

            columnNames.Add(columnTitle);
            underscores.Add(dashes);
        }
    }

    var header = new StringBuilder();

    header.AppendJoin(ColumnSeparator, columnNames);
    header.Append("\n");

    header.AppendJoin(ColumnSeparator, underscores);
    header.Append("\n");
```

```
        return header;
    }
```

And here’s the output:

```
# Report

Total | Part # | Name | Amount | Price
----- | ------ | ---- | ------ | -----
```

Discussion

Attributes, which are metadata, typically exist to support tooling on code. The solution in this section takex a similar approach where the `ColumnAttribute` is metadata for a column of data in a report. You can see where the `AttributeUsage` specifies that you can apply `ColumnAttribute` to either properties or methods. Thinking of how many features that a report column might be able to support, this attribute boils down to two typical features: `Name` and `Format`. Because a C# property name might not represent the text of a column header, `Name` lets you specify anything you want. Also, without specifying a string format, `DateTime` and `decimal` columns would take default displays, which is often not what you want. This essentially solves the problem where a consumer of a report library wants to pass any type of object they want, using `ColumnAttribute` to share important details.

`InventoryItem` shows how `ColumnAttribute` works. Notice how the positional property, `Name`, differs from the name of the properties and method. Section 5.2 has an example of how the `Format` property works, while this section only concentrates on how to extract and display the metadata as a Markdown formatted column.

NOTE

Architecurally, you should look at this probject as two separate applications. There’s a reusable report library that anyone can submit objects to. The report library consists of a `Report` class and the `ColumnAttribute` attribute. Then there’s a consumer application, which is the `Main` method. For simplicity, the source code for this demo puts all the code into the same project, but in practice, these would be separate.

The `Main` method instantiates a `List<object>` that contains `InventoryItem` instances. This is data that would typically come from a database or other data source. It instantiates the `Report`, passes the data, and prints the result.

The `Generate` method belongs to the `Report` class. Notice that it accepts a `List<object>`, which is why `Main` passed a `List<object>`. Essentially, `Report` wants to be able to operate on any object type, making it reusable.

After validating input, items, `Generate` uses reflection to discover what members exist in the objects passed. You see, we’re no longer able to know because the objects aren’t strongly typed and we want maximum flexibility in what types can be passed. This is a good case for reflection. That said, we no longer have the guarantee that all instances in items are the same type and that has to be an implied contract, rather than enforced by code. Section 5.3 fixes this by showing how to use generics so we have both type safety and the ability to use generics.

We’re assuming all objects are the same and `Generate` calls `First` on `items`, because it has the exact same attributes of all objects in `items`. `Generate` then calls `GetType` on the first item. The `Type` instance is the gateway for performing reflection.

After getting the `Type` instance, you can ask for anything about a type and work with particular instances. This example calls `GetMembers` to get a `MemberInfo[]`. A `MemberInfo` has all the information about a particular type member, like it’s name and type. In this example, the `MemberInfo[]` contains the properties and methods from the `InventoryItem` that `Main` passed in: `PartNumber`, `Description`, `Count`, `ItemPrice`, and `CalculateTotal`.

Because the report is a string of Markdown text and there is a lot of concatenation, the solution uses `StringBuilder`. Section 2.1, Processing Strings Efficiently, explains why this is a good approach.

Because we’re concerned with attributes, this solution only prints the report header and later sections in this chapter explain a lot of different ways to generate the report body, depending on your needs. The `GetHeader` method takes the `MemberInfo[]` and uses reflection to learn what those header titles should be.

In Markdown, we separate table headers with pipes, `|`, and add an underscore, which is why we have two arrays for `columnNames` and `underscores`. The `foreach` loop examines each `MemberInfo`, calling `GetCustomAttribute`. Notice that the type parameter for `GetCustomAttribute` is `ColumnAttribute` - members can have multiple attributes, but we only want that one. The instance returned from `GetCustomAttribute` is `ColumnAttribute`, so we have access to it’s properties, such as `Name`. The code populates `columnNames` with `Name` and adds an underscore that is the same length as `Name`.

Finally, `GetHeaders` concatenates values with pipes, `|`, and returns the resulting header. Following this back through the call chain. `Generate` appends the `GetHeaders` results and `Main` prints the header, which you can see in the solution output.

See Also

Section 2.1 Section 5.2

5.2 Accessing Type Members with Reflection

Problem

You need to examine an object to see what properties you can read.

Solution

This class represents a record to display:

```
public class InventoryItem
{
    [Column("Part #")]
    public string PartNumber { get; set; }

    [Column("Name")]
    public string Description { get; set; }
```

```
        [Column("Amount")]
        public int Count { get; set; }

        [Column("Price", Format = "{0:c}")]
        public decimal ItemPrice { get; set; }
    }
```

Here’s a class that contains metadata for each report column:

```
    public class ColumnDetail
    {
        public string Name { get; set; }

        public ColumnAttribute Attribute { get; set; }

        public PropertyInfo PropertyInfo { get; set; }
    }
```

This method collects the data to populate column metadata:

```
    Dictionary<string, ColumnDetail> GetColumnDetails(
        List<object> items)
    {
         object itemInstance = items.First();
         Type itemType = itemInstance.GetType();
         PropertyInfo[] itemProperties = itemType.GetProperties();

        return
            (from prop in itemProperties
             let attribute = prop.GetCustomAttribute<ColumnAttribute>()
             where attribute != null
             select new ColumnDetail
             {
                 Name = prop.Name,
                 Attribute = attribute,
                 PropertyInfo = prop
             })
            .ToDictionary(
               key => key.Name,
               val => val);
    }
```

Here’s a more streamlined way to get header data with LINQ:

```
    StringBuilder GetHeaders(
        Dictionary<string, ColumnDetail> details)
```

```
{
    var header = new StringBuilder();

    header.AppendJoin(
        ColumnSeparator,
        from detail in details.Values
        select detail.Attribute.Name);

    header.Append("\n");

    header.AppendJoin(
        ColumnSeparator,
        from detail in details.Values
        let length = detail.Attribute.Name.Length
        select "".PadLeft(length, '-'));

    header.Append("\n");

    return header;
}
```

This method uses reflection to pull the value out of an object property:

```
(object, Type) GetReflectedResult(
    object item, PropertyInfo property)
{
    object result = property.GetValue(item);
    Type type = property.PropertyType;

    return (result, type);
}
```

This method uses reflection to retrieve and format property data:

```
List<string> GetColumns(
    IEnumerable<ColumnDetail> details,
    object item)
{
    var columns = new List<string>();

    foreach (var detail in details)
    {
        PropertyInfo member = detail.PropertyInfo;
        string format =
            string.IsNullOrWhiteSpace(
                detail.Attribute.Format) ?
                "{0}" :
```

```
                detail.Attribute.Format;

        (object result, Type columnType) =
            GetReflectedResult(item, member);

        switch (columnType.Name)
        {
            case "Decimal":
                columns.Add(
                    string.Format(format, (decimal)result));
                break;
            case "Int32":
                columns.Add(
                    string.Format(format, (int)result));
                break;
            case "String":
                columns.Add(
                    string.Format(format, (string)result));
                break;
            default:
                break;
        }
    }

    return columns;
}
```

This method combines and formats all rows of data:

```
StringBuilder GetRows(
    List<object> items,
    Dictionary<string, ColumnDetail> details)
{
    var rows = new StringBuilder();

    foreach (var item in items)
    {
        List<string> columns =
            GetColumns(details.Values, item);

        rows.AppendJoin(ColumnSeparator, columns);

        rows.Append("\n");
    }

    return rows;
}
```

Finally, this method uses all of the others to build a complete report:

```
const string ColumnSeparator = " | ";

public string Generate(List<object> items)
{
    var report = new StringBuilder("# Report\n\n");

    Dictionary<string, ColumnDetail> columnDetails =
        GetColumnDetails(items);
    report.Append(GetHeaders(columnDetails));
    report.Append(GetRows(items, columnDetails));

    return report.ToString();
}
```

And here's the output:

```
|| Total | Part # | Name | Amount | Price ||
| $15.78 | 1 | Part #1 | 3 | 5.26 |
| $7.95 | 2 | Part #2 | 1 | 7.95 |
| $46.26 | 3 | Part #3 | 2 | 23.13 |
```

Discussion

The report library in the solution receives a `List<object>` so that consumers can send any type they want. Since the input objects aren't strongly typed, the `Report` class needs to perform reflection to extract data from each object. Section 5.1 explained how the `Main` method passes this data and how the solution generates the header. This section concentrates on data and the solution doesn't repeat the exact code from Section 5.1.

The `InventoryItem` class uses `ColumnAttribute` attributes. Notice that `ItemPrice` now has the named property, `Format`, specifying that this column should be formatted in the report as currency.

During reflection, we need to extract a set of data from the objects that helps with report layout and formatting. The `ColumnDetail` helps with this because when processing each column, we need to know:

- `Name` to ensure we're working on the right column
- `Attribute` for formatting column data

- `PropertyInfo` for getting property data

The `GetColumnDetails` method populates a `ColumnDetail` for each column. Getting the first object in the data, it gets the type and then calls `GetProperties` on the types for a `PropertyInfo[]`. Unlike Section 5.1, which calls `GetMembers` for a `MemberInfo[]`, this only gets the properties from the type and not any other members.

TIP

In addition to `GetMembers` and `GetProperties`, `Type` has other reflection methods that will only get constructors, fields, or methods. These would be useful if you need to restrict the type of member you're working with.

Because reflection returns a collection of objects, `PropertyInfo[]` in this solution, we can use LINQ to Objects for a more declarative approach. This is what `GetColumnDetails` does, projecting into `ColumnDetails` instances and returning a `Dictionary` with the column name as key and `ColumnDetail` as value.

NOTE

As you'll see later in the solution, the code iterates through the `Dictionary<string, ColumnDetail>`, assuming that columns and their data are laid out in the order returned by reflection queries. However, imagine a future implementation where `ColumnAttribute` had an `Order` property or the consumer could pass `include/exclude` column metadata that didn't guarantee the order of the columns match what reflection returned. In that case, having the dictionary is essential to look up `ColumnDetail` metadata based on which column you're working on. Although that's left out of this example to reduce complexity and focus on the original problem statement, it might give you ideas on how something like this could be extended.

The `GetHeaders` method does exactly the same thing as Section 5.1, except it's written as LINQ statements to reduce and simplify the code.

The `GetReflectedResult` returns a tuple, `(object, type)`. It's task is to pull out the value from the property and the type of that `PropertyInfo`, property. Here, item is the actual object instance and `PropertyInfo` is the reflected metadata for that property. Using property, the code calls `GetValue` with item as the paramter - it reads that property from item. Again, we’re using reflection and don’t know the type for the property, so we put it in type object. `PropertyInfo` also has a `PropertyType`, which is where we get the `Type` object.

WARNING

This application uses reflection to put property data into a variable of type `object`. If the property type is a value type (e.g. `int`, `double`, `decimal`), you incur a boxing penalty, which affects application performance. If you were doing this millions of times, you might take a second look at your requirements and analyze whether this was a good approach for your scenario. That said, this is a report. Think about how many records you might include in a report for the purpose of displaying the data to a human and any performance issues would be negligible. It’s a classic trade-off of flexibility vs. performance and you just need to think about how it affects your situation.

The `GetColumns` method uses `GetReflectedResult` as it loops through each column for a given object. The collection of `ColumnDetail` is useful here, providing `PropertyInfo` for the current column. The format defaults to no format if the `ColumnAttribute` for a column doesn’t include the `Format` property. The `switch` statement applies the format to the object based on the `Type` returned from `GetReflectedResult`.

NOTE

For simlicity, the `switch`, in `GetColumns`, only contains types in the solution, though you might imagine it including all built-in types. We might have used reflection to invoke `ToString` with a format specifier and type, which we’ll discuss in Section 5.4, to reduce code. However, at some point the additional complexity doesn’t add value. In this case, we’re just covering a finite set of built-in types and once that code is written, it will be unlikely to change. My thoughts on this tradeoff is that sometimes being too clever results in code that’s difficult to read and takes longer to write.

Finally, `GetRows` calls `GetColumns` for each row and returns to `Generate`. Then `Generate`, after having called `GetHeaders` and `GetRows`, appends the results to a `StringBuilder` and returns the string to the caller with the entire report, which you can see in the solution output.

See Also

Section 5.1 Section 5.4

5.3 Instantiating Type Members with Reflection

Problem

You need to instantiate generic types, but don’t know the type or type parameters ahead of time.

Solution

The solution generates a uniquely formatted report, depending on this enum:

```
public enum ReportType
{
    Html,
    Markdown
}
```

Here’s a reusable base class for generating reports:

```
public abstract class GeneratorBase<TData>
{
    public string Generate(List<TData> items)
    {
        StringBuilder report = GetTitle();

        Dictionary<string, ColumnDetail> columnDetails =
            GetColumnDetails(items);
        report.Append(GetHeaders(columnDetails));
        report.Append(GetRows(items, columnDetails));

        return report.ToString();
    }

    protected abstract StringBuilder GetTitle();

    protected abstract StringBuilder GetHeaders(
        Dictionary<string, ColumnDetail> details);

    protected abstract StringBuilder GetRows(
        List<TData> items,
        Dictionary<string, ColumnDetail> details);


    Dictionary<string, ColumnDetail> GetColumnDetails(
        List<TData> items)
    {
        TData itemInstance = items.First();
        Type itemType = itemInstance.GetType();
        PropertyInfo[] itemProperties = itemType.GetProperties();

        return
            (from prop in itemProperties
             let attribute = prop.GetCustomAttribute<ColumnAttribute>
()
             where attribute != null
             select new ColumnDetail
             {
                 Name = prop.Name,
                 Attribute = attribute,
                 PropertyInfo = prop
             })
            .ToDictionary(
                key => key.Name,
                val => val);
    }

    protected List<string> GetColumns(
        IEnumerable<ColumnDetail> details,
        TData item)
    {
        var columns = new List<string>();

        foreach (var detail in details)
        {
            PropertyInfo member = detail.PropertyInfo;
            string format =
                string.IsNullOrWhiteSpace(
                    detail.Attribute.Format) ?
                    "{0}" :
                    detail.Attribute.Format;

            (object result, Type columnType) =
                GetReflectedResult(item, member);

            switch (columnType.Name)
            {
                case "Decimal":
                    columns.Add(
                        string.Format(format, (decimal)result));
                    break;
                case "Int32":
                    columns.Add(
                        string.Format(format, (int)result));
                    break;
                case "String":
                    columns.Add(
                        string.Format(format, (string)result));
                    break;
                default:
                    break;
            }
        }

        return columns;
    }

    (object, Type) GetReflectedResult(TData item, PropertyInfo
property)
    {
        object result = property.GetValue(item);
        Type type = property.PropertyType;

        return (result, type);
    }
}
```

This class uses that base class to generate Markdown reports:

```
    public class MarkdownGenerator<TData> : GeneratorBase<TData>
    {
        const string ColumnSeparator = " | ";

        protected override StringBuilder GetTitle()
        {
            return new StringBuilder("# Report\n\n");
        }

        protected override StringBuilder GetHeaders(
            Dictionary<string, ColumnDetail> details)
        {
            var header = new StringBuilder();

            header.AppendJoin(
                ColumnSeparator,
                from detail in details.Values
                select detail.Attribute.Name);
            header.Append("\n");

            header.AppendJoin(
                ColumnSeparator,
                from detail in details.Values
                let length = detail.Attribute.Name.Length
                select "".PadLeft(length, '-'));

            header.Append("\n");

            return header;
        }

        protected override StringBuilder GetRows(
            List<TData> items,
            Dictionary<string, ColumnDetail> details)
        {
            var rows = new StringBuilder();

            foreach (var item in items)
            {
                List<string> columns =
                    GetColumns(details.Values, item);

                rows.AppendJoin(ColumnSeparator, columns);

                rows.Append("\n");
            }

            return rows;
        }
    }
```

And this class uses that base class to generate HTML reports:

```
public class HtmlGenerator<TData> : GeneratorBase<TData>
{
    protected override StringBuilder GetTitle()
    {
        return new StringBuilder("<h1>Report</h1>\n");
    }

    protected override StringBuilder GetHeaders(
        Dictionary<string, ColumnDetail> details)
    {
        var header = new StringBuilder("<tr>\n");

        header.AppendJoin(
            "\n",
            from detail in details.Values
            let columnName = detail.Attribute.Name
            select $"  <th>{columnName}</th>");

        header.Append("\n</tr>\n");

        return header;
    }

    protected override StringBuilder GetRows(
        List<TData> items,
        Dictionary<string, ColumnDetail> details)
    {
        StringBuilder rows = new StringBuilder();
        Type itemType = items.First().GetType();

        foreach (var item in items)
        {
            rows.Append("<tr>\n");

            List<string> columns =
                GetColumns(details.Values, item);

            rows.AppendJoin(
                "\n",
                from columnValue in columns
                select $"  <td>{columnValue}</td>");

            rows.Append("\n</tr>\n");
        }

        return rows;
    }
}
```

This method manages the report generation process:

```
    public string Generate(List<TData> items, ReportType reportType)
    {
        GeneratorBase<TData> generator = CreateGenerator(reportType);

        string report = generator.Generate(items);

        return report;
    }
```

Here’s a method that uses an enum to figure out which report format to generate:

```
GeneratorBase<TData> CreateGenerator(ReportType reportType)
{
    Type generatorType;

    switch (reportType)
    {
        case ReportType.Html:
            generatorType = typeof(HtmlGenerator<>);
            break;
        case ReportType.Markdown:
            generatorType = typeof(MarkdownGenerator<>);
            break;
        default:
            throw new ArgumentException(
                $"Unexpected ReportType: '{reportType}'");
    }

    Type dataType = typeof(TData);
    Type genericType = generatorType.MakeGenericType(dataType);

    object generator = Activator.CreateInstance(genericType);

    return (GeneratorBase<TData>)generator;
}
```

Here’s another way, via convention, to figure out which report format to generate:

```
GeneratorBase<TData> CreateGenerator(ReportType reportType)
{
    Type dataType = typeof(TData);

    string generatorNamespace = "Section_05_03.";
    string generatorTypeName = $"{reportType}Generator`1";
    string typeParameterName = $"[[{dataType.FullName}]]";

    string fullyQualifiedTypeName =
        generatorNamespace +
        generatorTypeName +
        typeParameterName;

    Type generatorType = Type.GetType(fullyQualifiedTypeName);

    object generator = Activator.CreateInstance(generatorType);

    return (GeneratorBase<TData>)generator;
}
```

The `Main` method passes data and specifies which report format it wants:

```
static void Main()
{
    var inventory = new List<InventoryItem>
    {
        new InventoryItem
        {
            PartNumber = "1",
            Description = "Part #1",
            Count = 3,
            ItemPrice = 5.26m
        },
        new InventoryItem
        {
            PartNumber = "2",
            Description = "Part #2",
            Count = 1,
            ItemPrice = 7.95m
        },
        new InventoryItem
        {
            PartNumber = "3",
            Description = "Part #3",
            Count = 2,
            ItemPrice = 23.13m
        },
    };

    string report =
        new Report<InventoryItem>()
        .Generate(inventory, ReportType.Markdown);

    Console.WriteLine(report);
}
```

Discussion

Section 5.2 created reports, based on a generic object type and this caused us to lose the type safety we’re accustomed to. This section fixes that problem by using generics and showing how to use reflection to instantiate objects with a generic type parameter.

The concept of the previous sections was to generate a report in Markdown format. However, a report generator could be much more useful if it had the ability to generate reports in any format of your choosing. This example refactors the example in Section 5.2 to offer both a Markdown and an HTML output report.

The `ReportType` enum specifies the type of report output to generate: `Html` or `Markdown`. Because we can generate multiple formats, we need separate classes for each format: `HtmlGenerator` and `MarkdownGenerator`. Further, we don't want to duplicate code, so each format generation class derives from `GeneratorBase`.

Notice that `GeneratorBase` is an abstract class (you can't instantiate it), with both `abstract` and implemented methods. The implemented methods in `GeneratorBase` have code that is independent of output formated and all derived generator classes will use: `GetColumns`, `GetColumnDetails`, and `GetReflectedResult`. By definition, the derived generator classes must `override` the `abstract` methods, which are format specific: `GetTitle`, `GetHeaders`, `GetRows`. Looking at `HtmlGenerator` and `MarkdownGenerator`, you can see the `override` implementations for these `abstract` methods.

Now, lets put this all together so it makes sense. When the program starts, the first method called is `Generate`, in `GeneratorBase`. Notice how `Generate` calls the sequence: `GetTitle`, `GetColumnDetails`, `GetHeaders`, and then `GetRows`. This is essentially the same sequence as described in Section 5.2. You can imagine a report being generated top to bottom by writing the title, getting metadata for the rest of the report, writing the header, and then writing each of the rows of the report. To get code reuse and create an extisible framework for adding report formats in the future, we have a general abstract base class `GeneratorBase`, and derived classes that understand the format. Using `MarkdownGenerator` as an example, here's the sequence:

1. External code calls `GeneratorBase.Generate`
2. `Generator.Generate` calls `MarkdownGenerator.GetTitle`
3. `Generator.Generate` calls `Generator.GetColumnDetails`
4. `Generator.Generate` calls `MarkdownGenerator.GetHeader`
5. `Generator.Generate` calls `MarkdownGenerator.GetRows`
6. `MarkdownGenerator.GetRows` calls `Generator.GetColumns`

7. `Generator.GetColumns` calls `Generator.GetReflectedResult`
8. `MarkdownGenerator.GetRows` completes, returning to `Generator.Generate`
9. `Generator.Generate` returns the report to calling code

The `HtmlGenerator` works exactly the same way and so would any future report format. In fact, Section 5.6 extends this example by adding a 3rd format to support creating an Excel report.

NOTE

The solution uses a pattern known as the Template pattern. This is where a base class implements common logic and delegates implementation-specific work to derived classes. This is the object-oriented principle of polymorphism at work.

The fact that we can extend this framework without needing to re-write boiler plate logic makes this a viable approach. Section 5.6 shows how that works.

The `GenerateBase` class is intentionally `abstract` because the only way for this to work is via an instance of a derived class. The `Report.Generate` method calls `GeneratorBase.Generate`. Before doing so, it must figure out which specific `GeneratorBase` derived class to instantiate, via `CreateGenerator` of which are two examples.

The first example of `CreateGenerator` examines the `ReportType` enum to see which type of report to generate via a `switch` statement. As explained in earlier sections, you need a `Type` object to perform reflection, which the `typeof` operator does. Notice that we're getting a generic type with the `<>` suffix, without the type. After that, we use the `typeof` operator to get the type of the type parameter passed to the `Report` class, `TData`. Now, we have a type for both the generic type and it's type parameter. Next, we need to bring the generic type and it's parameter type together to get a fully constructed type. e.g. `HtmlGenerator<TData>` for `Html`. Once you have a fully constructed type, you can use the `Activator` class to call `CreateInstance`, which instantiates the type. With a new instance of the `GeneratorBase`-derived type, `CreateGenerate` returns to `ReportGenerate`, which calls `Generate` on the new instance. As you learned earlier, `GeneratorBase` implements `Generate` for all derived instances.

That is one way to use reflection to instantiate a generic type, as specified by the problem statement. One thing to consider though is whether you want to add more formats to support in the future. You'll have to go back into the `Report` class and change the `switch` statement, which is a configuration by code change. What if you prefer to write the `Report` class one time and never touch it again? Further, what if you preferred a design by the principles of convention over configuration? A good example of convention over configuration in .NET is ASP.NET MVC. A couple of ASP.NET MVC conventions are that controllers go in a `Controllers` folder and views go into a `Views` folder. Another is that the controller name is the URL path with a `Controller` suffix to it's name. Things just work because that's the convention. The second example of `CreateGenerator` uses the convention over configuration approach.

Notice that the second implementation of `CreateGenerator` builds a fully-qualified type name with namespace and typename. e.g. `Section_05_03.HtmlGenerator` for `Html`. Also notice that the `ReportType` enum members match the class names exactly. This means that any time in the future, you can create a new formate, derived from `GeneratorBase`, and add the prefix to `ReportType` with `Generator` as the suffix and it will work. No need to ever touch the `Report` class again, unless adding a new feature.

After getting type objects, both `CreateGenerator` examples call `Activator.CreateInstance` to return a new instance to `Report.Generate`.

Finally, looking at the `Main` method, all a user of this report library needs to do is pass in the data and the `ReportType` they want to generate.

See Also

Section 5.2 Section 5.6

5.4 Invoking Methods with Reflection

Problem

An object you're received has methods that you need to invoke.

Solution

The column metadata class has a `MemberInfo` property:

```
public class ColumnDetail
{
    public string Name { get; set; }

    public ColumnAttribute Attribute { get; set; }

    public MemberInfo MemberInfo { get; set; }
}
```

This class, to be reflected upon, has properties and a method:

```
public class InventoryItem
{
    [Column("Part #")]
    public string PartNumber { get; set; }

    [Column("Name")]
    public string Description { get; set; }

    [Column("Amount")]
    public int Count { get; set; }

    [Column("Price", Format = "{0:c}")]
    public decimal ItemPrice { get; set; }

    [Column("Total", Format = "{0:c}")]
    public decimal CalculateTotal()
    {
        return ItemPrice * Count;
    }
}
```

This method calls `GetMembers` to work with `MemberInfo` instances:

```
Dictionary<string, ColumnDetail> GetColumnDetails(
    List<object> items)
{
    return
        (from member in
            items.First().GetType().GetMembers()
         let attribute =
            member.GetCustomAttribute<ColumnAttribute>()
         where attribute != null
         select new ColumnDetail
         {
             Name = member.Name,
             Attribute = attribute,
             MemberInfo = member
         })
        .ToDictionary(
            key => key.Name,
            val => val);
}
```

This method uses the `MemberInfo` type to determine how to retrieve a value:

```
(object, Type) GetReflectedResult(
    Type itemType, object item, MemberInfo member)
{
    object result;
    Type type;

    switch (member.MemberType)
    {
        case MemberTypes.Method:
            MethodInfo method =
                itemType.GetMethod(member.Name);
            result = method.Invoke(item, null);
            type = method.ReturnType;
            break;
        case MemberTypes.Property:
            PropertyInfo property =
                itemType.GetProperty(member.Name);
            result = property.GetValue(item);
            type = property.PropertyType;
            break;
        default:
            throw new ArgumentException(
                "Expected property or method.");
    }

    return (result, type);
    }
```

Discussion

Earlier sections in this chapter worked primarily with properties as report inputs. In this section we'll modify the example in Section 5.2 and add a method that we'll need to invoke via reflection.

The first change is that `ColumnDetail` has a `MemberInfo` property, which holds metadata for any type member.

The `InventoryItem` class has a `CalculateTotal` method. It multiplies the `ItemPrice` and `Count` to show the total price for that amount of items.

The change in `GetColumnDetails` is in the LINQ statement, where it iterates on the result of `GetMembers`, which is a `MemberInfo[]`. Unlike Section 5.2, we're using `MemberInfo`, which can hold any type member. This is required for this solution because we want information on both properties and methods.

Finally, `GetReflectedResult` has a `switch` statement to figure out how to get a member's value. Since it's a `MemberInfo`, we look at the `MemberType` property to figure out whether we're working with a property or method. In either case, we have to call `GetProperty` or `GetMethod` to get a `PropertyInfo` or `MethodInfo`, respectively. Call the `Invoke` method for methods, with `item` as the object instance to invoke the method on. The second parameter to `Invoke` is `null`, indicating that the method, `CalculateTotal` in this example, doesn't have arguments. If you need to pass arguments, put an `object[]` in the second parameter of `Invoke` with the members in the order that the method expects. As in Section 5.2, call `GetValue` on the `PropertyInfo` instance, with `item` as the object reference to get the value of that property.

To summarize, any time you need to call a method on an object, via reflection, get it's `Type` object, get a `MethodInfo` (even if you nedd the intermediate step of pulling from a `MemberInfo`), and call the `Invoke` method on the `MethodInfo` with the object instance as the argument.

See Also

Section 5.2

5.5 Replacing Reflection with Dynamic Code

Problem

You’re using reflection, but know what some of a type’s members are and want to simplify code.

Solution

This class contains the list of data for a report:

```
public class Inventory
{
    public string Title { get; set; }

    public List<object> Data { get; set; }
}
```

Here’s the `Main` method that populates the data:

```
static void Main()
{
    var inventory = new Inventory
    {
        Title = "Inventory Report",
        Data = new List<object>
        {
            new InventoryItem
            {
                PartNumber = "1",
                Description = "Part #1",
                Count = 3,
                ItemPrice = 5.26m
            },
            new InventoryItem
            {
                PartNumber = "2",
                Description = "Part #2",
                Count = 1,
                ItemPrice = 7.95m
            },
            new InventoryItem
            {
                PartNumber = "3",
                Description = "Part #3",
                Count = 2,
                ItemPrice = 23.13m
            },
        }
    };

    string report = new Report().Generate(inventory);

    Console.WriteLine(report);
}
```

This method uses reflection to extract a property values:

```
    public string Generate(object reportDetails)
    {
        Type reportType = reportDetails.GetType();
        PropertyInfo titleProp = reportType.GetProperty("Title");
        string title = (string)titleProp.GetValue(reportDetails);

        var report = new StringBuilder($"# {title}\n\n");

        PropertyInfo dataProp = reportType.GetProperty("Data");
        List<object> items =
            (List<object>)dataProp.GetValue(reportDetails);

        Dictionary<string, ColumnDetail> columnDetails =
            GetColumnDetails(items);
        report.Append(GetHeaders(columnDetails));
        report.Append(GetRows(items, columnDetails));

        return report.ToString();
    }
```

And this class extracts the same property values, but uses dynamic:

```
    public string Generate(dynamic reportDetails)
    {
        string title = reportDetails.Title;

        var report = new StringBuilder(
            $"# {title}\n\n");

        List<object> items = reportDetails.Data;
        Dictionary<string, ColumnDetail> columnDetails =
            GetColumnDetails(items);
        report.Append(GetHeaders(columnDetails));
        report.Append(GetRows(items, columnDetails));

        return report.ToString();
    }
```

Discussion

The concept of this solution is again to give the user of the report library maximum control over what types they want to work with. However, what if you did have some constraints. E.g. there must be some way to set the report title and you would need to know what that property is. This solution meets the user half-way by telling them to provide an object with `Title` and `Data` properties. `Title` has the report title and `Data` has report rows. They can use any object they want as long as they provide those properties. e.g. if the input objects had other properties on the object we don't care about, it won't affect the report library.

The class we'll use is `Inventory`, with a `Title` string and `Data` collection. The `Main` method populates an `Inventory` instance and passes it to `Generate`.

We have two examples of `Generate`: one that uses reflection and the other uses dynamic. After getting the type, the first example calls `GetProperty` and `GetValue` to get the value of each property. the rest of the method works just like in Section 5.2.

As you see, reflection can be verbose, making many method calls and converting types. This is a good case for using dynamic. We know that `Title` and `Data` exist, so why not just access them? That's what the second example does. First, notice that the `reportDetails` parameter type is `dynamic`. Then observe how the code calls `Title` and `Data`, placing them in strongly typed variables.

NOTE

The `dynamic` type is still type `object`, but with a little extra magic via the DLR.

While you don’t get Intellisense during development, because dynamic doesn’t know what types it’s working with, you do get readable code. Behind the scenes, the Dynamic Language Runtime (DLR) did all the work for you. When you know the members of the types being passed to the code, dynamic is a better mechanism for reflection.

See Also

Section 5.2

5.6 Performing Interop with Office Apps

Problem

You need to populate an Excel spreadsheet with object data with the simplest code possible.

Solution

Here’s an enum with extra members for Excel:

```
public enum ReportType
{
    Html,
    Markdown,
    ExcelTyped,
    ExcelDynamic
}
```

Excel report generator without dynamic:

```
public class ExcelTypedGenerator<TData> : GeneratorBase<TData>
{
    ApplicationClass excelApp;
    Workbook wkBook;
    Worksheet wkSheet;

    public ExcelTypedGenerator()
    {
        excelApp = new ApplicationClass();
        excelApp.Visible = true;
        wkBook = excelApp.Workbooks.Add(Missing.Value);
        wkSheet = (Worksheet)wkBook.ActiveSheet;
    }

    protected override StringBuilder GetTitle()
    {
        wkSheet.Cells[1, 1] = "Report";

        return new StringBuilder("Added Title...\n");
    }

    protected override StringBuilder GetHeaders(
        Dictionary<string, ColumnDetail> details)
    {
        ColumnDetail[] values = details.Values.ToArray();

        for (int i = 0; i < values.Length; i++)
        {
            ColumnDetail detail = values[i];
            wkSheet.Cells[3, i+1] = detail.Attribute.Name;
        }

        return new StringBuilder("Added Header...\n");
    }

    protected override StringBuilder GetRows(
        List<TData> items,
        Dictionary<string, ColumnDetail> details)
    {
        const int DataStartRow = 4;

        int rows = items.Count;
        int cols = details.Count;

        var data = new string[rows, cols];

        for (int i = 0; i < rows; i++)
        {
            List<string> columns =
                GetColumns(details.Values, items[i]);

            for (int j = 0; j < cols; j++)
            {
                data[i, j] = columns[j];
            }
        }

        int FirstCol = 'A';
        int LastExcelCol = FirstCol + cols - 1;
```

```
            int LastExcelRow = DataStartRow + rows - 1;
            string EndRangeCol = ((char)LastExcelCol).ToString();
            string EndRangeRow = LastExcelRow.ToString();

            string EndRange = EndRangeCol + EndRangeRow;
            string BeginRange = "A" + DataStartRow.ToString();

            var dataRange = wkSheet.get_Range(BeginRange, EndRange);
            dataRange.Value2 = data;

            wkBook.SaveAs(
                "Report.xlsx", Missing.Value, Missing.Value,
                Missing.Value, Missing.Value, Missing.Value,
                XlSaveAsAccessMode.xlShared, Missing.Value, Missing.Value,
                Missing.Value, Missing.Value, Missing.Value);

            return new StringBuilder(
                "Added Data...\n" +
                "Excel file created at Report.xlsx");
        }
    }
```

Excel report generator with dynamic:

```
    public class ExcelDynamicGenerator<TData> : GeneratorBase<TData>
    {
        ApplicationClass excelApp;
        dynamic wkBook;
        Worksheet wkSheet;

        public ExcelDynamicGenerator()
        {
            excelApp = new ApplicationClass();
            excelApp.Visible = true;

            wkBook = excelApp.Workbooks.Add();
            wkSheet = wkBook.ActiveSheet;
        }

        protected override StringBuilder GetTitle()
        {
            wkSheet.Cells[1, 1] = "Report";

            return new StringBuilder("Added Title...\n");
        }

        protected override StringBuilder GetHeaders(
            Dictionary<string, ColumnDetail> details)
```

```
{
    ColumnDetail[] values = details.Values.ToArray();

    for (int i = 0; i < values.Length; i++)
    {
        ColumnDetail detail = values[i];
        wkSheet.Cells[3, i+1] = detail.Attribute.Name;
    }

    return new StringBuilder("Added Header...\n");
}

protected override StringBuilder GetRows(
    List<TData> items,
    Dictionary<string, ColumnDetail> details)
{
    const int DataStartRow = 4;

    int rows = items.Count;
    int cols = details.Count;

    var data = new string[rows, cols];

    for (int i = 0; i < rows; i++)
    {
        List<string> columns =
            GetColumns(details.Values, items[i]);

        for (int j = 0; j < cols; j++)
        {
            data[i, j] = columns[j];
        }
    }

    int FirstCol = 'A';
    int LastExcelCol = FirstCol + cols - 1;
    int LastExcelRow = DataStartRow + rows - 1;
    string EndRangeCol = ((char)LastExcelCol).ToString();
    string EndRangeRow = LastExcelRow.ToString();

    string EndRange = EndRangeCol + EndRangeRow;
    string BeginRange = "A" + DataStartRow.ToString();

    var dataRange = wkSheet.get_Range(BeginRange, EndRange);
    dataRange.Value2 = data;

    wkBook.SaveAs(
        "Report.xlsx",
        XlSaveAsAccessMode.xlShared);
```

```
            return new StringBuilder(
                "Added Data...\n" +
                "Excel file created at Report.xlsx");
        }
    }
```

Discussion

This example is based on the multiple report format generation code in Section 5.3, which briefly explains how to add another report type. This solution shows how to do it.

First, notice that the `ReportType` enum has two extra members: `ExcelTyped` and `ExcelDynamic`. Both use the convention where `ExcelTyped` creates a `ExcelTypedGenerator` instance and `ExcelDynamic` creates an `ExcelDynamicGenerator` instance. The difference is that `ExcelTypedGenerator` uses strongly typed code to generate an Excel report and `ExcelDynamicGenerator` uses dynamic code to generate an Excel report.

TIP

You can use techniques like this to automate any Microsoft Office application. The trick is to ensure you've installed Visual Studio Tools for Office (VSTO), via the Visual Studio Installer. This will install what is called Primary Interop Assemblies (PIA). After installation, you can find these PIAs under your Visual Studio installation folder. e.g. the folder on my machine is C:\Program Files (x86)\Microsoft Visual Studio\Shared\Visual Studio Tools for Office\PIA and use the version corresponding to the Microsoft Office version you have installed. If you have an older version of office that the VSTO couldn't install, you should search the Microsoft Downloads site (via Internet search) for Office PIAs.

To see the differences between the two examples, go member by member. In particular, `ExcelTypedGenerator` has strongly typed fields, must use the `Missing.Value` placeholder anytime it doesn't use a parameter, and needs to perform a conversion on return types. Notice the `SaveAs` method call at the end of the `GetRows` method, which is particularly onerous.

In contrast, compare those examples with the `ExcelDynamicGenerator` code. Making the `wkBook` field `dynamic`, rather than strongly typed, transforms the code. No more `Missing.Value` placeholders or type conversions. The code is much easier to write and easier to read.

See Also

Section 5.3

5.7 Creating an Inherently Dynamic Type

Problem

You have data in a proprietary format, but want to access members through an object without parsing yourself.

Solution

This class holds data to display in a report:

```
public class LogEntry
{
    [Column("Log Date", Format = "{0:yyyy-MM-dd hh:mm}")]
    public DateTime CreatedAt { get; set; }

    [Column("Severity")]
    public string Type { get; set; }

    [Column("Location")]
    public string Where { get; set; }

    [Column("Message")]
    public string Description { get; set; }
}
```

These methods get log data and return a list of `DynamicObject` types with that data:

```
static List<dynamic> GetData()
{
    string headers = "Date|Severity|Location|Message";

    string logData = GetLogData();

    return
```

```
        (from line in logData.Split('\n')
         select new DynamicLog(headers, line))
        .ToList<dynamic>();
}

static string GetLogData()
{
    return
"2022-11-12 12:34:56.7890|INFO|Section_05_07.Program|Got this far\n" +
"2022-11-12 12:35:12.3456|ERROR|Section_05_07.Report|Index out of
range\n" +
"2022-11-12 12:55:34.5678|WARNING|Section_05_07.Report|Please check
this";
}
```

This class is a `DynamicObject` that knows how to read log files and dynamically expose properties:

```
public class DynamicLog : DynamicObject
{
    Dictionary<string, string> members =
        new Dictionary<string, string>();

    public DynamicLog(string headerString, string logString)
    {
        string[] headers = headerString.Split('|');
        string[] logData = logString.Split('|');

        for (int i = 0; i < headers.Length; i++)
            members[headers[i]] = logData[i];
    }

    public override bool TryGetMember(
        GetMemberBinder binder, out object result)
    {
        result = members[binder.Name];
        return true;
    }

    public override bool TryInvokeMember(
        InvokeMemberBinder binder, object[] args, out object result)
    {
        return base.TryInvokeMember(binder, args, out result);
    }

    public override bool TrySetMember(
        SetMemberBinder binder, object value)
    {
        members[binder.Name] = (string)value;
        return true;
    }
}
```

The `Main` method consumes the dynamic data, populates data objects, and gets a new report:

```
    static void Main()
    {
        List<dynamic> logData = GetData();

        var tempDateTime = DateTime.MinValue;
        List<object> inventory =
            (from log in logData
             let canParse =
                DateTime.TryParse(
                    log.Date, out tempDateTime)
             select new LogEntry
             {
                 CreatedAt = tempDateTime,
                 Type = log.Severity,
                 Where = log.Location,
                 Description = log.Message
             })
            .ToList<object>();

        string report = new Report().Generate(inventory);

        Console.WriteLine(report);
    }
```

Discussion

The `DynamicObject` type is part of the .NET Framework and supports the Dynamic Language Runtime (DLR) for interoperability with dynamic languages. It's a peculiar type that lets anyone call type members and it can intercept the call and behave in any way you've programmed it to. Rather than wave hands and enumerate several ways to use `DynamicObject`, this solution focuses on the problem where you need an object to work on proprietary data. In this solution, the data is a log file format. Here, we'll use the `DynamicObject` to provide the data and the report library from Section 5.2 to display the log data.

The `LogEntry` class represents a row in the report. We can't give a

DynamicObject instance to Report because there isn't a way to reflect on it and extract attributes. Any work-around is cumbersome and it's easier to use the DynamicObject for working with the data, populate LogEntry, and give the collection of LogEntry to the Report.

The GetLogData method shows what the log file looks like. GetData creates a headers string, which is metadata for each entry of the log file. The LINQ query iterates through each line of the log, resulting in a List<dynamic>. The projection instantiates a new DynamicLog instance with the header and log entry.

The DynamicLog type derives from DynamicObject, implementing only the methods it needs. The DynamicLog implementation shows a few of these members: TryGetMember, TryInvokeMember, and TrySetMember. The solution doesn't use TryInvokeMember, but I left it in there to show that DynamicObject does more that work with properties and there are other overloads. The Dictionary<string, string>, members, hold a value for each field in the log with the key coming from the header and the value coming from the identially positioned string in the log file.

The constructor populates members. It splits each field on the pipe, |, separator and iterates through the headers until members has an entry for each column. The TryGetMembers method reads from the dictionary to return the value via the out object result parameter. Remember to return true when successful because returning false indicates that you couldn't perform the operation and the user will receive a runtime exception. TrySetMember populates the dictionary with the value.

GetMemberBinder and SetMemberBinder contain metadata on the property that is being accessed. For example, the following would call TryGetMember:

```
string severity = log.Severity;
```

Assuming that log is an instance of DynamicLog, this the GetMemberBinder Name property would be "Severity". It would index into the dictionary and return whatever value is assigned to that key. Similarly, the following would call TrySetMember:

```
log.Severity = "ERROR";
```

In this case, `binder.Name` would be “Severity” and it would update that key in the dictionary with the value “ERROR”.

That means now we have an object where you can set property names of your choosing and provide any log file of the same format (pipe-separated). No need for a custom class every time you want to accommodate a pipe-separated format log file.

`GetData` returns a `List<dynamic>`. Because it’s a dynamic object and we already know what the property names should be (they match the header), we can project into `LogEntry` instances by only specifying the property name on the dynamic object. Additionally, you could specify what those headers should be in a configuration file or database where they can be data driven and change every time. Maybe you even want the ability to change the delimeter on the file to accomodate handling even more file types. As you can see, that’s easy to do with `DynamicObject`.

See Also

Section 5.2

5.8 Adding and Removing Type Members Dynamically

Problem

You want a fully dynamic object, just like JavaScript that you can add members to during runtime.

Solution

This method uses an `ExpandoObject` to collect data:

```
static List<dynamic> GetData()
{
    const int Date = 0;
```

```
        const int Severity = 1;
        const int Location = 2;
        const int Message = 3;

        var logEntries = new List<dynamic>();

        string logData = GetLogData();

        foreach (var line in logData.Split('\n'))
        {
            string[] columns = line.Split('|');

            dynamic logEntry = new ExpandoObject();

            logEntry.Date = columns[Date];
            logEntry.Severity = columns[Severity];
            logEntry.Location = columns[Location];
            logEntry.Message = columns[Message];

            logEntries.Add(logEntry);
        }

        return logEntries;
    }

    static string GetLogData()
    {
        return
            "2022-11-12 12:34:56.7890|INFO" +
            "|Section_05_07.Program|Got this far\n" +
            "2022-11-12 12:35:12.3456|ERROR" +
            "|Section_05_07.Report|Index out of range\n" +
            "2022-11-12 12:55:34.5678|WARNING" +
            "|Section_05_07.Report|Please check this";
    }
```

The `Main` method converts a `List<dynamic>` to a `List<LogEntry>` and gets the report:

```
    static void Main()
    {
        List<dynamic> logData = GetData();

        var tempDateTime = DateTime.MinValue;
        List<object> inventory =
            (from log in logData
             let canParse =
                DateTime.TryParse(
                    log.Date, out tempDateTime)
             select new LogEntry
             {
                 CreatedAt = tempDateTime,
                 Type = log.Severity,
                 Where = log.Location,
                 Description = log.Message
             })
            .ToList<object>();

        string report = new Report().Generate(inventory);

        Console.WriteLine(report);
    }
```

Discussion

This is similar to the `DynamicObject` example in Section 5.7, except it covers a simpler case where you don't need as much flexibility. What if you knew what the file format was ahead of time and know it won't change, yet you just want a simple way to pull the data into a dynamic object without creating a new type every time you need to send data to the report.

In this case, you can use `ExpandoObject`, a .NET Framework type that lets you add and remove type members on-the-fly, the same as in JavaScript.

In the solution, the `GetData` method instantiates an `ExpandoObject`, assigning it to the `dynamic logEntry`. Then, it adds properties on-the-fly and populates them with the parsed log file data.

The `Main` method accepts a `List<dynamic>` from `GetData`. As long as each object has the properties it expects, everything works well.

See Also

Section 5.7

5.9 Calling Python Code from C#

Problem

You have a C# program and want to use Python code, but don't want to re-write it.

Solution

This Python file has code that we need to use:

```
import sys
sys.path.append(
    "/System/Library/Frameworks/Python.framework" +
    "/Versions/Current/lib/python2.7")

from random import *

class SemanticAnalysis:
    @staticmethod
    def Eval(text):
        val = random()
        return val < .5
```

This class represents social media data:

```
public class Tweet
{
    [Column("Screen Name")]
    public string ScreenName { get; set; }

    [Column("Date")]
    public DateTime CreatedAt { get; set; }

    [Column("Text")]
    public string Text { get; set; }

    [Column("Semantic Analysis")]
    public string Semantics { get; set; }
}
```

The `Main` method gets data and generates a report:

```
static void Main()
{
    List<object> tweets = GetTweets();

    string report = new Report().Generate(tweets);

    Console.WriteLine(report);
}
```

These are the required namespaces that are part of the IronPython NuGet package:

```
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;
```

This method sets up the Python interop:

```
static List<object> GetTweets()
{
    ScriptRuntime py = Python.CreateRuntime();
    dynamic semantic = py.UseFile("../../../Semantic.py");
    dynamic semanticAnalysis = semantic.SemanticAnalysis();

    DateTime date = DateTime.UtcNow;

    var tweets = new List<object>
    {
        new Tweet
        {
            ScreenName = "SomePerson",
            CreatedAt = date.AddMinutes(5),
            Text = "Comment #1",
            Semantics = GetSemanticText(semanticAnalysis, "Comment
#1")
        },
        new Tweet
        {
            ScreenName = "SomePerson",
            CreatedAt = date.AddMinutes(7),
            Text = "Comment #2",
            Semantics = GetSemanticText(semanticAnalysis, "Comment
#2")
        },
        new Tweet
        {
            ScreenName = "SomePerson",
            CreatedAt = date.AddMinutes(12),
            Text = "Comment #3",
            Semantics = GetSemanticText(semanticAnalysis, "Comment
#3")
        },
    };

    return tweets;
}
```

This method calls the Python code via dynamic instance:

```
static string GetSemanticText(dynamic semantic, string text)
{
    bool result = semantic.Eval(text);
    return result ? "Positive" : "Negative";
}
```

Discussion

The scenario in this example is where you’re working with social media data. One of the report items is semantics, telling whether a user’s tweet was positive or negative. You’ve got this great semantic analysis AI model, but it’s built with TensorFlow in a Python module. You really don’t want to re-write that code and it would be great to reuse it.

This is where the Dynamic Language Runtime (DLR) comes in because it lets you call Python (and other dynamic languages) from C#. Considering that it could have taken many months to build a machine learning model (or any other type of module), the advantage of reusing that code accross languages can be huge.

The `SemanticAnalysis` class in the Python file simulates a model, returning `true` for a positive result or `false` for a negative result.

The `Main` method calls `GetTweets` to get data and uses the `Report` class, which is the same as Section 5.2. The `List<object>` returned from `GetTweets` contains `Tweet` objects that can work with the report generator.

TIP

To set this up, you’ll need to reference the `IronPython` package, which you can find on NuGet. You also might find it useful to install Python Tools for Visual Studio via the Visual Studio Installer.

The `GetTweets` method needs a reference to the Python `SemanticAnalysis` class. Calling `CreateRuntime` creates a DLR reference. Then you need to specify the location of the Python file via

`UseFile`. After that, you can instantiate the `SemanticAnalysis` class. Each `Tweet` instance sets the `Semantics` property with a call to `GetSemanticText`, passing the `SemanticAnalysis` reference and `text` to evaluate.

The `GetSemanticText` method calls `Eval` with `text` as it's parameter and returns a `bool` result, which it then translates to a report-friendly "Positive" or "Negative" string.

In just a few lines of code, you saw how easy it is to reuse important code that was written in a dynamic language. Other supported languages include Ruby, and JavaScript.

See Also

Section 5.2

5.10 Calling C# Code from Python

Problem

You have a Python program and want to use C# code, but don't want to re-write it .

Solution

Here's the main Python application that needs to use the report generator:

```
import clr, sys

sys.path.append(
    r"C:\Path Where You Cloned The Project" +
    "\Chapter05\Section-05-10\bin\Debug")
clr.AddReference(
    r"C:\Path Where You Cloned The Project" +
    \Chapter05\Section-05-10\bin\Debug\PythonToCS.dll")

from PythonToCS import Report
from PythonToCS import InventoryItem
from System import Decimal
```

```
inventory = [
    InventoryItem("1", "Part #1", 3, Decimal(5.26)),
    InventoryItem("2", "Part #2", 1, Decimal(7.95)),
    InventoryItem("3", "Part #1", 2, Decimal(23.13))]

rpt = Report()

result = rpt.GenerateDynamic(inventory)

print(result)
```

This class has a constructor to make it easier to work with in Python:

```
public class InventoryItem
{
    public InventoryItem(
        string partNumber, string description,
        int count, decimal itemPrice)
    {
        PartNumber = partNumber;
        Description = description;
        Count = count;
        ItemPrice = itemPrice;
    }

    [Column("Part #")]
    public string PartNumber { get; set; }

    [Column("Name")]
    public string Description { get; set; }

    [Column("Amount")]
    public int Count { get; set; }

    [Column("Price", Format = "{0:c}")]
    public decimal ItemPrice { get; set; }
}
```

Here’s the C# method that the Python code calls to generate the report:

```
public string GenerateDynamic(dynamic[] items)
{
    List<object> inventory =
        (from item in items
         select new InventoryItem
         (
             item.PartNumber,
             item.Description,
             item.Count,
             item.ItemPrice
         ))
        .ToList<object>();

    return Generate(inventory);
}
```

Discussion

In Section 5.9, the scenario was to call Python from C#. The scenario in this problem is opposite in that I have a Python application and need to be able to generate reports. However, the report generator is written in C#. So much work has gone into the report library and it doesn't make sense to rewrite in Python. Fortunately, the Dynamic Language Runtime (DLR) allows us to call that C# code with Python.

The report is the same one used in Section 5.2 and the C# code has the same `InventoryItem` class.

TIP

To set this up, you might need to install the pythonnet package:

```
>pip install pythonnet
```

You can find more info at *https://pypi.org/project/pythonnet/*

You set up the Python code by importing `clr` and `sys`, calling `sys.path.append` as a reference to the path where the C# dll resides and then calling `clr.AddReference` to add a reference to the C# dll you want to use.

In Python, whenever you need to use a .NET type from either the framework or custom assembly, use the `from Namespace import type` syntax, which is roughly equivalent to a C# `using` declaration. The namespace in the C# source code is `PythonToCS` and the code uses that to import a reference to `Report` and `InventoryItem`. It also uses `System` namespace to get a reference to the `Decimal` type, which aliases the C# `decimal` type.

In Python, whenever you use square brackets, `[]`, you're creating a data structure called a `list`. It's a collection of objects with Python semantics. In this example, we're creating a list of `InventoryItem`, assigning it to a variable named `inventory`.

Notice how we're using `Decimal` for the last parameter, `itemPrice`, of the `InventoryItem` constructor. Python doesn't have a concept of `decimal` and will pass that value as a `float`, which causes an error because the C# `InventoryItem` defines that parameter as a `decimal`.

Next, the Python code instantiates `Report`, `rpt`, and calls `GenerateDynamic`, passing `inventory`.

This calls the `GenerateDynamic` in `Report` and automatically translates inventory from a Python `list` into a C# `dynamic[]`, `items`. Because each object in `items` is `dynamic`, we can query with a LINQ statement, accessing the names of each object dynamically in the projection.

Finally, `GenerateDynamically` calls `Generate`, the the application returns a report, and the Python code prints the report.

See Also

Section 5.2 Section 5.9

About the Author

Joe Mayo is an author, instructor, and independent consultant who has been working with C# and .NET since its announcement in the summer of the year 2000. As an independent consultant, he’s worked with a variety of organizations from startup to fortune 500 enterprise. His experience in this journey includes desktop, web, mobile, cloud, and AI technologies. In addition to practical hands-on application, he’s also taught C# and .NET for many years via in-person, live video, and recorded video courses. His top open-source project is LINQ to Twitter (on GitHub), with over one million NuGet downloads. When Joe isn’t serving valued customers, he contributes to the community through Q&A forums, presenting, and (one of his favorite pastimes) writing.