Тема: «Делегаты, события и лямбды»

Цель: познакомиться с использованием делегатов в приложениях. Научится описывать собственные события. Познакомится с механизмом обработки событий.

Время выполнения: 8 часов.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Делегаты

Делегаты предоставляют механизм позднего связывания в .NET. Позднее связывание означает, что создается алгоритм, где вызывающий объект также предоставляет по крайней мере один метод, который реализует часть алгоритма.

Ключевое слово *"delegate"* – это основной элемент, используемый при работе с делегатами. Код, который компилятор создает при использовании ключевого слова *delegate*, будет сопоставляться с вызовами методов, вызывающих члены классов *Delegate* и *MulticastDelegate*.

Тип делегата определяется с помощью синтаксиса, подобному синтаксису определения сигнатуры метода. К определению нужно просто добавить ключевое слово *delegate*.

В качестве примера будем использовать метод *List.Sort()*. Первым шагом является создание типа для делегата сравнения.

```
// From the .NET Core library
// Define the delegate type:
public delegate int Comparison<in T>(T left, T right);
```

Компилятор создает класс, производный от *System.Delegate*, соответствующий используемой сигнатуре (в данном случае – метод, который возвращает целое число и имеет два аргумента). Тип делегата – *Comparison*. Тип делегата *Comparison* является универсальным типом.

Обратите внимание, что синтаксис может отображаться так, будто он объявляет переменную, но на самом деле он объявляет тип. Можно определить типы делегатов внутри классов, непосредственно внутри пространств имен или даже в глобальном пространстве имен.

Примечание

Не рекомендуется объявлять типы делегатов (или другие типы) непосредственно в глобальном пространстве имен.

Компилятор также создает обработчики добавления и удаления для этого нового типа, чтобы клиенты этого класса могли добавлять и удалять методы из списка вызовов экземпляра. Компилятор будет обеспечивать соответствие подписи добавляемого или удаляемого метода подписи, используемой при объявлении метода.

Объявление экземпляров делегатов

После определения делегата можно создать экземпляр этого типа. Как и все переменные в C#, экземпляры делегата нельзя объявлять непосредственно в пространстве имен или в глобальном пространстве имен.

```
// inside a class definition:
// Declare an instance of that type:
public Comparison<T> comparator;
```

Тип переменной – `Comparison<T>`, тип делегата определен ранее. Имя переменной – `comparator`.

В приведенном выше фрагменте кода была объявлена переменная-член в классе. Можно также объявить переменные делегатов, локальные переменные или аргументов для методов.

Вызов делегатов

Чтобы вызвать методы, которые находятся в списке вызова делегата, нужно вызвать этот делегат. В методе `Sort()` код вызовет метод сравнения, чтобы определить порядок размещения объектов:

```
int result = comparator(left, right);
```

В строке выше код вызывает метод, подключенный к делегату. Переменная считается именем метода и вызывается с помощью обычного синтаксиса вызова метода.

Эта строка кода содержит небезопасное условие: нет никакой гарантии, что целевой объект было добавлен к делегату. Если целевые объекты не были вложены, строка выше приведет к возникновению исключения `NullReferenceException`.

Назначение, добавление и удаление целевых объектов вызова

Сведения о том, как определяется тип делегата и как объявляются и вызываются экземпляры делегата.

Разработчикам, которые хотят использовать метод `List.Sort()`, нужно определить метод, сигнатура которого совпадает с определением типа делегата, и назначить его делегату, используемому методом `sort`. Это назначение добавляет метод в список вызовов данного делегата объекта.

Предположим, требуется отсортировать список строк по их длине. Функция сравнения может выглядеть следующим образом:

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

Метод объявляется как закрытый метод. Вам может быть не нужно, чтобы этот метод был частью общедоступного интерфейса. Этот метод, присоединенный к делегату, по-прежнему можно использовать в качестве метода сравнения. В вызывающем коде этот метод будет присоединен к целевому списку объекта делегата и будет доступен через этот делегат.

Чтобы создать эту связь, передайте метод в метод `List.Sort()`:

```
phrases.Sort(CompareLength);
```

Обратите внимание, что имя метода используется без скобок. Использование метода как аргумента указывает компилятору преобразовать ссылку на метод в ссылку, которая может применяться как целевой объект вызова делегата, и присоединить этот метод в качестве целевого объекта вызова.

Вы также явно объявили переменную типа `Comparison<string>` и выполнили назначение:

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

Если в качестве объекта делегата используется небольшой метод, для назначения обычно применяется синтаксис лямбда-выражения:

```
Comparison<string> comparer = (left, right) =>
left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

В примере *Sort()* к делегату обычно подключается один целевой метод. Однако объекты делегатов поддерживают списки вызовов, где к объекту делегата присоединено несколько целевых методов.

Классы *Delegate* и *MulticastDelegate*

Описанная выше поддержка языка предоставляет функции и поддержку, которые обычно необходимы для работы с делегатами. Эти возможности основаны на двух классах в платформе *.NET Core*: *Delegate* и *MulticastDelegate*.

Класс *System.Delegate* и его прямой вложенный класс *System.MulticastDelegate* обеспечивают поддержку платформы для создания делегатов, регистрации методов в качестве целевых объектов делегатов и вызова всех методов, которые зарегистрированы как целевые объекты делегатов.

Сами классы *System.Delegate* и *System.MulticastDelegate* не являются типами делегатов. Они являются основой для всех конкретных типов делегатов. Тот же процесс конструкции языка требует, что нельзя объявить класс, который является производным от *Delegate* или *MulticastDelegate*. Это запрещено правилами языка C#.

Вместо этого компилятор C# создает экземпляры класса, производного от *MulticastDelegate*, при использовании ключевого слова языка C# для объявления типов делегатов.

Разберем наиболее распространенные методы, которые будут использоваться при работе с делегатами.

Первый и самый важный момент состоит в том, что каждый делегат является производным от *MulticastDelegate*. Многоадресный делегат означает, что при вызове с помощью делегата может быть вызвано несколько целевых объектов метода.

Методы, которые чаще всего будут использоваться с делегатами, – *Invoke()* и *BeginInvoke()* / *EndInvoke()*. *Invoke()* будет вызывать все методы, которые были прикреплены к определенному экземпляру делегата. Как было показано выше, для вызова делегатов обычно используется синтаксис вызова метода в переменной делегата.

Абстрактный класс делегата предоставляет инфраструктуру для слабой взаимозависимости (*loose coupling*) и вызовов. Конкретные типы делегата становятся гораздо полезнее, поскольку включают и обеспечивают безопасность типов для методов, добавляемых в список вызовов для объекта делегата. Компилятор создает эти методы, если вы используете ключевое слово *delegate* и определяете конкретный тип делегата.

В результате новые типы делегатов создаются всякий раз, когда возникает необходимость в новой сигнатуре метода. Со временем эта работа может стать слишком громоздкой. Для каждого нового компонента требуются новые типы делегатов. Однако, платформа .NET Core содержит несколько типов, которые можно использовать всякий раз, когда вам нужно делегировать типы. Эти определения универсальны, поэтому всякий раз, когда вам нужно объявить новый метод, можно объявить настройку.

Первый из этих типов –это тип `Action` и несколько вариантов:

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

Существуют варианты делегата `Action`, содержащие до 16 аргументов, такие как `Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>`. Очень важно, чтобы для каждого из аргументов делегата в этих определениях использовались универсальные аргументы – это дает максимальную гибкость. Аргументы метода могут быть одного типа, однако это необязательно.

В качестве типа делегата используйте один из типов `Action` с типом возвращаемого значения `void`.

Платформа включает также несколько типов универсальных делегатов, которые можно использовать в качестве типов делегатов, возвращающих значения:

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1,
T2 arg2);
// Other variations removed for brevity
```

Существуют варианты делегата `Func`, содержащие до 16 входных аргументов, такие как `Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>`. Как правило, тип результата должен быть последним параметром типа во всех объявлениях `Func`.

В качестве типа делегата, возвращающего значение, используйте один из типов `Func`.

Существует также специальный тип `Predicate<T>`, предназначенный для делегатов, возвращающих тест одного значения:

```
public delegate bool Predicate<in T>(T obj);
```

Можно заметить, что для каждого типа `Predicate` существует структурно эквивалентный тип `Func`, например:

```
Func<string, bool> TestForString;
Predicate<string> AnotherTestForString;
```

Можно подумать, что два эти типа эквивалентны. Это не так. Эти две переменные не заменяют друг друга. Переменную одного типа нельзя назначить другому типу. В системе типов C# используются имена определенных типов, а не структуры.

Все эти определения типов делегатов в библиотеке .NET Core должны означать, что новый тип делегата для каждой создаваемой функции, которой требуются делегаты, создавать не нужно. Эти универсальные определения должны предоставлять типы делегатов, необходимые в большинстве ситуаций. Можно просто создать экземпляр одного из этих типов с необходимыми параметрами типа. Если алгоритмы можно сделать универсальными, эти делегаты можно использовать как универсальные типы.

Это позволит сэкономить время и свести к минимуму число новых типов, которые нужно создать для работы с делегатами.

События

События, так же, как и делегаты, представляют собой механизм позднего связывания. На самом деле события основаны на тех же средствах языка, которые обеспечивают поддержку делегатов.

С помощью событий объект может сообщить всем компонентам системы, которым это необходимо, о том, что что-то произошло. Любой другой компонент может подписаться на событие, чтобы получать уведомления о его наступлении.

Во многих графических системах есть модель событий, которая позволяет сообщать о действиях пользователей. Такие события сообщают о перемещениях мыши, нажатиях кнопок и иных подобных действиях. Это наиболее распространенный, но, безусловно, не единственный вариант использования событий.

Вы можете определить события, которые должны вызываться для классов. Важным моментом при работе с событиями является то, что для определенного события, может быть, не зарегистрирован ни один объект.

Код необходимо писать так, чтобы он не вызывал событий, если прослушиватели не настроены.

При подписке на событие также создается взаимосвязь между двумя объектами (источником события и приемником событий). Если приемник событий больше не должен получать события, необходимо отменить его подписку на источник события.

Цели при проектировании поддержки событий

Ниже перечислены цели модели событий, реализуемой в языке:

1. Между источником события и приемником событий должна быть минимальная взаимосвязь. Эти два компонента могут создаваться разными организациями и даже обновляться по совершенно разным графикам.

2. Подписка на событие и отмена подписки на него должны производиться максимально просто.

3. Источники событий должны поддерживать несколько подписчиков на события. Кроме того, должен поддерживаться сценарий, когда подписчики на события не подключены.

Как можно увидеть, цели в отношении событий очень похожи на цели в отношении делегатов. Вот почему языковая поддержка событий основана на механизмах поддержки делегатов.

Языковая поддержка событий

Синтаксис определения событий, а также подписки и отмены подписки на них является расширением синтаксиса для делегатов.

Для определения события используется ключевое слово `event`.

```
public event EventHandler<FileListArgs> Progress;
```

Тип события (в этом примере `EventHandler<FileListArgs>`) должен быть типом делегата. При объявлении события должен соблюдаться ряд соглашений. Как правило, тип делегата события имеет возвращаемый тип `void`. Объявление события должно представлять собой глагол или глагольное словосочетание. Если событие сообщает о том, что уже произошло, используйте прошедшее время. Для сообщения о том, что должно произойти, используйте глагол в настоящем времени (например, `Closing`). Настоящее время часто указывает на то, что класс поддерживает какую-либо настройку. Одна из самых распространенных ситуаций – поддержка отмены. Например, событие `Closing` может иметь аргумент, который указывает на то, должна ли продолжаться операция закрытия. В других ситуациях вызывающим объектам может предоставляться возможность изменения поведения путем

изменения свойств аргументов события. Событие может вызываться для указания действия, которое алгоритму предлагается выполнить в следующую очередь. Обработчик событий может потребовать другое действие, изменив свойства аргумента события.

Если нужно инициировать событие, то следует вызвать соответствующий обработчик событий с помощью синтаксиса вызова делегатов:

```
Progress?.Invoke(this, new FileListArgs(file));
```

Как описано в разделе, посвященном делегатам, оператор «*?.*» позволяет легко предотвратить попытки вызова события, если на него нет подписчиков.

Подписка на событие производится с помощью оператора «*+=*»:

```
EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>
    Console.WriteLine(eventArgs.FoundFile);

fileLister.Progress += onProgress;
```

Имя метода обработчика обычно содержит имя события с префиксом On, как показано выше.
Для отмены подписки на событие служит оператор «*-=*»:

```
fileLister.Progress -= onProgress;
```

Важно, чтобы для выражения, представляющего обработчик событий, была объявлена локальная переменная. Благодаря этому при отмене подписки обработчик удаляется. Если вместо этого использовалось тело лямбда-выражения, то производится попытка удалить обработчик, который не был подключен, что не приводит ни к какому результату.

Сигнатуры делегата событий

Стандартной сигнатурой делегата события .NET является:

```
void EventRaised(object sender, EventArgs args);
```

Тип возвращаемого значения – *void*. События основаны на делегатах и являются делегатами многоадресной рассылки. Это обеспечивает поддержку

нескольких подписчиков для любого источника событий. Одно значение, возвращаемое из метода, не масштабируется на несколько подписчиков событий.

Список аргументов содержит два аргумента: отправителя и аргументы события. Тип времени компиляции *sender—System.Object*.

Второй аргумент обычно являлся типом, производным от *System.EventArgs*. Даже если тип события не требует дополнительных аргументов, необходимо предоставить оба аргумента. Существует специальное значение *EventArgs.Empty*, которое следует использовать для обозначения того, что событие не содержит никаких дополнительных сведений.

Создадим класс, который перечисляет соответствующие шаблону файлы в каталоге или любом из его подкаталогов. Этот компонент создает событие для каждого найденного файла, который соответствует шаблону.

Использование модели событий обеспечивает некоторые преимущества разработки. Можно создать несколько прослушивателей событий, которые выполняют разные действия при нахождении искомого файла. Сочетание разных прослушивателей позволяет создавать более надежные алгоритмы.

Ниже показано объявление аргумента исходного события для поиска искомого файла:

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

        public FileFoundArgs(string fileName) => FoundFile = 
fileName;
}
```

Несмотря на то, что этот тип выглядит как небольшой тип, содержащий только данные, вы должны выполнить соглашение и назначить его ссылочным типом (*class*). Это означает, что объект аргумента будет передаваться по ссылке, а любые обновления данных будут доступны всем подписчикам. Первая версия является неизменяемым объектом. Рекомендуется сделать свойства в типе аргумента события неизменяемыми. Таким образом, один подписчик не сможет изменить значения до того, как их увидит другой подписчик. (Существуют исключения, как можно будет увидеть ниже.)

Затем нужно создать объявление события в классе *FileSearcher*. Использование типа *EventHandler<T>* означает, что вам не требуется

создавать еще одно определение типа. Вы просто используете универсальную специализацию.

Заполним класс `FileSearcher` для поиска файлов, соответствующих шаблону, и вызова правильного события при обнаружении совпадения.

```
public class FileSearcher
{
    public event EventHandler<FileFoundArgs>? FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory,
searchPattern))
        {
            RaiseFileFound(file);
        }
    }

    private void RaiseFileFound(string file) =>
        FileFound?.Invoke(this, new FileFoundArgs(file));
}
```

Определение и вызов событий, подобных полям

Самый простой способ добавить событие в класс –объявить это событие как открытое поле, как показано в предыдущем примере.

```
public event EventHandler<FileFoundArgs>? FileFound;
```

В таком коде объявляется открытое поле, что не рекомендуется в объектно-ориентированном программировании, поскольку необходимо обеспечить защиту доступа к данным с помощью свойств и методов. Хотя это выглядит нарушением рекомендаций, код, созданный компилятором, создает программы-оболочки, чтобы доступ к объектам событий мог осуществляться только безопасным образом. Единственные операции, доступные для событий, подобных полям, –обработчик `add`:

```
var fileLister = new FileSearcher();
int filesFound = 0;

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
```

```
};

fileLister.FileFound += onFileFound;
```

и обработчик remove:

```
fileLister.FileFound -= onFileFound;
```

Обратите внимание, что для обработчика используется локальная переменная. Если вы используете тело лямбда-выражения, удаление не будет работать корректно. Будет существовать другой экземпляр делегата, не выполняющий никаких действий.

Код вне класса не может вызывать события, а также выполнять другие операции.

Возвращаемые значения от подписчиков событий

При вызове события `found` прослушиватели должны иметь возможность остановить дальнейшую обработку, если этот файл является последним искомым.

Обработчики событий не возвращают значение, поэтому вам нужно выполнить это другим способом. Стандартный шаблон событий использует `EventArgs` объект для включения полей, которые подписчики событий могут использовать для передачи данных об отмене.

Можно использовать два разных шаблона на основе семантики контракта отмены. В обоих случаях в `EventArguments` добавляется логическое поле для события найденного файла.

Один шаблон позволяет любому одному подписчику отменить операцию. Для этого шаблона новое поле инициализируется значением `false`. Любой подписчик может изменить его на `true`. После того как все подписчики увидят событие, компонент `FileSearcher` проверяет логическое значение и выполняет действие.

Второй шаблон отменяет операцию только в том случае, если все подписчики хотят отменить операцию. В этом шаблоне новое поле инициализируется для указания того, что операцию следует отменить, и любой подписчик может изменить его, чтобы указать, что следует продолжить операцию. После того как все подписчики увидят событие, компонент `FileSearcher` проверяет логическое значение и выполняет действие. В этом шаблоне есть еще один дополнительный шаг: компонент должен знать, все ли подписчики видели событие. Если подписчики отсутствуют, поле неверно сообщит об отмене.

Реализуем первую версию для этого примера. Добавим логическое поле с именем *CancelRequested* в тип *FileFoundArgs*:

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

        public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

Это новое поле автоматически инициализируется *false* значением по умолчанию для *Boolean* поля, поэтому случайное отмена не выполняется. Единственным другим изменением в компоненте является установка флага после вызова события для просмотра, если любой из подписчиков запросил отмену:

```
private void SearchDirectory(string directory, string searchPattern)
{
      foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        FileFoundArgs args = RaiseFileFound(file);
        if (args.CancelRequested)
        {
            break;
        }
    }
}

private FileFoundArgs RaiseFileFound(string file)
{
    var args = new FileFoundArgs(file);
    FileFound?.Invoke(this, args);
    return args;
}
```

Одно из преимуществ этого шаблона заключается в том, что это не является критическим изменением. Никто из подписчиков не запросил отмену ранее, и они по-прежнему не являются. Код подписчиков не требует

обновления, если не требуется поддержка нового протокола отмены. Они очень слабо связаны.

Изменим подписчик, чтобы он запрашивал отмену, когда обнаруживает первый исполняемый файл:

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};
```

Добавление еще одного объявления события

Добавим еще одну возможность и продемонстрируем другие выражения языка для событий. Добавим перегрузку метода `Search`, который проходит через все подкаталоги в поиске файлов.

Эта операция может выполняться длительное время в каталоге с большим числом вложенных каталогов. Добавим событие, которое вызывается в начале каждого нового поиска в каталоге. Это позволяет подписчикам отслеживать ход выполнения и сообщать о нем пользователю. Все примеры, которые мы создали до сих пор, являются открытыми. Сделаем это событие внутренним. Это означает, что типы, используемые для аргументов, также можно сделать внутренними.

Начнем с создания нового производного класса `EventArgs` для передачи сведений о новом каталоге и ходе выполнения.

```
internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

     internal SearchDirectoryArgs(string dir, int totalDirs, int
completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

Теперь определим событие. На этот раз будет использоваться другой синтаксис. Помимо синтаксиса полей можно явно создать свойство с

помощью обработчиков *add* и *remove*. В этом примере не будем добавлять код в эти обработчики, здесь просто демонстрируется их создание.

```
internal event EventHandler<SearchDirectoryArgs>
DirectoryChanged
{
    add { _directoryChanged += value; }
    remove { _directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs>? _directoryChanged;
```

Созданный здесь код очень похож на тот код, который компилятор создает для определения полей событий, как было показано ранее. Для создания события используется синтаксис, очень похожий на используемый для свойств. Обратите внимание, что обработчики имеют разные имена: *add* и *remove*. Они вызываются для подписки на событие или отмены подписки на событие. Учтите, что необходимо объявить закрытое резервное поле для хранения переменной событий. Оно инициализируется значением *NULL*.

Теперь добавим перегрузку метода *Search*, который обходит подкаталоги и вызывает оба события. Для этого проще всего использовать аргумент по умолчанию для задания поиска по всем каталогам:

```
public void Search(string directory, string searchPattern, bool
searchSubDirs = false)
{
    if (searchSubDirs)
    {
         var allDirectories = Directory.GetDirectories(directory,
"*.*", SearchOption.AllDirectories);
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
        {
                    RaiseSearchDirectoryChanged(dir, totalDirs,
completedDirs++);
              // Search 'dir' and its subdirectories for files
that match the search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
              RaiseSearchDirectoryChanged(directory, totalDirs,
completedDirs++);

        SearchDirectory(directory, searchPattern);
```

```
    }
    else
    {
        SearchDirectory(directory, searchPattern);
    }
}

private void SearchDirectory(string directory, string
searchPattern)
{
      foreach (var file in Directory.EnumerateFiles(directory,
searchPattern))
    {
        FileFoundArgs args = RaiseFileFound(file);
        if (args.CancelRequested)
        {
            break;
        }
    }
}

private void RaiseSearchDirectoryChanged(
    string directory, int totalDirs, int completedDirs) =>
    _directoryChanged?.Invoke(
        this,
                    new SearchDirectoryArgs(directory, totalDirs,
completedDirs));

private FileFoundArgs RaiseFileFound(string file)
{
    var args = new FileFoundArgs(file);
    FileFound?.Invoke(this, args);
    return args;
}
```

На этом этапе можно запустить приложение, вызывающее перегруженный метод для поиска всех вложенных каталогов. Для нового события `DirectoryChanged` нет подписчиков, однако благодаря использованию идиомы `?.Invoke()` мы можем гарантировать правильную работу метода.

Добавим обработчик для написания строки, показывающей ход выполнения в окне консоли.

```
fileLister.DirectoryChanged += (sender, eventArgs) =>
{
```

```
                                        Console.Write($"Entering
'{eventArgs.CurrentSearchDirectory}'.");
            Console.WriteLine($"   {eventArgs.CompletedDirs}   of
{eventArgs.TotalDirs} completed...");
};
```

В этой версии определение `EventHandler<TEventArgs>` больше не имеет ограничения, указывающего на то, что `TEventArgs` должен быть классом, производным от `System.EventArgs`.

В результате повышается гибкость разработки и обеспечивается обратная совместимость. Начнем с гибких возможностей. Класс `System.EventArgs` представляет один метод: `MemberwiseClone()`, который создает неполную копию объекта. Чтобы реализовать свою функциональность для любого класса, производного от `EventArgs`, этот метод должен использовать отражение. Функциональность проще создать в определенном производном классе. Это фактически означает, что наследование от `System.EventArgs` является ограничением, которое регламентирует разработку, но не предоставляет никаких дополнительных преимуществ. На самом деле, можно изменить определения `FileFoundArgs` и `SearchDirectoryArgs` так, чтобы они не были производными от `EventArgs`. Программа будет работать точно так же.

Вместо `SearchDirectoryArgs` можно также использовать структуру, внеся дополнительное изменение.

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

     internal SearchDirectoryArgs(string dir, int totalDirs, int
completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

Дополнительное изменение заключается в вызове конструктора без параметров перед входом в конструктор, который инициализирует все поля.

Без этого согласно правилам языка C# будет выведено сообщение о получении доступа к свойствам до их назначения.

Не следует изменять *FileFoundArgs* с класса (ссылочный тип) на структуру (тип значения). Это связано с тем, что протокол для обработки отмены требует передачи аргументов события по ссылке. Если тип изменить на структуру, класс поиска файла никогда не сможет отслеживать изменения, внесенные подписчиками событий. Для каждого подписчика будет использоваться новая копия, которая будет отличаться от той, которую обнаружил объект поиска файла.

Теперь рассмотрим обратную совместимость этого изменения. Удаление ограничения не влияет на существующий код. Все имеющиеся типы аргументов событий по-прежнему являются производными от *System.EventArgs*. Обратная совместимость является одной из основных причин, по которой они будут и далее являться производными от *System.EventArgs*. Все существующие подписчики на события будут подписчиками на событие, следующее классическому шаблону.

Согласно аналогичной логике, теперь тип события аргумента не будет иметь подписчиков в существующих базах кода. Новые типы событий, которые не являются производными от *System.EventArgs*, не нарушат функциональность этих баз кода.

Лямбда-выражения

Лямбда-выражение используется для создания анонимной функции. Используйте оператор объявления лямбда-выражения => для отделения списка параметров лямбда-выражения от исполняемого кода. Лямбда-выражение может иметь одну из двух следующих форм:

Лямбда выражения, имеющая выражение в качестве текста:

```
(input-parameters) => expression
```

Лямбда оператора, имеющая блок операторов в качестве текста:

```
(input-parameters) => { <sequence-of-statements> }
```

Чтобы создать лямбда-выражение, необходимо указать входные параметры (если они есть) с левой стороны лямбда-оператора и блок выражений или операторов с другой стороны.

Лямбда-выражение может быть преобразовано в тип делегата. Тип делегата, в который может быть преобразовано лямбда-выражение,

определяется типами его параметров и возвращаемым значением. Если лямбда-выражение не возвращает значение, оно может быть преобразовано в один из типов делегата *Action*; в противном случае его можно преобразовать в один из типов делегатов *Func*. Например, лямбда-выражение, которое имеет два параметра и не возвращает значение, можно преобразовать в делегат *Action<T1,T2>*. Лямбда-выражение, которое имеет два параметра и возвращает значение, можно преобразовать в делегат *Func<T,TResult>*. В следующем примере лямбда-выражение *x => x * x*, которое указывает параметр с именем x и возвращает значение x в квадрате, присваивается переменной типа делегата:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

Лямбда-выражения можно также преобразовать в типы дерева выражения, как показано в следующем примере:

```
System.Linq.Expressions.Expression<Func<int, int>> e = x =>
x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

Лямбда-выражения можно использовать в любом коде, для которого требуются экземпляры типов делегатов или деревьев выражений, например в качестве аргумента метода *Task.Run(Action)* для передачи кода, который должен выполняться в фоновом режиме. Можно также использовать лямбда-выражения при применении LINQ в C#, как показано в следующем примере:

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

Лямбда-выражение с выражением с правой стороны оператора => называется выражением лямбда. Выражения-лямбды возвращают результат выражения и принимают следующую основную форму.

```
input-parameters) => expression
```

Текст выражения лямбды может состоять из вызова метода. Но при создании деревьев выражений, которые вычисляются вне контекста поддержки общеязыковой среды выполнения (CRL) .NET, например, в SQL Server, вызовы методов не следует использовать в лямбда-выражениях. Методы не имеют смысла вне контекста среды CLR .NET.

Лямбда-инструкция напоминает лямбда-выражение, за исключением того, что инструкции заключаются в фигурные скобки:

```
(input-parameters) => { <sequence-of-statements> }
```

Тело лямбды оператора может состоять из любого количества операторов; однако на практике обычно используется не более двух-трех.

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

Лямбда-инструкции нельзя использовать для создания деревьев выражений.

Входные параметры лямбда-выражения

Входные параметры лямбда-выражения заключаются в круглые скобки. Нулевое количество входных параметров задается пустыми скобками:

```
Action line = () => Console.WriteLine();
```

Если лямбда-выражение имеет только один входной параметр, круглые скобки необязательны:

```
Func<double, double> cube = x => x * x * x;
```

Два и более входных параметра разделяются запятыми:

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

Иногда компилятор не может вывести типы входных параметров. Вы можете указать типы данных в явном виде, как показано в следующем примере:

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Для входных параметров все типы нужно задать либо в явном, либо в неявном виде. В противном случае компилятор выдает ошибку CS0748.

Вы можете использовать *dis карта s* для указания двух или нескольких входных параметров лямбда-выражения, которые не используются в выражении:

```
Func<int, int, int> constant = (_, _) => 42;
```

Параметры пустой переменной лямбда-выражения полезны, если лямбда-выражение используется для указания обработчика событий.

Примечание

Если только один входной параметр имеет имя _, для обеспечения обратной совместимости _ рассматривается как имя этого параметра в лямбда-выражении.

Начиная с C# 12, можно указать значения по умолчанию для параметров в лямбда-выражениях. Синтаксис и ограничения значений параметров по умолчанию совпадают с методами и локальными функциями. В следующем примере объявляется лямбда-выражение с параметром по умолчанию, а затем вызывает его один раз с использованием значения по умолчанию и один раз с двумя явными параметрами:

```
var IncrementBy = (int source, int increment = 1) => source + increment;
Console.WriteLine(IncrementBy(5)); // 6
Console.WriteLine(IncrementBy(5, 2)); // 7
```

Можно также объявить лямбда-выражения с массивами в `params` качестве параметров:

```
var sum = (params int[] values) =>
{
    int sum = 0;
    foreach (var value in values)
```

```
            sum += value;

        return sum;
    };

    var empty = sum();
    Console.WriteLine(empty); // 0

    var sequence = new[] { 1, 2, 3, 4, 5 };
    var total = sum(sequence);
    Console.WriteLine(total); // 15
```

В рамках этих обновлений, когда группе методов, которая имеет параметр по умолчанию, назначается лямбда-выражение, это лямбда-выражение также имеет тот же параметр по умолчанию. Группу методов с параметром массива `params` также можно назначить лямбда-выражению.

Лямбда-выражения с параметрами или `params` массивами по умолчанию в качестве параметров не имеют естественных типов, соответствующих `Func<>` или `Action<>` типам. Однако можно определить типы делегатов, которые включают значения параметров по умолчанию:

```
    delegate int IncrementByDelegate(int source, int increment
= 1);
    delegate int SumDelegate(params int[] values);
```

Кроме того, можно использовать неявно типизированные переменные с `var` объявлениями для определения типа делегата. Компилятор синтезирует правильный тип делегата.

Лямбда-выражения со стандартными операторами запросов

LINQ to Objects, среди других реализаций, имеет входной параметр, тип которого является одним из `Func<TResult>` семейств универсальных делегатов. Эти делегаты используют параметры типа для определения количества и типов входных параметров, а также тип возвращаемого значения делегата. Делегаты Func полезны для инкапсуляции пользовательских выражений, которые применяются к каждому элементу в наборе исходных данных. В качестве примера рассмотрим следующий тип делегата `Func<T,TResult>`:

```
    public delegate TResult Func<in T, out TResult>(T arg)
```

Экземпляр этого делегата можно создать как *Func<int, bool>*, где *int* –входной параметр, а *bool* – возвращаемое значение. Возвращаемое значение всегда указывается в последнем параметре типа. Например, *Func<int, string, bool>* определяет делегат с двумя входными параметрами, *int* и *string*, и типом возвращаемого значения *bool*. Следующий делегат *Func* при вызове возвращает логическое значение, которое показывает, равен ли входной параметр 5:

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result);   // False
```

Лямбда-выражения также можно использовать, когда аргумент имеет тип *Expression<TDelegate>*, например, в стандартных операторах запросов, которые определены в типе *Queryable*. При указании аргумента *Expression<TDelegate>* лямбда-выражение компилируется в дерево выражения.

В этом примере используется стандартный оператор запроса *Count*:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in
{string.Join(" ", numbers)}");
```

Компилятор может вывести тип входного параметра ввода; но его также можно определить явным образом. Данное лямбда-выражение подсчитывает указанные целые значения (n), которые при делении на два дают остаток 1.

В следующем примере кода показано, как создать последовательность, которая содержит все элементы массива *numbers*, предшествующие 9, так как это первое число последовательности, не удовлетворяющее условию:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n <
6);
Console.WriteLine(string.Join("                                ",
firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

В следующем примере показано, как указать несколько входных параметров путем их заключения в скобки. Этот метод возвращает все элементы в массиве *numbers* до того числа, значение которого меньше его порядкового номера в массиве:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n
>= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

Лямбда-выражения не используются непосредственно в выражениях запросов, но их можно использовать в вызовах методов в выражениях запросов, как показано в следующем примере:

```
var numberSets = new List<int[]>
{
    new[] { 1, 2, 3, 4, 5 },
    new[] { 0, 0, 0 },
    new[] { 9, 8 },
    new[] { 1, 0, 1, 0, 1, 0, 1, 0 }
};

var setsWithManyPositives =
    from numberSet in numberSets
    where numberSet.Count(n => n > 0) > 3
    select numberSet;

foreach (var numberSet in setsWithManyPositives)
{
    Console.WriteLine(string.Join(" ", numberSet));
}
// Output:
// 1 2 3 4 5
// 1 0 1 0 1 0 1 0
```

Естественный тип лямбда-выражения

Лямбда-выражение само по себе не имеет типа, так как система общих типов не имеет встроенной концепции "лямбда-выражения". Однако иногда удобно говорить о "типе" лямбда-выражения. Под неофициальным термином "тип" понимается тип делегата или тип *Expression*, в который преобразуется лямбда-выражение.

Начиная с C# 10, лямбда-выражение может иметь естественный тип. Вам не потребуется объявлять тип делегата, например *Func<...>* или *Action<...>* для лямбда-выражения, потому что компилятор может вывести тип делегата из лямбда-выражения. В качестве примера рассмотрим следующее объявление:

```
var parse = (string s) => int.Parse(s);
```

Компилятор может определить *parse* как *Func<string, int>*. Компилятор использует доступный делегат *Func* или *Action*, если он существует. Если нет, компилятор синтезирует тип делегата. Например, тип делегата синтезируется, если лямбда-выражение имеет параметры *ref*. Если лямбда-выражение имеет естественный тип, его можно присвоить менее явному типу, например, *System.Object* или *System.Delegate*:

```
object parse = (string s) => int.Parse(s);       // Func<string, int>
Delegate parse = (string s) => int.Parse(s); // Func<string, int>
```

Группы методов (то есть имена методов без списков параметров) с ровно одной перегрузкой имеют естественный тип:

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

Если присвоить лямбда-выражение *System.Linq.Expressions.LambdaExpression* или *System.Linq.Expressions.Expression*, и лямбда имеет естественный тип делегата, выражение имеет естественный тип *System.Linq.Expressions.Expression<TDelegate>* с естественным типом делегата, используемым в качестве аргумента для параметра типа:

```
LambdaExpression parseExpr = (string s) => int.Parse(s); // Expression<Func<string, int>>
Expression parseExpr = (string s) => int.Parse(s);       // Expression<Func<string, int>>
```

Не у всех лямбда-выражений есть естественный тип. Рассмотрим следующее объявление:

```
var parse = s => int.Parse(s); // ERROR: Not enough type info in the lambda
```

Компилятор не может определить тип параметра для s. Если компилятор не может определить естественный тип, необходимо объявить тип:

```
Func<string, int> parse = s => int.Parse(s);
```

Явный тип возвращаемого значения

Как правило, тип возвращаемого значения лямбда-выражения является очевидным и легко выводится. Для некоторых выражений, которые не работают:

```
var choose = (bool b) => b ? 1 : "two"; // ERROR: Can't infer return type
```

Начиная с C# 10, можно указать тип возвращаемого значения лямбда-выражения перед входными параметрами. Если вы указываете явный тип возвращаемого значения, заключите входные параметры в скобки:

```
var choose = object (bool b) => b ? 1 : "two"; // Func<bool, object>
```

Атрибуты

Начиная с C# 10, вы можете добавлять атрибуты в лямбда-выражение и его параметры. В следующем примере показано, как добавить атрибуты в лямбда-выражение:

```
Func<string?, int?> parse = [ProvidesNullCheck] (s) => (s is not null) ? int.Parse(s) : null;
```

Кроме того, вы можете добавить атрибуты во входные параметры или возвращаемое значение, как показано в следующем примере:

```
var concat = ([DisallowNull] string a, [DisallowNull] string b) => a + b;
var inc = [return: NotNullIfNotNull(nameof(s))] (int? s) => s.HasValue ? s++ : null;
```

Как показано в предыдущих примерах, при добавлении атрибутов в лямбда-выражение или его параметры вам нужно заключить входные параметры в скобки.

Важно!

Лямбда-выражения вызываются через базовый тип делегата. Это отличается от методов и локальных функций. Метод делегата `Invoke` *не проверяет атрибуты в лямбда-выражении. При вызове лямбда-выражения атрибуты не оказывают никакого влияния. Атрибуты лямбда-выражений полезны для анализа кода и могут быть обнаружены с помощью отражения. Одно из последствий этого решения – невозможность применить* `System.Diagnostics.ConditionalAttribute` *к лямбда-выражению.*

Запись внешних переменных и области видимости переменной в лямбда-выражениях

Лямбда-выражения могут ссылаться на внешние переменные. Эти внешние переменные являются переменными, которые находятся в области в методе, определяющем лямбда-выражение, или в области в типе, который содержит лямбда-выражение. Переменные, полученные таким способом, сохраняются для использования в лямбда-выражениях, даже если бы в ином случае они оказались за границами области действия и уничтожились сборщиком мусора. Внешняя переменная должна быть определенным образом присвоена, прежде чем она сможет использоваться в лямбда-выражениях. В следующем примере демонстрируются эти правила.

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int>? updateCapturedLocalVariable;
                                internal    Func<int,    bool>?
isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                    Console.WriteLine($"{j} is greater than
{input}: {result}");
```

```
                };

                isEqualToCapturedLocalVariable = x => x == j;

                      Console.WriteLine($"Local variable before
lambda invocation: {j}");
                updateCapturedLocalVariable(10);
                 Console.WriteLine($"Local variable after lambda
invocation: {j}");
            }
        }

        public static void Main()
        {
            var game = new VariableCaptureGame();

            int gameInput = 5;
            game.Run(gameInput);

            int jTry = 10;
             bool result = game.isEqualToCapturedLocalVariable!
(jTry);
                Console.WriteLine($"Captured local variable is
equal to {jTry}: {result}");

            int anotherJ = 3;
            game.updateCapturedLocalVariable!(anotherJ);

                                      bool    equalToAnother    =
game.isEqualToCapturedLocalVariable(anotherJ);
              Console.WriteLine($"Another lambda observes a new
value of captured variable: {equalToAnother}");
        }
        // Output:
        // Local variable before lambda invocation: 0
        // 10 is greater than 5: True
        // Local variable after lambda invocation: 10
        // Captured local variable is equal to 10: True
        // 3 is greater than 5: False
          // Another lambda observes a new value of captured
variable: True
    }
```

Следующие правила применимы к области действия переменной в лямбда-выражениях.

1. Захваченная переменная не будет уничтожена сборщиком мусора до тех пор, пока делегат, который на нее ссылается, не перейдет в статус подлежащего уничтожению при сборке мусора.

2. Переменные, представленные в лямбда-выражении, невидимы в заключающем методе.

3. Лямбда-выражение не может непосредственно захватывать параметры *in*, *ref* или *out* из заключающего метода.

4. Оператор *return* в лямбда-выражении не вызывает возврат значения заключающим методом.

5. Лямбда-выражение не может содержать операторы *goto*, *break* или *continue*, если целевой объект этого оператора перехода находится за пределами блока лямбда-выражения. Если целевой объект находится внутри блока, использование оператора перехода за пределами лямбда-выражения также будет ошибкой.

6. К лямбда-выражению можно применить модификатор *static*, чтобы предотвратить непреднамеренный захват локальных переменных или состояния экземпляра лямбда-выражения:

```
Func<double, double> square = static x => x * x;
```

Статическое лямбда-выражение не может сохранять локальные переменные или состояние экземпляров из охватывающих областей, но может ссылаться на статические элементы и определения констант.

Индивидуальные задания для лабораторной работы

1. Для созданного в предыдущей лабораторной работе контейнерного класса реализовать методы, которые в качестве аргумента принимают делегат:
   - метод сортировки;
   - метод поиска;
   - метод фильтрации.
2. Для каждого делегата определить один метод и одно лямбда-выражение.
3. Коллекция изменяется при удалении/добавлении элементов или при изменении одной из входящих в коллекцию ссылок, например, когда одной из ссылок присваивается новое значение. В этом случае в соответствующих методах или свойствах класса бросаются события.

4. При изменении данных объектов, ссылки на которые входят в коллекцию, значения самих ссылок не изменяются. Этот тип изменений не порождает событий. Для событий, извещающих об изменениях в коллекции, определяется свой делегат. События регистрируются в специальных классах-слушателях

5. Для событий предусмотреть возможность подписки и отписки от события.

6. Для обработки всех ошибочных ситуаций использовать конструкцию try…catch().

7. В Main создать два экземпляра шаблонного класса-контейнера для разных типов данных. Работа с этими объектами должна демонстрироваться на следующих операциях: добавить – просмотреть – найти – удалить – найти – просмотреть.

8. Отладить и выполнить полученную программу. Проверить обработку исключительных ситуаций (например, чтение из пустого стека, дублирование объектов и т.п.).

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретическую часть лабораторной работы.

2. Реализовать индивидуальное задание согласно варианту, сделать скриншоты работающих программ. Написать комментарии.

3. Написать отчет, содержащий:

1. Титульный лист, на котором указывается:

а) полное наименование министерства образование и название учебного заведения;

б) название дисциплины;

в) номер практического занятия;

г) фамилия преподавателя, ведущего занятие;

д) фамилия, имя и номер группы студента;

е) год выполнения лабораторной работы.

2. Индивидуальное задание (листинг кода программы, скриншоты консоли с результатами работы).

3. Вывод о проделанной работе.