✓ 100 XP

Exercise - Refactor and improve code sections using GitHub Copilot tools

15 minutes

GitHub Copilot can help you update your code by suggesting changes that improve the code's quality, reliability, performance, and security.

In this exercise, you use GitHub Copilot to refactor and improve the following code sections:

- You refactor the EnumHelper class to use dictionaries instead of reflection. Using dictionaries improves performance by reducing the overhead of reflection. Eliminating reflection also improves code readability, maintainability, and security.
- You 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.

Use GitHub Copilot to refactor the EnumHelper class

Reflection is a powerful 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.

The existing EnumHelper class uses reflection to retrieve the description attribute of an enum value. You can refactor the EnumHelper class to use dictionaries instead of reflection. Using dictionaries can improve performance and eliminate any security concerns associated with using reflection.

In this exercise, you use GitHub Copilot to help you refactor the EnumHelper class.

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

1. Use the Solution Explorer view to open the EnumHelper.cs file.

   C#

```
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();
        }
    }
}
```

2. Open the Chat view, and then add the following files to the Chat context: `EnumHelper.cs`, `LoanExtensionStatus.cs`, `LoanReturnStatus.cs`, `MembershipRenewalStatus.cs`.

3. Enter the following prompt:

plaintext

```
@workspace I want to refactor the `EnumHelper` class using dictionaries rather than reflection to get enum description attributes. I want a separate dictionary for each enum. The enum values are in the `LoanExtensionStatus.cs`, `LoanReturnStatus.cs`, and `MembershipRenewalStatus.cs` files. Explain how to update the EnumHelper class to use dictionaries and provide the updated code.
```

This prompt asks GitHub Copilot to provide an explanation of how to update the EnumHelper class to use dictionaries instead of reflection. It also asks for the updated code that uses dictionaries for each enum value.

4. Take a minute to review the explanation provided by GitHub Copilot.

   The explanation should be similar to the following description:

   markdown

   ```markdown
   To refactor the `EnumHelper` class to use dictionaries for each enum, you need to create a dictionary for each enum type that maps the enum values to their descriptions. This will eliminate the need for reflection at runtime.

   Here's how you can update the `EnumHelper` class:

   1. Create dictionaries for each enum type.
   2. Populate these dictionaries with the enum values and their corresponding descriptions.
   3. Update the `GetDescription` method to use these dictionaries.

   Below is the updated code for the EnumHelper class:
   ```

   C#

   ```csharp
   using System.ComponentModel;
   using System.Collections.Generic;

   namespace Library.ApplicationCore.Enums
   {
       public static class EnumHelper
       {
           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." }
           };

           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<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." }
        };

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

            return value switch
            {
                MembershipRenewalStatus status =>
MembershipRenewalStatusDescriptions[status],
                LoanReturnStatus status =>
LoanReturnStatusDescriptions[status],
                LoanExtensionStatus status =>
LoanExtensionStatusDescriptions[status],
                _ => value.ToString()
            };
        }
    }
}
```

markdown

```markdown
This refactored EnumHelper class uses dictionaries to store the descriptions for each enum type. The GetDescription method now uses pattern matching to determine the type of the enum and retrieves the description from the appropriate dictionary. This approach avoids the use of reflection and improves performance.
```

ⓘ Note

If GitHub Copilot doesn't populate the dictionaries with the correct descriptions, ensure that the GitHub Copilot extension isn't requesting a restart. If it is, restart the extension and try the prompt again.

5. Use GitHub Copilot's suggested `EnumHelper` class to update your EnumHelper.cs file.

6. Ensure that your updated `EnumHelper` class is using the enum values from the `LoanExtensionStatus.cs`, `LoanReturnStatus.cs`, and `MembershipRenewalStatus.cs` files.

   Open each of the enum files and verify that the enum values in the dictionaries are correct. If there are discrepancies, have GitHub Copilot update the dictionaries for each enum individually. For example, you can use the following prompt for the `LoanExtensionStatus` enum:

   plaintext

   ```plaintext
   @workspace 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.
   ```

   If necessary, use the same approach for the `LoanReturnStatus` and `MembershipRenewalStatus` enums.

7. Take a minute to review the `GetDescription` method.

   C#

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

       return value switch
       {
           MembershipRenewalStatus status =>
   MembershipRenewalStatusDescriptions[status],
           LoanReturnStatus status =>
   LoanReturnStatusDescriptions[status],
           LoanExtensionStatus status =>
   LoanExtensionStatusDescriptions[status],
           _ => 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 is not found in the dictionary, the method returns the enum value as a string.

If you ask GitHub Copilot to refactor this code and eliminate the lambda expressions, it becomes easier to read:

C#

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

    switch (value)
    {
        case MembershipRenewalStatus status:
            return MembershipRenewalStatusDescriptions[status];
        case LoanReturnStatus status:
            return LoanReturnStatusDescriptions[status];
        case LoanExtensionStatus status:
            return LoanExtensionStatusDescriptions[status];
        default:
            return value.ToString();
    }
}
```

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

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

Update the data access methods to use LINQ

LINQ (Language Integrated Query) 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 methods in JsonData.cs to use LINQ.
- Refactor the methods in JsonLoanRepository.cs to use LINQ.
- Refactor the methods in JsonPatronRepository.cs to use LINQ.

Refactor the methods in JsonData.cs to use LINQ

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. Open the JsonData.cs file.

2. Select the `GetPopulatedPatron` method.

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

   cs

   ```cs
   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;
   }
   ```

3. Open an inline chat, and then enter a prompt that refactor's the method using LINQ.

   plaintext

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

4. Take a minute to review the suggested update.

The suggested update should look similar to the following code:

C#

```csharp
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()
    };
}
```

First off, a LINQ query is used to replace the `foreach (Loan loan in Loans!)` loop. With the LINQ code in place, you can use an object initializer to assign object properties to the new `Patron` object. This alleviates the need for a separate `populated` instance of the `Patron` object. Overall, the updated code is shorter and more readable.

The code uses the input 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 input Patron, 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>`.

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

Now that you've seen how to refactor the `GetPopulatedPatron` method, you can refactor the `GetPopulatedLoan`, `GetPopulatedBookItem`, and `GetPopulatedBook` methods to use LINQ as well.

6. 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:

   plaintext

   ```
   #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:

   plaintext

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

   For the `GetPopulatedBook` method:

   plaintext

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

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

   You updated code should look similar to the following code:

   C#

   ```
   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()
    };
}
```

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

   The **Explain** smart action provides a detailed explanation of the 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#

```
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:

plaintext

```plaintext
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 ob-
jects named Authors. The code uses LINQ (Language Integrated Query) 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 ele-
ments. The lambda expression a => a.Id == b.AuthorId is used to find
all Author objects where the Id matches the AuthorId property of an-
other 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.
```

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

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

Refactor the methods in JsonLoanRepository.cs to use LINQ

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#

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

   For example, enter the following prompt:

   plaintext

   ```
   #selection refactor selection using LINQ with `_jsonData.Loans!. Use
   `Where`, `Select` and `GetPopulatedLoan` to return `FirstOrDefault`.
   ```

4. Take a minute to review the suggested update.

   The suggested update should look similar to the following code:

   C#

   ```
   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;
   }
   ```

   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.

This code could be further simplified:

C#

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

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

}
```

5. To accept the updated GetLoan method, select **Accept**.

6. Select the `UpdateLoan` method.

C#

```
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();
    }
}
```

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

   For example, enter the following prompt:

   plaintext

   ```plaintext
   #selection refactor selection using LINQ find an existing loan `_json-
   Data.Loans!. Replace existing loan.
   ```

8. Take a minute to review the suggested update.

   The suggested update should look similar to the following code:

   C#

   ```C#
   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#

   ```C#
   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();
    }
}
```

9. To accept the updated UpdateLoan method, select **Accept**.

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

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

Refactor the methods in JsonPatronRepository to use LINQ

The `JsonPatronRepository` class includes the `SearchPatrons`, `GetPatron`, and `UpdatePatron` methods, and it's a great candidate for optimization. You'll refactor these 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 `JsonPatronRepository.cs` file.

2. Select the `SearchPatrons` method.

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

C#

```
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;
}
```

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

   For example, enter the following prompt:

   plaintext

   ```plaintext
   #selection refactor selection using LINQ with `_jsonData.Patrons!.
   Replace the loop with `Where`, `OrderBy`, and `GetPopulatedPatrons`.
   ```

4. Take a minute to review the suggested update.

   The suggested update should look similar to the following code:

   C#

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

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

       searchResults = _jsonData.GetPopulatedPatrons(searchResults);

       return searchResults;
   }
   ```

5. To accept the updated SearchPatrons method, select **Accept**.

6. Select the `GetPatron` method.

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

   C#

```
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;
}
```

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

   For example, enter the following prompt:

   plaintext

   ```
   #selection refactor selection using LINQ with `_jsonData.Patrons!. Use
   `Where`, `Select` and `GetPopulatedPatron` to return `FirstOrDefault`
   ```

8. Take a minute to review the suggested update.

   The suggested update should look similar to the following code:

   C#

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

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

       return patron;
   }
   ```

   This code could be simplified further:

   C#

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

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

9. To accept the updated GetPatron method, select **Accept**.

10. Select the `UpdatePatron` method.

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

    C#

```csharp
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();
    }
}
```

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

    For example, enter the following prompt:

    plaintext

```plaintext
#selection refactor selection using LINQ to find `patron` in `_jsonData.Patrons!. Replace existing patron with `patron`.
```

12. Take a minute to review the suggested update.

    The suggested update should look similar to the following code:

    C#

    ```csharp
    public async Task UpdatePatron(Patron patron)
    {
        await _jsonData.EnsureDataLoaded();
        var patrons = _jsonData.Patrons!;
        var existingPatron = 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;
            await _jsonData.SavePatrons(patrons);
            await _jsonData.LoadData();
        }
    }
    ```

13. To accept the updated UpdatePatron method, select **Accept**.

Check your work

Use the following steps to check your work:

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 during (in the output directory).

2. Build the application and ensure there are no errors.

3. Run the application.

   You can run the application from the Solution Explorer view by right-clicking the `Library.Console` project, selecting **Debug**, and then selecting **Start New Instance**.

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

5. At the "Matching Patrons" prompt, type **2** and then press Enter.

6. At the "Book Loans" prompt, type **1** and then press Enter.
7. At the "Input Options" prompt, type **r** and then press Enter.
8. Verify that the message "Book was successfully returned." is displayed.
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 "Matching Patrons" 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 application.

Next unit: Module assessment

< **Previous** Next >