Задание №7.
Делегаты

Цели задания:

- Изучить применение делегатов в C#.

Делегаты (delegate) – это некоторые объекты, которые хранят ссылки на методы [1]. Аналогом делегатов в Си++ являются указатели на функции. Сами по себе указатели на функции относятся не к объектно-ориентированному программированию, а являются элементом структурного программирования. Однако, комбинации указателей на функции и ООП-механик позволяют создавать куда более сложные архитектурные решения.

Для объявления делегата используется ключевое слово delegate, после чего необходимо указать уникальное название и сигнатуру методов (набор входных аргументов и возвращаемый тип данных), которые смогут храниться в делегате:

```
delegate <возвращаемый тип> <уникальное имя>(<аргументы>);
```

Например:

```
delegate void DoSomething();
delegate void DoSomething2(int a, int b);
delegate bool DoSomething3(double[] values);
```

Здесь мы объявили три делегата. В первом могут храниться ссылки на методы, у которых нет входных аргументов и нет возвращаемого значения. Во втором могут храниться ссылки на методы, у которых есть два целочисленных аргумента. В третьем – у которых на вход подается массив вещественных чисел, и обязательно возвращается булево значение.

Делегаты, как и типы данных, должны объявляться вне других типов данных. После того, как мы объявили делегат, мы можем создавать переменные делегата:

```
delegate void DoSomething();

public class Program
{
   public void main()
   {
      // Объявление переменной делегата DoSomething
      DoSomething do;
   }
}
```

Переменные делегатов создаются аналогично обычным переменным – в начале указание имени делегата, затем через пробел указывается уникальное имя переменной. Как и обычные переменные, переменные делегата можно объявить внутри какого-либо метода или можно сделать поле внутри класса типа делегата.

Но, чтобы инициализировать делегат, нам нужен метод с подходящей сигнатурой.

```
delegate void DoSomething();

public class Program
{
   private void WriteHello()
   {
      Console.WriteLine(“Hello, World!”);
   }

   public void main()
   {
      // Объявление переменной делегата DoSomething
      DoSomething do;

      // Инициализация делегата
      do = WriteHello;

      // Вызов делегата
      do();
   }
}
```

В примере мы создали метод WriteHello(), который не принимает входных аргументов и не возвращает значение, как и делегат DoSomething. Поэтому в методе main мы можем присвоить ссылку на этот метод в переменную делегата:

```
do = WriteHello;
```

Обратите внимание, что здесь обращение к методу WriteHello идет без скобок:

```
do = WriteHello;   // присвоение ссылки на метод в делегат

do = WriteHello(); // вызов метода WriteHello() и
                   // попытка присвоить результат вызова
                   // в делегат – разумеется, это ошибка
```

После инициализации делегата мы можем вызывать метод WriteHello через нашу переменную do:

```
// Вызов делегата в методе main()
do();
```

Вызов осуществляется также, как и любой метод – пишется название делегата, а в круглых скобках передаются входные аргументы. Если делегат, имеет возвращаемое значение, то результат можно присвоить в другую переменную:

```
delegate bool DoSomething(double[] values);

public class Program
{
   private bool WriteArray(double[] values)
   {
      var strings = values.Select(t => t.PadRight(5)).ToList;
      var formattedOutput = string.Join(strings, “ ”);
      Console.WriteLine(formattedOutput);
      return true;
   }

   private bool IsOnlyPositiveValues(double[] values)
   {
      foreach (var value in values)
      {
         if (value < 0.0)
```

```
            return false;
        }

        return true;
    }

    public void main()
    {
        var values = new double[]{1.6, 5.0, -7.9, 12.2, 3.5}

        // Объявление переменной делегата DoSomething
        DoSomething do;

        // Инициализация делегата
        do = WriteArray;

        // Вызов делегата
        var result = do(values);
        Console.WriteLine($“Делегат вернул {result}”)

        // Присвоение в делегат другого метода
        do = IsOnlyPositiveValues;

        result = do(values);
        Console.WriteLine($“Делегат вернул {result}”)
    }
}
```

Как показано в примере, мы можем присвоить в делегат ссылку на другой метод. Если мы хотим очистить делегат от ссылки, мы можем присвоить в него значение null. Но вызов делегата со значением null приведет к выбросу исключения.

Как и с другими переменными, в C# для делегатов выполняется строгая проверка типов данных. В делегат можно присвоить ссылку на метод только заданной сигнатуры. Если сигнатура метода отличается количеством аргументов, их типами или порядком, или возвращаемым типом, то присвоить ссылку на такой метод в делегат не удастся. Если две переменные являются переменными делегата одного типа, то мы можем копировать значения из одной переменной в другой. Модификаторы методов (public, private, static) для делегата значения не имеют.

Фактически, делегаты позволяют работать с методами как с обычными переменными. Из примера выше преимущество делегатов не очевидно – зачем нужен делегат, если мы можем вызвать сам метод?

Преимущество заключается в том, что переменные делегатов могут передаваться в методы в качестве входных аргументов и вызываться как часть алгоритма. С точки зрения алгоритмизации это означает, что с помощью делегатов мы можем подменять часть алгоритма на любую функцию, тем самым получая разный результат. Рассмотрим на примере сортировки. Предположим, что мы разработали класс, который отвечает за сортировку. По заданию нам нужно две сортировки – по возрастанию и убыванию:

```
// Класс сортировщик
public static class Sorter
{
    // Сортирует массив по возрастанию
    public static void SortAscend(double[] values)
    {
        for (int i = 0; i < values.Length; i++)
        {
            for (int j = 1; j < values.Length; j++)
            {
                if (values[j] < values[j - 1])
                {
                    double temp = values[j];
                    values[j] = values[j - 1];
                    values[j - 1] = temp;
                }
            }
        }
    }

    // Сортирует массив по убыванию
    public static void SortDescend(double[] values)
    {
        for (int i = 0; i < values.Length; i++)
        {
            for (int j = 1; j < values.Length; j++)
            {
                if (values[j] > values[j - 1])
                {
                    double temp = values[j];
```

```
                    values[j] = values[j - 1];
                    values[j - 1] = temp;
                }
            }
        }
    }
}
```

Если мы посмотрим, то реализация обоих методов одинаковая, за исключением одной строки, а конкретнее, условия. В остальном методы полностью дублируются. Это важно по двум причинам:

- сортировка полным перебором – одна из самых простых. Более эффективные методы сортировки занимают в разы больше кода, что усугубляет проблему дублирования.
- если речь идет о сортировке не вещественных чисел, а, например, товаров Item, то товары могут быть отсортированы по каждому полю: по имени, стоимости, категории, рейтингу, весу, габаритам и т.п. Чем больше данных хранится в классе или структуре, тем больше способов упорядочить эти данные у нас есть. А, следовательно, во столько же раз мы дублируем реализацию сортировки. Если в классе пять полей, то количество дублирований будет равно 10. Абсолютно неприемлемое решение для разработки.

Делегаты позволяют нам организовать код таким образом, что условие сравнения будет вынесено в отдельный метод, который будет передаваться в метод сортировки. Для начала, выделим сравнение двух чисел в условии в отдельные методы:

```
// Класс сортировщик
public static class Sorter
{
    // Сравнение двух чисел для сортировки по возрастанию
    public static bool CompareAscending(double x1, double x2)
    {
        return x1 < x2;
    }

    // Сравнение двух чисел для сортировки по убыванию
    public static bool CompareDescending(double x1, double x2)
    {
        return x1 > x2;
```

```
    }

    // Сортирует массив по возрастанию
    public static void Sort(double[] values)
    {
        for (int i = 0; i < values.Length; i++)
        {
            for (int j = 1; j < values.Length; j++)
            {
                if (CompareAscending(values[j], values[j - 1]))
                {
                    double temp = values[j];
                    values[j] = values[j - 1];
                    values[j - 1] = temp;
                }
            }
        }
    }

    // Сортирует массив по убыванию
    public static void SortDescend(double[] values)
    {
        for (int i = 0; i < values.Length; i++)
        {
            for (int j = 1; j < values.Length; j++)
            {
                if (CompareDescending(values[j], values[j - 1]))
                {
                    double temp = values[j];
                    values[j] = values[j - 1];
                    values[j - 1] = temp;
                }
            }
        }
    }
}
```

Теперь создадим общий делегат, который может хранить методы сравнения чисел:

```
delegate bool CompareValues(double x1, double x2);
```

Последний шаг, это передача делегата в метод сортировки в качестве входного аргумента и вызов его в условии:

```
public delegate bool CompareValues(double x1, double x2);

// Класс сортировщик
public static class Sorter
{
    // Сравнение двух чисел для сортировки по возрастанию
    public static bool CompareAscending(double x1, double x2)
    {
        return x1 < x2;
    }

    // Сравнение двух чисел для сортировки по убыванию
    public static bool CompareDescending(double x1, double x2)
    {
        return x1 > x2;
    }

    // Сортирует массив по принципу,
    // который передадут в делегате compare
    public static void Sort(double[] values,
                            CompareValues compare)
    {
        for (int i = 0; i < values.Length; i++)
        {
            for (int j = 1; j < values.Length; j++)
            {
                if (compare(values[j], values[j - 1]))
                {
                    double temp = values[j];
                    values[j] = values[j - 1];
                    values[j - 1] = temp;
                }
            }
        }
    }
}
```

Теперь у нас нет дублирования сортировки. Есть одна реализация сортировки, где принцип упорядочивания задается извне:

```
public static main()
{
    var values = new double[] { 9.7, 9.3, 2.5, 5.6,
                                6.0, 2.1, 6.5, 3.0 };

    // Вызов сортировки по возрастанию
    Sorter.Sort(values, Sorter.CompareAscending);

    // Вызов сортировки по убыванию
    Sorter.Sort(values, Sorter.CompareDescending);
}
```

Если нам понадобится сортировка по другому принципу, то нам не нужно писать новый метод сортировки. Нам достаточно написать один метод сравнения чисел и использовать уже готовую сортировку. Важно, что новый метод сравнения чисел необязательно должен быть внутри класса Sorter, а может быть задан извне. Более того, механизм лямбда-выражений [2] позволяет не создавать отдельные методы сравнения, а писать принцип сравнения непосредственно в вызове метода сортировки:

```
// Вызов сортировки по возрастанию
Sorter.Sort(values, Sorter.CompareAscending);

// Вызов сортировки по убыванию
Sorter.Sort(values, Sorter.CompareDescending);

// Вызов сортировки по возрастанию с помощью лямбда-выражения
Sorter.Sort(values, (x1, x2) => { return x1 < x2; });

// Вызов сортировки по модулю с помощью лямбда-выражения
Sorter.Sort(values,
         (x1, x2) => { return Math.Abs(x1) < Math.Abs(x2); });
```

Поэтому делегаты так и называются. Делегаты позволяют ***делегировать*** выбор принципа сравнения чисел клиентскому коду, в то время как класс сортировки решает строго задачу сортировки.

Как итог:

- мы избавились от дублирования методов сортировки;

- если мы захотим поменять реализацию сортировку на более эффективный алгоритм, это можно будет сделать исправлением одного метода, а не множества скопированных методов;
- клиентский код может сам определять по какому принципу ему нужно упорядочить данные – расширяемость решения в разы выше.

Разумеется, разработчики уже давно не пишут методы сортировки самостоятельно, а используют готовую реализацию из стандартных библиотек, например, сортировку LINQ-запросами. Но LINQ-запросы также работают с делегатами. Все методы Order(), Select(), Where() и другие – все принимают на вход ссылку на метод или лямбда-выражение, за счёт чего LINQ-запросы и становятся такими удобными в использовании.

Если идею делегатов в классе сортировки скрестить с идеей обобщенных типов [3], то мы получаем класс, который умеет сортировать данные любого типа по любому заданному критерию. Если в качестве аргументов делегата передавать ссылки на интерфейс или базовый класс, мы получаем сортировщик полиморфных коллекций. Кроме того, делегаты могут храниться внутри класса в качестве полей. Делегаты могут храниться в списках или словарях, а сами делегаты могут хранить ссылки сразу на несколько методов (групповая адресация). Другими словами, делегаты значительно расширяют возможности разработчика при разработке алгоритмов и проектировании приложений. Делегаты хорошо комбинируются с другими техниками, которые были изучены ранее. Еще один вариант применения делегатов рассмотрен в [4].

В подавляющем большинстве случаев разработчики C# не объявляют собственные делегаты, а используют встроенные делегаты Action<>, Func<>, Predicate<> [5]. Это обобщенные делегаты, которые работают с любыми типами данных и разным количеством входных аргументов. Например, вместо создания собственного делегата `bool CompareValues(double x1, double x2)`, можно использовать встроенный делегат Func<double, double, bool>. В этом случае, реализация метода сортировки изменится следующим образом:

```
// Сортирует массив по принципу,
// который передадут в делегате compare
public static void Sort(double[] values,
                        Func<double, double, bool> compare)
{
    for (int i = 0; i < values.Length; i++)
    {
        for (int j = 1; j < values.Length; j++)
        {
```

```
            if (compare(values[j], values[j - 1]))
            {
                double temp = values[j];
                values[j] = values[j - 1];
                values[j - 1] = temp;
            }
        }
    }
}
```

То есть изменение заключается в типе входного аргумента, остальная реализация остается прежней. Так как Func<double, double, bool> описывает методы, принимающие два вещественных числа и возвращающие булево значение, то все ранее написанные методы сравнения и лямбда-выражения будут работать и с этим делегатом. Объявление собственного делегата из программы можно удалить.

Func<> является обобщенным и самым универсальным делегатом. Им можно заменить любые собственные делегаты. Встроенные делегаты Action<> и Predicate<> являются частными случаями и отличаются от Func<> тем, что в них предопределен возвращаемый тип. Action<> используется для методов, которые ничего не возвращают, а Predicate<> используется для методов, которые возвращают булево значение. Например, в коде выше можно заменить Func<double, double, bool> на Predicate<double, double>.

Несмотря на встроенные делегаты, объявление собственных делегатов может использоваться для повышения читаемости. Например, если вам в исходном коде приходится часто передавать делегат со сложными входными аргументами типа:

```
Func<Dictionary<string, List<Item>>, Dictionary<DateTime,
Order>, List<string>>
```

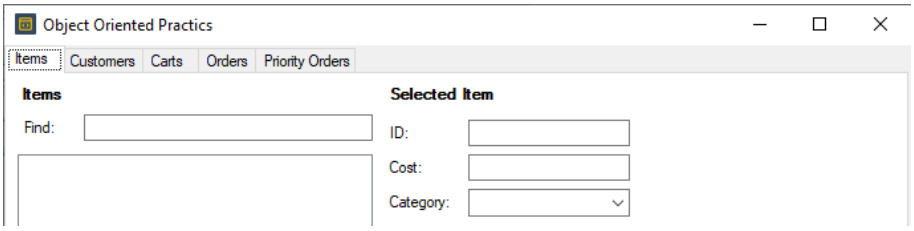
вы можете объявить делегат

```
delegate List<string> ProcessOrders(Dictionary<string,
List<Item>>, Dictionary<DateTime, Order>);
```

а затем передавать делегат в методы по простому имени ProcessOrders вместо длинного и сложно читаемого Func<...>. В остальных случаях используйте встроенные делегаты.

Создание алгоритмов на основе делегатов

1. Создать из ветки develop ветку Tasks/7_delegates, перейти в новую ветку.
2. Добавьте в подпапку проекта Services новый класс DataTools, который будет реализовывать разные методы по обработке данных.
3. Реализуйте в сервисном классе DataTools метод фильтрации, который принимает на вход список товаров и возвращает новый список товаров, стоимость которых выше 5000. Использовать LINQ-запросы для реализации в данном задании нельзя.
4. Реализуйте в сервисном классе DataTools метод фильтрации, который принимает на вход список товаров и возвращает новый список товаров, которые относятся к какой-либо категории (например, первой категории из вашего перечисления).
5. Проанализируйте код ваших методов фильтрации и придумайте, как вынести принцип фильтрации в делегат.
6. Объявите делегат для методов фильтрации. Вынесите критерии фильтрации в отдельные открытые методы. Избавьтесь от дублирования методов, оставив только один метод фильтрации, в который передается делегат критерия.
7. Протестируйте работу метода фильтрации на тестовых данных. Убедитесь, что фильтрация работает верно.
8. Добавьте на вкладку Items над ListBox с продуктами поисковую строку:



9. Реализуйте работу поисковой строки. Если в текстовом поле есть текст, тогда с помощью метода фильтрации из класса DataTools должны находиться все продукты, в имени которых есть введенная подстрока. Для этого в классе ItemsTab добавьте метод критерия фильтрации или передавайте его в качестве лямбда-выражения. Все найденные товары должны показываться в ListBox. Если пользователь сотрёт текст в текстовом поле, то снова отображаются все существующие продукты.
10. Заметьте, что теперь индекс выбранного в ListBox продукта Item может не совпадать с индексом продукта из общего списка продуктов Items – это может привести к тому, что при выборе отфильтрованного продукта в ListBox на правой панели будет показываться и редактироваться совершенно другой продукт. Решить проблему можно с помощью, например, создания отдельного списка

List<Item> _displayedItems и вызова метода IndexOf(). Протестируйте приложение и убедитесь, что выбор редактируемого продукта для полного списка продуктов и для отфильтрованного работает одинаково.

11. Замените собственный тип делегата на встроенный Func<>. Протестируйте работу фильтрации еще раз.

12. Аналогично методу фильтрации, добавьте в класс DataTools метод сортировки товаров по внешнему делегату – делегат принимает список товаров и возвращает новый упорядоченный список. В качестве способов упорядочения реализуйте – сортировку по имени, по возрастанию стоимости, по убыванию стоимости.

13. Протестируйте работу алгоритма сортировки на тестовых данных.

14. Добавьте на вкладку Items выпадающий список с выбором способа упорядочивания продуктов:

15. Реализуйте логику вкладки таким образом, чтобы по умолчанию при запуске приложения выполнялась сортировка по имени продукта. Если пользователь выберет в выпадающем списке другой способ, товары должны упорядочиться согласно выбранному способу (выбранный товар на правой при этом не должен сброситься).

16. Упорядочивание товаров должно учитывать текст, введенный в поисковую строку. То есть, если пользователь меняет способ упорядочивания на другой, то отфильтрованные товары не должны сброситься до полного списка. Если пользователь поменяет текст в поисковой строке, способ упорядочивания не должен сброситься на поиск по имени.

17. Тщательно протестируйте приложение, убедитесь, что вся функциональность работает корректно.

18. Проверьте оформление кода, именование элементов кода и пользовательского интерфейса. Проверьте наличие комментариев, сделайте коммит.

19. Как указывалось в теоретической части, в реальных приложениях такие сервисные классы для поиска и сортировки, как DataTools, не разрабатываются – вместо них используются LINQ-запросы. Однако, создание подобного класса является хорошим упражнением в создании и использовании делегатов.

Проектная документация

1. По результатам всех заданий нарисуйте новую диаграмму классов.
2. На ней должны быть отражены новые классы, а также связи между ними.
3. Объявленные типы делегатов не имеют специального отображения на диаграммах классов. Они просто указываются в качестве типов данных для входных и выходных переменных для методов или полей внутри описания класса.
4. Обратите внимание, что новые поля и свойства добавились в ранее написанные классы – изменения должны быть показаны на диаграмме.
5. Диаграмму сохраните под названием «Practics7.*» в папке doc. Сохраните диаграмму как в формате программы, которую вы будете использовать, так и в формате png или jpeg, чтобы диаграмму можно было просмотреть онлайн через GitHub.
6. Выполнить сливание текущей ветки с веткой develop.

Дополнительные задания

1. Реализуйте в приложении также сортировку и поиск для заказов на вкладке Orders и для покупателей на вкладке CustomersTab. Для заказов реализуйте поиск по статусу заказа, покупателю или адресу. Наилучшая реализация – это реализация с единой поисковой строкой. То есть пользователю достаточно ввести любую подстроку в поиск, и в таблице будут отображаться все товары, которую соответствуют запросу, вне зависимости от того, в каком свойстве заказа была найдена подстрока – номере заказа, имени покупателя, адресе, статусе или общей стоимости. Заметьте, что найденные заказы не должны дублироваться. Например, если введенная подстрока была найдена и в имени покупателя, и в адресе доставки, то в таблице этот заказ должен быть показан только один раз, а не по количеству найденных совпадений.
2. Попробуйте сделать методы класса DataTools обобщенными – методы поиска и сортировки должны работать с любыми типами данных, а не только с классом Item.