Задание №8. Сортировка и поиск

LINQ-запросы

Одна из первых тем, которые изучают студенты при обучении программированию, это написание алгоритмов сортировки и поиска. Это хорошее обучающее задание даёт понимание принципов обработки данных. Однако в профессиональной деятельности разработчику нет необходимости создавать собственные методы поиска и сортировки, так как стандартные библиотеки в любом современном языке программирования предоставляют готовые реализации.

Одним из подходов к обработке данных в языке C# являются LINQ-запросы. LINQ-запросы имеют специальный синтаксис, который стремится повторить SQL-запросы в базах данных. Применить LINQ-запросы можно к любой стандартной коллекции – массиву, списку, словарю и другим.

Наиболее стандартные LINQ-запросы:

- Where() – позволяет получить из коллекции все элементы, удовлетворяющие некоторому условию.
- Select() – позволяет создать новую коллекцию нового типа данных на основе данных исходной коллекции.
- OrderBy() – выполняет упорядочивание коллекции согласно некоторому условию.
- All() – определяет, удовлетворяют ли все элементы коллекции некоторому условию.
- Any() – определяет, удовлетворяет хотя бы один элемент коллекции некоторому условию.
- Contains() – определяет, содержит ли коллекция элемент, удовлетворяющий некоторому условию.
- First() – возвращает первый элемент коллекции, удовлетворяющий заданному условию.
- ToList() – преобразовать коллекцию в список.
- ToArray() – преобразовать коллекцию в массив.

Допустим, у нас есть класс Book:

```
public class Book
{
```

```
      public string Title {get;set;}
      public string Author {get;set;}
      public int PagesCount {get;set;}
      public Book(string title, string author, int pagesCount);
}
```

Создадим в клиентском коде коллекцию книг:

```
var books = new List<Book>
{
      new Book(“Алиса в стране чудес”, “Керролл Л.”, 120),
      new Book(“Мартин Иден”, “Лондон Д.”, 256),
      new Book(“Приключения Шерлока Холмса”, “Дойл А.К.”, 412),
      new Book(“Марсианские хроники”, “Брэдбери Р.”, 378),
      new Book(“Преступление и наказание”, “Достоевский Ф.М.”, 291),
}
```

Теперь рассмотрим примеры вызова различных LINQ-запросов:

```
// метод вернет новый List<Book>, в котором будут только те книги, в которых
количество страниц больше 300.
var whereBooks1 = books.Where(book => book.PagesCount > 300).ToList();

// метод вернет новый List<Book>, в котором хранятся книги, названия которых
содержат слово “Алиса”.
var whereBooks2 = books.Where(book => book.Title.Contains(“Алиса”)).ToList();

// метод вернет новый List<string>, состоящий только из имен авторов в том же
порядке, в котором они были в исходном списке.
var selectBooks = books.Select(book => book.Author).ToList();

// вернет новый List<Book>, где книги отсортированы по названию в алфавитном
порядке
var orderByBooks = books.OrderBy(book => book.Title).ToList();

// вернет true, если в исходной коллекции все книги больше 300 страниц, иначе
false. Удобно использовать в условиях if
var allBooks = books.All(book => book.PagesCount > 300);

// вернет true, если в исходной коллекции есть хоть одна книга больше 300 страниц,
иначе false. Удобно использовать в условиях if
var anyBooks = books.Any(book => book.PagesCount > 300);

// вернет true, если в исходной коллекции есть хоть одна книга, заголовок которой
больше 20 символов. Удобно использовать в условиях if
var containsBooks = books.Contains(book => book.Title.Length > 20);

// вернет первую книгу коллекции, название которой содержит слово “Приключения”,
а количество страниц больше 200. Если такой книги нет, то сгенерируется исключение.
var firstBook = books.First(book =>
        book.Title.Contains(“Приключения”) && book.PagesCount > 200);
```

Любой LINQ-запрос занимается перебором исходной коллекции и проверяет, подходит ли он некоторому заданному условию. Условие может быть любое, в том числе и составное (как показано для вызова метода First()). В LINQ-запрос передается делегат метода, который реализует условие. Название переменной book условное и может быть любым – оно нужно лишь для того, чтобы после оператора => можно было обратиться к объекту из коллекции и задать для него условие поиска или сортировки. Чаще всего для имени такой переменной используют t, хотя в случае сложных составных LINQ-запросов рекомендуется использовать более понятное имя переменной, чтобы разработчик сразу мог понять, объекты какого типа перебираются в коллекции.

Сами методы Where(), Select(), OrderBy() возвращают не готовый список, а лишь ***запрос*** для получения списка. Чтобы получить результат, в конце после вызова метода вызывается метод ToList() или ToArray().

Главное преимущество LINQ-запросов, что их можно объединять в одну цепочку, наподобие SQL-запросов в БД. Например:

```
// сначала вернется новая коллекция книг с количеством страниц более 300, а затем
из новой коллекции вернется список авторов этих книг.
var whereBooks1 = books.
      Where(book => book.PagesCount > 300).
      Select(book => book.Author).
      ToList();
```

Можно сделать сколько угодно длинную цепочку вызовов LINQ-запросов, при этом результат одного метода будет тут же передаваться на вход следующего. Таким образом, можно делать достаточно сложные обработки данных в лаконичной форме.

При создании цепочек LINQ-запросов следует учитывать сложность алгоритмов. Например последовательность Where().OrderBy() будет отрабатывать быстрее, чем OrderBy().Where(), так как в первом случае сначала из исходной коллекции будет отработана часть коллекции и потом отсортирована, а во втором случае вся исходная коллекция сначала будет отсортирована, а затем из неё будет выбрана часть коллекции. Результат выполнения обоих запросов будет одинаковый, однако сортировка всей исходной коллекции займет больше времени, чем сортировка только уже найденных объектов.

Описание задания:

1. Создайте новую ветку в локальном репозитории под названием «features/8_add_data_filtering». Перейдите в новую ветку.

1. Добавьте в класс Project открытые методы сортировки или поиска в зависимости от вашего варианта заданий. Например, для варианта NoteApp необходимо сделать сортировку заметок по дате редактирования и поиск по категории; для варианта ContactApp необходимо сделать сортировку контактов по алфавиту и поиск по дате рождения для вывода всех именинников и т.д.
2. В главном окне реализуйте правильный вызов созданных методов. Обеспечьте вывод списка объектов пользователю упорядоченных данных согласно вашему варианту.
3. Реализуйте логику поиска/категоризации данных в вашем списке. В зависимости от выбора пользователя, покажите ему найденные данные (заметки заданной категории; контакты, содержащие подстроку).
4. При создании функции поиска или сортировки нужно учитывать, что список отображаемых пользователю объектов и список объектов в поле _project могут отличаться. Например, если пользователь варианта NoteApp выбрал отображать заметки только определенной категории, то в списке на экране должны отображаться только те заметки, которые совпадают с выбранной категорией. При этом в поле _project всё равно должны храниться все заметки, которые пользователь вводил ранее. Это может стать проблемой для функции удаления данных и функции редактирования – пользователь выбирает в списке на экране одну заметку определенной категории, но её SelectedIndex не будет совпадать с индексом заметки в поле _project. Необходимо придумать механизм, с помощью которого для каждой выбранной заметки (или контакта, или любого другого объекта в списке) можно будет однозначно установить объект в исходном списке объектов _project.
5. Для создания такого соответствия можно завести в главном окне новое закрытое поле List<***Object***> _current***Objects*** (под ***Object*** подразумевается название типа данных, который соответствует вашему варианту). В нём должны хранится те объекты из _project, которые сейчас отображаются в списке на экране пользователя, в то время как в _project будут храниться все объекты, вне зависимости от того, отображаются ли они сейчас на экране или нет. Таким образом, выбранная строка SelectedIndex для ListBox будет совпадать с редактируемым/удаляемым объектом в _currentObjects. Получив объект из _currentObjects по индексу SelectedIndex, можно будет найти его индекс уже внутри _project.
6. Подумайте над тем, как правильно организовать работу с _currentObjects в главном окне: в какой момент этот список должен обновляться и как это повлияет на работу уже существующих методов, таких как AddObject(), EditObject(), RemoveObject().
7. Проверьте правильность работы приложения – как новой функциональности, так и всей ранее созданной. Часто студенты не замечают, что при добавлении поиска или сортировки добавление, редактирование или удаление данных начинает работать некорректно.

8. В Visual Studio перейдите в ветку develop. Затем выполните слияние (merge) ветки features/8_add_data_filtering в ветку develop. Теперь все изменения из ветки с добавленным решением должны переместиться в ветку develop. Убедитесь в этом с помощью GitHub.