Before you start

Exercise scenario

Set up the library application in Visual Studio Code

Analyze and refactor code using the Chat view in Ask and Edit mode

Refactor code using inline chat and the Chat view in Edit and Agent modes

Summary

Clean up

Refactor existing code using GitHub Copilot

GitHub Copilot can be used to evaluate your entire codebase and suggest updates that help you to refactor and improve your code. In this exercise, you use GitHub Copilot to refactor specified sections of a C# application while making improvements to code quality, reliability, performance, and security.

This exercise should take approximately **30** minutes to complete.

> **IMPORTANT**: To complete this exercise, you must provide your own GitHub account and GitHub Copilot subscription. If you don't have a GitHub account, you can sign up for a free individual account and use a GitHub Copilot Free plan to complete the exercise. If you have access to a GitHub Copilot Pro, GitHub Copilot Pro+, GitHub Copilot Business, or GitHub Copilot Enterprise subscription from within your lab environment, you can use your existing GitHub Copilot subscription to complete this exercise.

Before you start

Your lab environment must include the following: Git 2.48 or later, .NET SDK 9.0 or later, Visual Studio Code with the C# Dev Kit extension, and access to a GitHub account with GitHub Copilot enabled.

If you're using a local PC as a lab environment for this exercise:

- For help configuring your local PC as your lab environment, open the following link in a browser: Configure your lab environment resources.
- For help enabling your GitHub Copilot subscription in Visual Studio Code, open the following link in a browser: Enable GitHub Copilot within Visual Studio Code.

If you're using a hosted lab environment for this exercise:

- For help enabling your GitHub Copilot subscription in Visual Studio Code, paste the following URL into a browser's site navigation bar: Enable GitHub Copilot within Visual Studio Code.
- Open a command terminal and then run the following commands:

  To ensure that Visual Studio Code is configured to use the correct version of .NET, run the following command:

  Code Copy

  ```
  dotnet nuget add source https://api.nuget.org/v3/index.json -n nuget.org
  ```

Exercise scenario

You're a developer working in the IT department of your local community. The backend systems that support the public library were lost in a fire. Your team needs to develop a temporary solution to help the library staff manage their operations until the system can be replaced. Your team chose GitHub Copilot to accelerate the development process.

You handed off an initial version of the library application for review. The review team identified opportunities to improve code quality, performance, readability, maintainability, and security.

The following updates are assigned to you:

1. Refactor the EnumHelper class to use static dictionaries instead of reflection.
    - Using static dictionaries will improve performance (removes the overhead of reflection).

- Eliminating reflection also improves code readability, maintainability, and security.

2. Refactor the data access methods to use LINQ (Language Integrated Query) rather than foreach loops.
    - Using LINQ provides a more concise and readable way to query collections, databases, and XML documents.
    - Using LINQ can improve code readability, maintainability, and performance.

This exercise includes the following tasks:

1. Set up the library application in Visual Studio Code.
2. Analyze and refactor code using the Chat view in Ask and Edit modes.
3. Refactor code using inline chat and the Chat view in Edit and Agent modes.

Set up the library application in Visual Studio Code

You need to download the existing application, extract the code files, and then open the solution in Visual Studio Code.

Use the following steps to set up the library application:

1. Open a browser window in your lab environment.
2. To download a zip file containing the library application, paste the following URL into your browser’s address bar: GitHub Copilot lab - refactor existing code

    The zip file is named **AZ2007LabAppM5.zip**.

3. Extract the files from the **AZ2007LabAppM5.zip** file.

    For example:

    a. Navigate to the downloads folder in your lab environment.

    b. Right-click **AZ2007LabAppM5.zip**, and then select **Extract all**.

    c. Select **Show extracted files when complete**, and then select **Extract**.

4. Open the extracted files folder, then copy the **AccelerateDevGHCopilot** folder to a location that’s easy to access, such as your Windows Desktop folder.
5. Open the **AccelerateDevGHCopilot** folder in Visual Studio Code.

    For example:

    a. Open Visual Studio Code in your lab environment.

    b. In Visual Studio Code, on the **File** menu, select **Open Folder**.

    c. Navigate to the Windows Desktop folder, select **AccelerateDevGHCopilot** and then select **Select Folder**.

6. In the Visual Studio Code SOLUTION EXPLORER view, verify the following solution structure:
    - AccelerateDevGHCopilot
        - src
            - Library.ApplicationCore\
            - Library.Console\
            - Library.Infrastructure\
        - tests
            - UnitTests\
7. Ensure that the solution builds successfully.

For example, in the SOLUTION EXPLORER view, right-click **AccelerateDevGHCopilot**, and then select **Build**.

You'll see some Warnings, but there shouldn't be any Errors.

Analyze and refactor code using the Chat view in Ask and Edit mode

Reflection is a powerful coding feature that allows you to inspect and manipulate objects at runtime. However, reflection can be slow and there are potential security risks associated with reflection that should be considered.

You need to:

1. Analyze your workspace and investigate how to address your assigned task.
2. Refactor the EnumHelper class to use static dictionaries instead of reflection.

Analyze the EnumHelper class using the Chat view in Ask mode

GitHub Copilot's Chat view has three modes: **Ask**, **Edit**, and **Agent**. Each mode is designed for different types of interactions with GitHub Copilot.

- **Ask**: Use this mode to ask GitHub Copilot questions about your codebase. You can ask GitHub Copilot to explain code, suggest changes, or provide information about the codebase.
- **Edit**: Use this mode to edit selected code files. You can use GitHub Copilot to refactor code, add comments, or make other changes to your code.
- **Agent**: Use this mode to run GitHub Copilot as an agent. You can use GitHub Copilot to run commands, execute code, or perform other tasks in your workspace.

In this section of the exercise, you use the Chat view in Ask mode to analyze your coding assignment.

Use the following steps to complete this section of the exercise:

1. In the SOLUTION EXPLORER view, expand the **Library.ApplicationCore** folder, and then expand the **Enums** folder.

2. Open the EnumHelper.cs file and review the existing code.

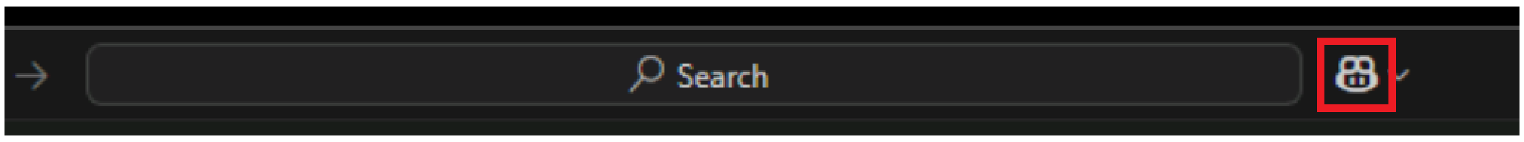
   C# Copy

```
 using System.ComponentModel;
 using System.Reflection;

 namespace Library.ApplicationCore.Enums;

 public static class EnumHelper
 {
     public static string GetDescription(Enum value)
     {
         if (value == null)
             return string.Empty;

         FieldInfo fieldInfo = value.GetType().GetField(value.ToString())!;

         DescriptionAttribute[] attributes =

(DescriptionAttribute[])fieldInfo.GetCustomAttributes(typeof(DescriptionAttribute),
false);

         if (attributes != null && attributes.Length > 0)
         {
             return attributes[0].Description;
         }
         else
         {
             return value.ToString();
         }
     }
 }
```

3. Open the GitHub Copilot Chat view.

   The Chat view provides a managed conversational interface for interacting with GitHub Copilot.

   You can toggle the Chat view between open and closed using the **Toggle Chat** button, which is located at the top of the Visual Studio Code window, just to the right of the search textbox.

   You can also use the keyboard shortcut **Ctrl+Alt+I** to toggle the Chat view.

4. Notice that the Chat view opens in **Ask** mode by default.

   The current Chat mode is displayed near the bottom-right corner of the Chat view. Chat responses are displayed in the Chat view when you're working in **Ask** mode.

5. Select the code in the EnumHelper.cs file.

6. Review and then submit the following prompt:

```
@workspace Explain how the GetDescription method uses reflection to assign the
return value.
```

7. Take a minute to review the response.

   The **GetDescription** method uses reflection to retrieve the description attribute of an enum parameter named **value**.

The method checks if the **value** parameter is null. If it is, the method returns an empty string. Otherwise, it uses reflection to get the field information for the enum value and retrieves the attributes of type **DescriptionAttribute**. If any attributes are found, it returns the description; otherwise, it returns the string representation of the enum value.

8. Review and then submit the following prompt:

   Code Copy

   ```
    @workspace Which files in this workspace are used to store the enum values passed to
   the GetDescription method?
   ```

   The response should tell you to check the Enums folder. The enum values are defined in the **LoanExtensionStatus**, **LoanReturnStatus**, and **MembershipRenewalStatus** files.

9. Add the following files to the Chat context:

   - EnumHelper.cs
   - LoanExtensionStatus.cs
   - LoanReturnStatus.cs
   - MembershipRenewalStatus.cs

   You can use a drag-and-drop operation to add the files from Visual Studio Code’s explorer view to the Chat view. You can also use the **Add Context** button in the Chat view to add files and other resources.

   > **NOTE**: Adding files to the Chat context ensures that GitHub Copilot considers those files when generating a response. The relevance and accuracy of responses increase when GitHub Copilot understands the context associated with your prompts.

10. Review and then submit the following prompt:

    Code Copy

    ```
     @workspace I need to refactor the `EnumHelper` class and remove any code that uses
    reflection. Use static dictionaries to supply enum description attributes. Use a
    separate dictionary for each enum. The dictionaries should use values from the
    `LoanExtensionStatus.cs`, `LoanReturnStatus.cs`, and `MembershipRenewalStatus.cs`
    files. Explain how to update the EnumHelper class using dictionaries and show me the
    updated code.
    ```

    When writing any prompt, clarity and context are important. Using chat participants, slash commands, and chat variables helps to define context in a way that GitHub Copilot can understand.

    When writing a prompt that asks GitHub Copilot how to solve a problem, start with the problem that you’re trying to solve. Use concise sentences to describe details, specify constraints, and identify resources. Finally, be sure to tell GitHub Copilot what to include in the response.

    In this case, your prompt begins with a description of your problem/goal. You tell GitHub Copilot that you need to refactor the **EnumHelper** class and remove any code that uses reflections. You tell GitHub Copilot that you want to use dictionaries to supply enum description attributes. You specify that you want a separate dictionary for each enum, and you tell GitHub Copilot that the enum values should be sourced from the files that you name. You finish by asking GitHub Copilot to explain how to update the **EnumHelper** class, and you ask for a code sample containing the updated code.

11. Take a minute to review the response provided by GitHub Copilot.

    The response should be similar to the following markdown and code samples:

    Code Copy

```
 To refactor the EnumHelper class and remove reflection, you can create static
dictionaries for each enum type. These dictionaries will map each enum value to its
description. The dictionaries will be initialized with the descriptions defined in
the Description attributes of the enums.

 Here's how to update the **EnumHelper** class:

 1. Create a static dictionary for each enum (LoanExtensionStatus, LoanReturnStatus,
and MembershipRenewalStatus).
 2. Populate these dictionaries with the enum values as keys and their descriptions
as values.
 3. Update the GetDescription method to retrieve the description from the
appropriate dictionary based on the enum type.

 Here’s the updated code:
```

C# Copy

```
using System.Collections.Generic;

namespace Library.ApplicationCore.Enums;

public static class EnumHelper
{
    private static readonly Dictionary<LoanExtensionStatus, string> LoanExtensionStatusDescriptions = new()
    {
        { LoanExtensionStatus.Success, "Book loan extension was successful." },
        { LoanExtensionStatus.LoanNotFound, "Loan not found." },
        { LoanExtensionStatus.LoanExpired, "Cannot extend book loan as it already has expired. Return the book instead." },
        { LoanExtensionStatus.MembershipExpired, "Cannot extend book loan due to expired patron's membership." },
        { LoanExtensionStatus.LoanReturned, "Cannot extend book loan as the book is already returned." },
        { LoanExtensionStatus.Error, "Cannot extend book loan due to an error." }
    };

    private static readonly Dictionary<LoanReturnStatus, string> LoanReturnStatusDescriptions = new()
    {
        { LoanReturnStatus.Success, "Book was successfully returned." },
        { LoanReturnStatus.LoanNotFound, "Loan not found." },
        { LoanReturnStatus.AlreadyReturned, "Cannot return book as the book is already returned." },
        { LoanReturnStatus.Error, "Cannot return book due to an error." }
    };

    private static readonly Dictionary<MembershipRenewalStatus, string> MembershipRenewalStatusDescriptions = new()
    {
        { MembershipRenewalStatus.Success, "Membership renewal was successful." },
        { MembershipRenewalStatus.PatronNotFound, "Patron not found." },
        { MembershipRenewalStatus.TooEarlyToRenew, "It is too early to renew the membership." },
        { MembershipRenewalStatus.LoanNotReturned, "Cannot renew membership due to an outstanding loan." },
        { MembershipRenewalStatus.Error, "Cannot renew membership due to an error." }
    };

    public static string GetDescription<TEnum>(TEnum value) where TEnum : Enum
    {
        return value switch
        {
            LoanExtensionStatus loanExtensionStatus => LoanExtensionStatusDescriptions[loanExtensionStatus],
            LoanReturnStatus loanReturnStatus => LoanReturnStatusDescriptions[loanReturnStatus],
            MembershipRenewalStatus membershipRenewalStatus => MembershipRenewalStatusDescriptions[membershipRenewalStatus],
            _ => value.ToString()
        };
    }
```

```
 }
```

Code Copy

```
 Explanation:

 1. Static Dictionaries: Each dictionary is specific to an enum type and maps enum
values to their descriptions.
 2. Generic Method: The GetDescription method is now generic and uses a switch
expression to determine which dictionary to use based on the enum type.
 3. No Reflection: The use of reflection is completely removed, improving
performance and type safety.

 This approach ensures that the descriptions are statically defined and avoids the
runtime overhead of reflection.
```

12. In the Chat view, hover the mouse pointer over the code sample included in the response.
13. Notice the three buttons that appear in the top-right corner of the code snippet.
14. Hover the mouse pointer over each of the buttons to see a tooltip that describes the action.

    The first two buttons copies code into the editor. The third button copies code to the clipboard.

> **NOTE**: You could use the Ask mode to update the **EnumHelper** class. However, the Edit mode refactors your code directly within the code editor and provides more options for accepting updates.

Refactor the EnumHelper class using the Chat view in Edit mode

The Chat view’s Edit mode is designed for editing code in your workspace. You can use the Edit mode to refactor code, add comments, or make other changes to your code.

1. In the Chat view, select **Set Mode**, and then select **Edit**.

   When prompted to start a new session in the Edit mode, select **Yes**.

   In **Edit** mode, GitHub Copilot displays responses as code update suggestions in code editor. The Edit mode is generally used when implementing a new feature, fixing a bug, or refactoring code.

2. Add the following files to the Chat context:

   - EnumHelper.cs
   - LoanExtensionStatus.cs
   - LoanReturnStatus.cs
   - MembershipRenewalStatus.cs

3. Review and then submit the following prompt:

   Code Copy

```
#codebase I need to refactor the `EnumHelper` class and remove any code that uses reflection. Use static dictionaries to supply enum description attributes. Use a separate dictionary for each enum. The dictionaries should use values from the `LoanExtensionStatus.cs`, `LoanReturnStatus.cs`, and `MembershipRenewalStatus.cs` files.
```

This prompt tells GitHub Copilot to refactor the **EnumHelper** class using dictionaries rather than reflection to assign enum description attributes. It specifies that a separate dictionary should be used for each enum, and that the enum values should be sourced from specific files.

4. Take a minute to review the suggested code updates.

   Review the suggested updates to ensure that the enum values are coming from the **LoanExtensionStatus.cs**, **LoanReturnStatus.cs**, and **MembershipRenewalStatus.cs** files.

   You can open each of the enum files to verify that the enum values in the dictionaries are correct. If you find discrepancies, have GitHub Copilot update the dictionaries for each enum individually. For example, you can use the following prompt for the **LoanExtensionStatus** enum:

   Code Copy

   ```
   #codebase Use the description values in LoanExtensionStatus.cs to update the LoanExtensionStatus dictionary in the EnumHelper class. Provide the updated code for the LoanExtensionStatus dictionary in the EnumHelper class.
   ```

5. In the Chat view, to accept all updates, select **Keep**.

   You could also use the Chat Edits toolbar near the bottom of the code editor tab to accept or reject code updates.

6. Take a minute to review the updated **GetDescription** method.

   GitHub Copilot should have updated the **GetDescription** method to use pattern matching and static dictionaries instead of reflection. The updated method should look similar to one of the following examples:

   C# Copy

```
public static string GetDescription(Enum value)
{
    if (value == null)
        return string.Empty;

    // Use type checks to select the correct dictionary
    if (value is LoanExtensionStatus les && LoanExtensionStatusDescriptions.TryGetValue(les, out var lesDesc))
        return lesDesc;
    if (value is LoanReturnStatus lrs && LoanReturnStatusDescriptions.TryGetValue(lrs, out var lrsDesc))
        return lrsDesc;
    if (value is MembershipRenewalStatus mrs && MembershipRenewalStatusDescriptions.TryGetValue(mrs, out var mrsDesc))
        return mrsDesc;

    return value.ToString();
}
```

or

C# Copy

```
public static string GetDescription<TEnum>(TEnum value) where TEnum : Enum
{
    return value switch
    {
        MembershipRenewalStatus status => MembershipRenewalDescriptions.TryGetValue(status, out var description) ? description : status.ToString(),
        LoanReturnStatus status => LoanReturnDescriptions.TryGetValue(status, out var description) ? description : status.ToString(),
        LoanExtensionStatus status => LoanExtensionDescriptions.TryGetValue(status, out var description) ? description : status.ToString(),
        _ => value.ToString()
    };
}
```

This code uses pattern matching to determine the type of the enum and retrieve the description from the appropriate dictionary. The **switch** statement checks the type of the enum **value** and returns the corresponding description from the dictionary. If the enum value isn't found in the dictionary, the method falls back to calling ToString() on the enum value, which returns the name of the enum member as a string.

If you have GitHub Copilot refactor the GetDescription method to eliminate the lambda expressions, the underlying logic is easier to follow:

C# Copy

```
public static string GetDescription<TEnum>(TEnum value) where TEnum : Enum
{
    switch (value)
    {
        case MembershipRenewalStatus status:
            string membershipDescription;
            if (MembershipRenewalDescriptions.TryGetValue(status, out
membershipDescription))
            {
                return membershipDescription;
            }
            return status.ToString();

        case LoanReturnStatus status:
            string loanReturnDescription;
            if (LoanReturnDescriptions.TryGetValue(status, out
loanReturnDescription))
            {
                return loanReturnDescription;
            }
            return status.ToString();

        case LoanExtensionStatus status:
            string loanExtensionDescription;
            if (LoanExtensionDescriptions.TryGetValue(status, out
loanExtensionDescription))
            {
                return loanExtensionDescription;
            }
            return status.ToString();

        default:
            return value.ToString();
    }
}
```

7. Build your solution to ensure that there are no errors were introduced.

   You'll see the same warnings that you saw at the start of this exercise, but there shouldn't be any error messages.

Refactor code using inline chat and the Chat view in Edit and Agent modes

LINQ is a powerful feature in C# that allows you to query collections, databases, and XML documents in a uniform way. LINQ provides a more concise and readable way to query data compared to traditional foreach loops.

This section of the exercise includes the following tasks:

- Refactor the JsonData class using inline chat.
- Refactor the JsonLoanRepository class using the Chat view in Edit mode.
- Refactor the JsonPatronRepository class using the Chat view in Agent mode.

Refactor the JsonData class using inline chat

The JsonData class includes the following data access methods: GetPopulatedPatron, GetPopulatedLoan, GetPopulatedBookItem, GetPopulatedBook. These methods use foreach loops to iterate over collections and populate objects. You can refactor these methods to use LINQ to improve code readability and maintainability.

Use the following steps to complete this section of the exercise:

1. In the SOLUTION EXPLORER view, expand the **Library.Infrastructure** project, and then expand the **Data** folder.

2. Open the JsonData.cs file.

3. Scroll down to locate the **GetPopulatedPatron** method.

   The **GetPopulatedPatron** method is designed to create a fully populated library **Patron** object. It copies the basic properties of the **Patron** and populates its **Loans** collection with detailed **Loan** objects.

4. Select the **GetPopulatedPatron** method.

   C# Copy

   ```csharp
   public Patron GetPopulatedPatron(Patron p)
   {
       Patron populated = new Patron
       {
           Id = p.Id,
           Name = p.Name,
           ImageName = p.ImageName,
           MembershipStart = p.MembershipStart,
           MembershipEnd = p.MembershipEnd,
           Loans = new List<Loan>()
       };

       foreach (Loan loan in Loans!)
       {
           if (loan.PatronId == p.Id)
           {
               populated.Loans.Add(GetPopulatedLoan(loan));
           }
       }

       return populated;
   }
   ```

5. Open an inline chat, and then enter a prompt that refactors the method using LINQ.

   Code Copy

   ```
   #selection refactor selection to `return new Patron` using LINQ
   ```

6. Take a minute to review the suggested update.

   The suggested update should look similar to the following code:

   C# Copy

```
public Patron GetPopulatedPatron(Patron p)
{
    return new Patron
    {
        Id = p.Id,
        Name = p.Name,
        ImageName = p.ImageName,
        MembershipStart = p.MembershipStart,
        MembershipEnd = p.MembershipEnd,
        Loans = Loans!
            .Where(loan => loan.PatronId == p.Id)
            .Select(GetPopulatedLoan)
            .ToList()
    };
}
```

Notice that a LINQ query is used to replace the foreach loop.

The LINQ code uses the object initializer to assign object properties to the new **Patron** object. This removes the requirement for a separate **populated** instance of the **Patron** object. Overall, the updated code is shorter and more readable.

The code uses the patron **p** to assign some basic properties to the new **Patron** object. Then it populates the **Loans** collection with loans that are associated with the Patron parameter **p**, transforming each loan using the **GetPopulatedLoan** method.

You can break down the LINQ code line that populates the **Loans** collection:

- **Loans!**: The **Loans!** expression accesses the **Loans** collection. The **!** operator is a null-forgiving operator, indicating that the developer is confident that **Loans** is not null. You should ensure that **Loans** is properly initialized before calling the **GetPopulatedPatron** method.
- **.Where(loan => loan.PatronId == p.Id)**: This code filters the loans to include only those that belong to the input patron **p**.
- **.Select(GetPopulatedLoan)**: This code transforms each filtered loan using the **GetPopulatedLoan** method.
- **.ToList()**: Converts the result to a **List<Loan>**.

7. To accept the suggested update, select **Accept**.

   You’re going to use this same approach to refactor three other methods.

8. Refactor the **GetPopulatedLoan**, **GetPopulatedBookItem**, and **GetPopulatedBook** methods using the same approach.

   For example, use the following prompts to refactor the three methods:

   For the **GetPopulatedLoan** method:

   Code Copy

```
 #selection refactor selection to `return new Loan` using LINQ. Use
`GetPopulatedBookItem` for the `BookItem` property. Use `Single` for BookItem and
Patron properties.
```

   For the **GetPopulatedBookItem** method:

Code Copy

```
#selection refactor selection to `return new BookItem` using LINQ. Use
`GetPopulatedBook` and `Single` for the `BookItem` property.
```

For the **GetPopulatedBook** method:

Code Copy

```
#selection refactor selection to `return new Book` using LINQ. Use `Where` and
`Select` for `Author` property. Use `First` author.
```

9. After accepting the suggested updates, take a minute to review your code changes.

   Your updated code should look similar to the following code:

   C# Copy

```
public Loan GetPopulatedLoan(Loan l)
{
    return new Loan
    {
        Id = l.Id,
        BookItemId = l.BookItemId,
        PatronId = l.PatronId,
        LoanDate = l.LoanDate,
        DueDate = l.DueDate,
        ReturnDate = l.ReturnDate,
        BookItem = GetPopulatedBookItem(BookItems!.Single(bi => bi.Id ==
l.BookItemId)),
        Patron = Patrons!.Single(p => p.Id == l.PatronId)
    };
}

public BookItem GetPopulatedBookItem(BookItem bi)
{
    return new BookItem
    {
        Id = bi.Id,
        BookId = bi.BookId,
        AcquisitionDate = bi.AcquisitionDate,
        Condition = bi.Condition,
        Book = GetPopulatedBook(Books!.Single(b => b.Id == bi.BookId))
    };
}

public Book GetPopulatedBook(Book b)
{
    return new Book
    {
        Id = b.Id,
        Title = b.Title,
        AuthorId = b.AuthorId,
        Genre = b.Genre,
        ISBN = b.ISBN,
        ImageName = b.ImageName,
        Author = Authors!.Where(a => a.Id == b.AuthorId).Select(a => new Author {
            Id = a.Id,
            Name = a.Name
        }).First()
    };
}
```

10. Use the **Explain** smart action to see an explanation of the LINQ queries.

    To open the **Explain** smart action, select code in the editor, right-click the selected code, select Copilot, and then select **Explain**. The **Explain** smart action provides a detailed explanation of the selected code. In this case, th LINQ queries used in the code.

    For example, you can use the **Explain** smart action on the **GetPopulatedBook** method to see an explanation of the LINQ query used to populate the **Author** property of the **Book** object.

    C# Copy

```
Author = Authors!.Where(a => a.Id == b.AuthorId).Select(a => new Author {
    Id = a.Id,
    Name = a.Name
}).First()
```

The **Explain** smart action provides a detailed explanation of the LINQ query used to populate the **Author** property of the **Book** object.

For example, the explanation might look like this:

Code Copy

```

 The active selection is a C# code snippet that assigns a value to the Author
property. This value is derived from a collection of Author objects named Authors.
The code uses LINQ to filter and transform the data within this collection.

 First, the Authors! expression uses the null-forgiving operator (!) to indicate that
Authors is not null, even if the compiler might think otherwise. This is a way to
suppress nullable warnings. The Where method is then called on the Authors collection
to filter the elements. The lambda expression a => a.Id == b.AuthorId is used to find
all Author objects where the Id matches the AuthorId property of another object b.

 After filtering, the Select method is used to project each filtered Author object
into a new Author object. This is done by creating a new instance of the Author class
and copying the Id and Name properties from the original Author object.

 Finally, the First method is called to retrieve the first element from the resulting
sequence. This means that the Author property will be assigned the first Author
object that matches the filter criteria and has been projected into a new Author
instance.

 This approach ensures that the Author property is set to a new Author object with
the same Id and Name as the first matching Author in the Authors collection.

```

11. Build your solution to ensure that there are no errors.

Refactor the JsonLoanRepository class using the Chat view in Edit mode

The JsonLoanRepository class includes the **GetLoan** and **UpdateLoan** data access methods. You'll refactor these two methods, replacing foreach loops with LINQ to improve code readability and maintainability.

Use the following steps to complete this section of the exercise:

1. Open the **JsonLoanRepository.cs** file.

2. Select the **GetLoan** method.

   The **GetLoan** method is designed to retrieve a loan by its ID.

   C# Copy

```
public async Task<Loan?> GetLoan(int id)
{
    await _jsonData.EnsureDataLoaded();

    foreach (Loan loan in _jsonData.Loans!)
    {
        if (loan.Id == id)
        {
            Loan populated = _jsonData.GetPopulatedLoan(loan);
            return populated;
        }
    }

    return null;
}
```

3. Ensure that the Chat view is open in **Edit** mode.

   If the Chat view isn’t open, select **Toggle Chat** set the mode to **Edit**.

4. Enter a prompt that refactors the method using LINQ.

   For example, enter the following prompt:

   Code Copy

   ```
    refactor the foreach using LINQ. Use Where, Select, and GetPopulatedLoan return the
   first matching loan.
   ```

5. Take a minute to review the suggested update.

   The suggested update should look similar to the following code:

   C# Copy

   ```
   public async Task<Loan?> GetLoan(int id)
   {
       await _jsonData.EnsureDataLoaded();

       return _jsonData.Loans!
           .Where(l => l.Id == id)
           .Select(l => _jsonData.GetPopulatedLoan(l))
           .FirstOrDefault();

   }
   ```

   The updated code uses LINQ to filter the loans collection to include only the loan with the specified ID. Notice that **loan** should be declared as nullable (**Loan? loan**). It then transforms the loan using the **GetPopulatedLoan** method and returns the first result. If no matching loan is found, **FirstOrDefault** returns **null**. The method then returns this loan object, which may be null if no loan with the specified **id** exists. This approach ensures that the returned loan is fully populated with all necessary related data, providing a comprehensive view of the loan record.

   GitHub Copilot could also suggest the following code, which is functionally equivalent:

   C# Copy

```
public async Task<Loan?> GetLoan(int id)
{
    await _jsonData.EnsureDataLoaded();

    Loan? loan = _jsonData.Loans!
        .Where(l => l.Id == id)
        .Select(l => _jsonData.GetPopulatedLoan(l))
        .FirstOrDefault();

    return loan;
}
```

6. To accept the updated GetLoan method, select **Keep**.

7. Select the **UpdateLoan** method.

C# Copy

```
public async Task UpdateLoan(Loan loan)
{
    Loan? existingLoan = null;
    foreach (Loan l in _jsonData.Loans!)
    {
        if (l.Id == loan.Id)
        {
            existingLoan = l;
            break;
        }
    }

    if (existingLoan != null)
    {
        existingLoan.BookItemId = loan.BookItemId;
        existingLoan.PatronId = loan.PatronId;
        existingLoan.LoanDate = loan.LoanDate;
        existingLoan.DueDate = loan.DueDate;
        existingLoan.ReturnDate = loan.ReturnDate;

        await _jsonData.SaveLoans(_jsonData.Loans!);

        await _jsonData.LoadData();
    }
}
```

8. Enter a prompt that refactors the method using LINQ.

   For example, enter the following prompt:

   Code Copy

```
 refactor selection using LINQ. find existing loan in `_jsonData.Loans!. replace
existing loan.
```

9. Take a minute to review the suggested update.

   The suggested update should look similar to one of the following examples:

C#

Copy

```csharp
public async Task UpdateLoan(Loan loan)
{
    var loans = _jsonData.Loans!;
    var index = loans.FindIndex(l => l.Id == loan.Id);

    if (index >= 0)
    {
        loans[index] = loan;
        await _jsonData.SaveLoans(loans);
        await _jsonData.LoadData();
    }
}
```

or

C#

Copy

```csharp
public async Task UpdateLoan(Loan loan)
{
    Loan? existingLoan = _jsonData.Loans!.FirstOrDefault(l => l.Id == loan.Id);

    if (existingLoan != null)
    {
        existingLoan.BookItemId = loan.BookItemId;
        existingLoan.PatronId = loan.PatronId;
        existingLoan.LoanDate = loan.LoanDate;
        existingLoan.DueDate = loan.DueDate;
        existingLoan.ReturnDate = loan.ReturnDate;

        await _jsonData.SaveLoans(_jsonData.Loans!);

        await _jsonData.LoadData();
    }
}
```

The updated code uses LINQ to find the existing loan in the loans collection. It then updates the existing loan with the new loan data. The method then saves the updated loans collection and reloads the data. This approach ensures that the loan data is updated correctly and that the changes are persisted to the data store.

You can also add the code to ensure the data is loaded before the method is executed:

C#

Copy

```
public async Task UpdateLoan(Loan loan)
{
    await _jsonData.EnsureDataLoaded();

    Loan? existingLoan = _jsonData.Loans!.FirstOrDefault(l => l.Id == loan.Id);

    if (existingLoan != null)
    {
        existingLoan.BookItemId = loan.BookItemId;
        existingLoan.PatronId = loan.PatronId;
        existingLoan.LoanDate = loan.LoanDate;
        existingLoan.DueDate = loan.DueDate;
        existingLoan.ReturnDate = loan.ReturnDate;

        await _jsonData.SaveLoans(_jsonData.Loans!);

        await _jsonData.LoadData();
    }
}
```

10. To accept the updated UpdateLoan method, select **Keep**.

11. Build your solution to ensure that no errors were introduced.

    You'll see warnings. You can ignore them for now.

Refactor the JsonPatronRepository class using the Chat view in Agent mode

The **JsonPatronRepository** class includes the following three methods:

- SearchPatrons: The SearchPatrons method is used to search for patrons by name. This method returns a sorted list of patrons.
- GetPatron: The GetPatron method is used to retrieve a patron by ID. This method returns a populated patron object.
- UpdatePatron: The UpdatePatron method is used to update a patron's information. This method updates the existing patron with the new data and saves the updated patrons collection.

Each of the three methods uses a foreach loop to iterate over the patrons and find matches based on the search input or ID.

You'll use the Chat view in Agent mode to refactor the methods, replacing foreach loops with LINQ queries, in the same way that you did for the **JsonData** and **JsonLoanRepository** classes.

Use the following steps to complete this section of the exercise:

1. Open the **JsonPatronRepository.cs** file.

   The **JsonPatronRepository** class is designed to manage library patrons.

2. Take a minute to review three methods included in the **JsonPatronRepository** class.

   The **SearchPatrons** method is designed to search for patrons by name.

   C# Copy

```
public async Task<List<Patron>> SearchPatrons(string searchInput)
{
    await _jsonData.EnsureDataLoaded();

    List<Patron> searchResults = new List<Patron>();
    foreach (Patron patron in _jsonData.Patrons)
    {
        if (patron.Name.Contains(searchInput))
        {
            searchResults.Add(patron);
        }
    }
    searchResults.Sort((p1, p2) => String.Compare(p1.Name, p2.Name));

    searchResults = _jsonData.GetPopulatedPatrons(searchResults);

    return searchResults;
}
```

Notice that the **SearchPatrons** method uses a foreach loop to iterate over the patrons and find matches based on the **searchInput** string. The method then sorts the results by name and returns a list of populated patrons.

The **GetPatron** method is designed to return the patron matching the specified **id**.

C# Copy

```
public async Task<Patron?> GetPatron(int id)
{
    await _jsonData.EnsureDataLoaded();

    foreach (Patron patron in _jsonData.Patrons!)
    {
        if (patron.Id == id)
        {
            Patron populated = _jsonData.GetPopulatedPatron(patron);
            return populated;
        }
    }
    return null;
}
```

Notice that the **GetPatron** method uses a foreach loop to iterate over the patrons and find a match based on the **id** parameter. The method then returns the populated patron object.

The **UpdatePatron** method is designed to update the patron with the specified **id**.

C# Copy

```
public async Task UpdatePatron(Patron patron)
{
    await _jsonData.EnsureDataLoaded();
    var patrons = _jsonData.Patrons!;
    Patron existingPatron = null;
    foreach (var p in patrons)
    {
        if (p.Id == patron.Id)
        {
            existingPatron = p;
            break;
        }
    }
    if (existingPatron != null)
    {
        existingPatron.Name = patron.Name;
        existingPatron.ImageName = patron.ImageName;
        existingPatron.MembershipStart = patron.MembershipStart;
        existingPatron.MembershipEnd = patron.MembershipEnd;
        existingPatron.Loans = patron.Loans;
        await _jsonData.SavePatrons(patrons);
        await _jsonData.LoadData();
    }
}
```

Notice that the **UpdatePatron** method uses a foreach loop to iterate over the patrons and find a match based on the **id** parameter. The method then updates the existing patron with the new data and saves the updated patrons collection.

3. In the Chat view, change the mode to **Agent**

   Agent mode is designed for running GitHub Copilot as an agent. You can use natural language to specify a high-level task. The agent will evaluate the assigned task, plan the work needed, and apply the changes to your codebase.

   Agent mode uses a combination of code editing and tool invocation to accomplish the task you specified. As it processes your request, it monitors the outcome of edits and tools, and iterates to resolve any issues that arise. If the agent is unable to resolve an issue, it will ask you to intervene. For example, if the agent uses several iterations working to resolve the same issue, it will pause the process and ask you to provide additional context to clarify your request or cancel the process.

   > **IMPORTANT**: When you use the Chat view in agent mode, GitHub Copilot may make multiple premium requests to complete a single task. Premium requests can be used by user-initiated prompts and follow-up actions Copilot takes on your behalf. The total number of premium requests used is based on the complexity of the task, the number of steps involved, and the model selected.

4. Take a minute to consider the task that you need to assign to the agent.

   The task is to refactor the **JsonPatronRepository** class. The goal is to replace the foreach loops with LINQ queries that produce the same result as the original foreach code.

   You can use your experience with the **JsonData** and **JsonLoanRepository** classes to help you write the task for the agent. The LINQ queries should use **Where**, **Select**, and **FirstOrDefault** to find matching patrons. The LINQ queries should also use **OrderBy** to preserve sorting in the original foreach code.

5. To assign the agent task, enter the following prompt:

Code Copy

```
 Review the LINQ code used in the JsonData and JsonLoanRepository classes. Notice how Where, Select, and FirstOrDefault are used. Refactor the methods in the JsonPatronRepository class, replacing foreach loops with LINQ queries that produce the same result as the original foreach code. Use OrderBy to preserve sorting in original foreach code. Use ! to suppress nullability warnings when accessing collections.
```

This prompt tells the agent to refactor the **JsonPatronRepository** class. It specifies that the foreach loops should be replaced with LINQ queries that produce the same result as the original foreach code. It also specifies that **OrderBy** should be used to preserve sorting in the original foreach code, and that **!** should be used to suppress nullability warnings when accessing collections.

6. Monitor the agent's progress as it refactors the code.

   Notice that the agent completes the task in several iterations. Each code edit pass is followed by a review pass that check for issues. If the agent encounters an issue, it will refactor the code to resolve the issue. If the agent is unable to resolve an issue, it will ask you to intervene.

7. Once the agent has finished, take a minute to review the suggested updates.

   The suggested update should look similar to the following code:

C# Copy

```
public async Task<List<Patron>> SearchPatrons(string searchInput)
{
    await _jsonData.EnsureDataLoaded();

    var searchResults = _jsonData.Patrons!
        .Where(patron => patron.Name.Contains(searchInput))
        .OrderBy(patron => patron.Name)
        .ToList();

    return _jsonData.GetPopulatedPatrons(searchResults);
}

public async Task<Patron?> GetPatron(int id)
{
    await _jsonData.EnsureDataLoaded();

    return _jsonData.Patrons!
        .Where(patron => patron.Id == id)
        .Select(patron => _jsonData.GetPopulatedPatron(patron))
        .FirstOrDefault();
}

public async Task UpdatePatron(Patron patron)
{
    await _jsonData.EnsureDataLoaded();

    var existingPatron = _jsonData.Patrons!.FirstOrDefault(p => p.Id == patron.Id);
    if (existingPatron != null)
    {
        existingPatron.Name = patron.Name;
        existingPatron.ImageName = patron.ImageName;
        existingPatron.MembershipStart = patron.MembershipStart;
        existingPatron.MembershipEnd = patron.MembershipEnd;
        existingPatron.Loans = patron.Loans;

        if (_jsonData.Patrons != null)
        {
            await _jsonData.SavePatrons(_jsonData.Patrons);
            await _jsonData.LoadData();
        }
    }
}
```

8. To accept all updates, select **Keep**.

Build and run the application

Now that you've refactored the code, it's time to build and run the application to ensure that everything is working correctly. You'll also test the application to ensure that the refactored code is functioning as expected.

1. To clean the solution, right-click **AccelerateAppDevGitHubCopilot**, and then select **Clean**.

   This action removes any build artifacts from the previous build. Cleaning the solution will effectively reset the JSON data files to their original values (in the output directory).

2. Ensure that the solution builds successfully.

For example, in the SOLUTION EXPLORER view, right-click **AccelerateDevGHCopilot**, and then select **Build**.

You'll see some Warnings, but there shouldn't be any Errors.

3. To run the application, right-click **Library.Console**, select **Debug**, and then select **Start New Instance**.

   The following steps guide you through a simple use case.

4. When prompted for a patron name, type **One** and then press Enter.

   You should see a list of patrons that match the search query.

   > **NOTE**: The application uses a case-sensitive search process.

5. At the "Input Options" prompt, type **2** and then press Enter.

   Entering **2** selects the second patron in the list.

   You should see the patron's name and membership status followed by book loan details.

6. At the "Input Options" prompt, type **1** and then press Enter.

   Entering **1** selects the first book in the list.

   You should see book details listed, including the due date and return status.

7. At the "Input Options" prompt, type **r** and then press Enter.

   Entering **r** returns the book.

8. Verify that the message "Book was successfully returned." is displayed.

   The message "Book was successfully returned." should be followed by the book details. Returned books are marked with **Returned: True**.

9. To begin a new search, type **s** and then press Enter.
10. When prompted for a patron name, type **One** and then press Enter.
11. At the "Input Options" prompt, type **2** and then press Enter.
12. Verify that first book loan is marked **Returned: True**.
13. At the "Input Options" prompt, type **q** and then press Enter.
14. Stop the debug session.

Summary

In this exercise, you learned how to refactor code using GitHub Copilot. You used the Chat view in Edit mode to refactor the **EnumHelper** class, replacing reflection with static dictionaries. You also used the inline chat and Edit mode to refactor the **JsonData** and **JsonLoanRepository** classes, replacing foreach loops with LINQ queries. Finally, you used the Agent mode to refactor the **JsonPatronRepository** class, replacing foreach loops with LINQ queries.

Clean up

Now that you've finished the exercise, take a minute to ensure that you haven't made changes to your GitHub account or GitHub Copilot subscription that you don't want to keep. If you made any changes, revert them now.