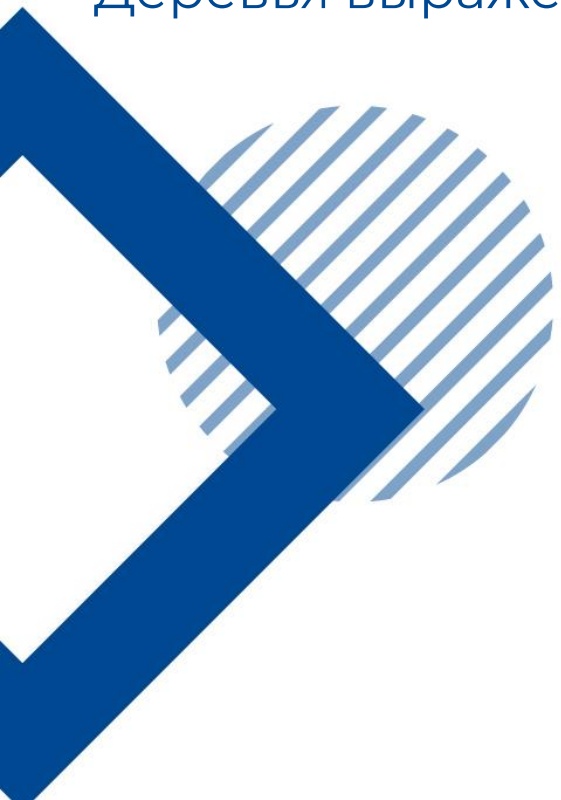
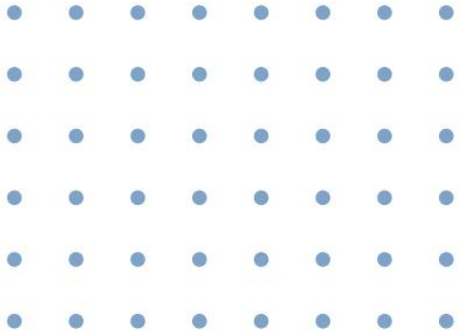
Лекция 5

Делегаты
Встроенные делегаты
Событийное программирование
Анонимные методы
Лямбда-выражения
Деревья выражений





2022

Делегаты

Делегаты

Делегаты

Делегат – это тип, который представляет собой ссылки на методы с определенным списком параметров и возвращаемым типом.
При создании экземпляра делегата этот экземпляр можно связать с любым методом с совместимой сигнатурой и возвращаемым типом.
Метод можно вызвать (активировать) с помощью экземпляра делегата. Для этого у экземпляра делегата надо вызвать метод Invoke.

<мод. доступа> delegate <тип возвр. знач.> <имя делегата>(<параметры>);

Делегаты

Делегаты

```
Del1 del1 = new Del1(Meth1);
Del1 del2 = Meth1;

del1();
del2.Invoke();

Del2 del21 = Meth2;

void Meth1()
{
    Console.WriteLine("Call Meth1");
}

int Meth2(int a, double c)
{
    Console.WriteLine($"int: {a}, double: {c}");
    return a * Convert.ToInt32(c);
}

del21.Invoke(34, 45.6);
del21(23, 78.8);

Console.ReadKey();

delegate void Del1();
```

```
namespace Lection05
{
    internal delegate int Del2(int i, double d);
}
```

```
U:\Teaching\RPP\FirstSemester\Fir
Hello, World!
Call Meth1
Call Meth1
int: 34, double: 45,6
int: 23, double: 78,8
```

Делегаты

Асинхронный обратный вызов

Поскольку созданный экземпляр делегата является объектом, его можно передавать как параметр или назначать свойству.
Это позволяет методу принимать делегат в качестве параметра и вызывать делегат в дальнейшем.
Эта процедура называется **асинхронным обратным вызовом** и обычно используется для уведомления вызывающего объекта о завершении длительной операции.

Делегаты

Асинхронный обратный вызов

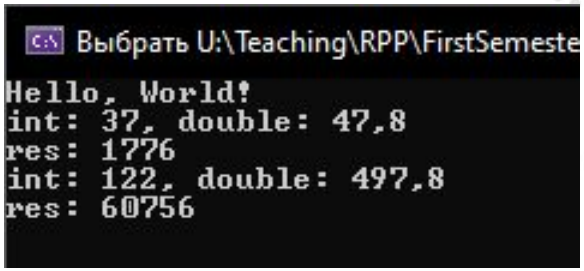
```
CallDel(37,47.8, del21);
CallDel(122, 497.8, Meth2);

Console.ReadKey();

void Meth1()...

int Meth2(int a, double c)
{
    Console.WriteLine($"int: {a}, double: {c}");
    return a * Convert.ToInt32(c);
}

void CallDel(int f, double d, Del2 del)
{
    int res = del(f, d);
    Console.WriteLine($"res: {res}");
}
```



Делегаты

Многоадрестность

При привязке метода к объекту-делегату в объекте сохраняется **адрес** привязываемого метода. К объекту-делегату можно привязать сразу **несколько** методов. Это называется **многоадресностью**.
Чтобы добавить в список методов делегата (список вызова) дополнительный метод, необходимо просто добавить его к делегату с помощью оператора сложения или назначения сложения ("+" или "+=").

Делегаты

Многоадрестность

Все методы в делегате попадают в специальный список – список вызова или invocation list. И при вызове делегата все методы из этого списка последовательно вызываются.
При добавлении методов следует учитывать, что можно добавить ссылку на один и тот же метод несколько раз, и в списке вызова делегата тогда будет несколько ссылок на один и то же метод. Соответственно при вызове делегата добавленный метод будет вызываться столько раз, сколько он был добавлен.
Если делегат возвращает некоторое значение, то возвращается значение последнего метода из списка вызова (если в списке вызова несколько методов)

Делегаты

Многоадрестность

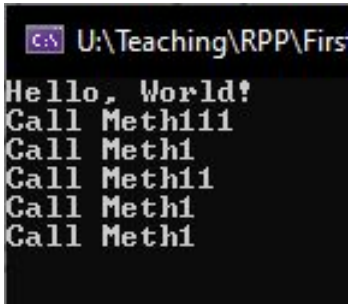
```
Del1 del1 = Meth111;

del1 += Meth1;
del1 += Meth11;
del1 += Meth1;
del1 += Meth1;

del1();

Console.ReadKey();
```

```
void Meth1()
{
    Console.WriteLine("Call Meth1");
}
void Meth11()
{
    Console.WriteLine("Call Meth11");
}
void Meth111()
{
    Console.WriteLine("Call Meth111");
}
```



Делегаты

Удаление метода

Чтобы удалить метод из списка вызова, следует использовать оператор decrement или назначения decrement ("-" или «-=»).
Стоит отметить, что при удалении метода может сложиться ситуация, что в делегате **не будет** методов, и тогда переменная будет иметь значение **null**.
При удалении следует учитывать, что если делегат содержит несколько ссылок на один и тот же метод, то операция -= начинает поиск с **конца** списка вызова делегата и удаляет только первое найденное вхождение. Если подобного метода в списке вызова делегата нет, то операция -= не имеет никакого эффекта.

Делегаты

Удаление метода

```
void Meth1()
{
    Console.WriteLine("Call Meth1");
}
void Meth11()
{
    Console.WriteLine("Call Meth11");
}
void Meth111()
{
    Console.WriteLine("Call Meth111");
}
```

```
Del1 del1 = Meth111;

del1 += Meth1;
del1 += Meth11;
del1 += Meth1;
del1 += Meth1;

del1();

Console.ReadKey();
```

```
del1 -= Meth1;
del1 -= Meth1;
del1 -= Meth1;
del1 -= Meth1;
Console.WriteLine();
del1();


del1 -= Meth111;
del1 -= Meth11;
Console.WriteLine();
del1?.Invoke();
```

Делегаты

Обобщенные (параметризованные) делегаты

Делегаты, как и другие типы, могут быть обобщенными.
Универсальный делегат задается также, как и универсальный метод, после имени делегата в угловых скобках прописываются параметры, которые могут использоваться в передаваемых параметрах или возвращаемом значении.

Делегаты

Обобщенные (параметризованные) делегаты

```
namespace Lection05
{
    internal delegate int Del2(int i, double d);

    internal delegate T2 Del3<T1, T2>(T1 i);
}
```

```
Del3<int, string> del31 = Meth31;
Del3<string, double> del32 = Meth32;

Console.ReadKey();

string Meth31(int n)
{
    return n.ToString();
}

double Meth32(string s)
{
    return double.Parse(s);
}
```

Делегаты

Ковариантность и контравариантность

Делегаты могут быть ковариантными и контравариантными.
Ковариантность делегата предполагает, что возвращаемым типом может быть производный тип. К делегату можно привязать метод, возвращаемый тип которого является производным (класс-наследник) от типа, указанного в качестве возвращаемого в делегате.
Контрвариантность делегата предполагает, что типом параметра может быть более универсальный тип. К делегату можно привязать метод, тип параметра которого является более универсальным (класс-родитель) по отношению к типу параметра делегата.

Делегаты

Ковариантность и контравариантность

```
namespace Lection05
{
    internal class ClassParent
    {
    }
}
```

```
namespace Lection05
{
    internal class ClassChild : ClassParent
    {
    }
}
```

```
namespace Lection05
{
    internal delegate int Del2(int i, double d);

    internal delegate T2 Del3<T1, T2>(T1 i);

    internal delegate ClassParent Del4(ClassChild elem);
}
```

```
Del4 del4 = Meth40;
del4 += Meth41;
del4 += Meth42;
del4 += Meth43;

Console.ReadKey();

ClassParent Meth40(ClassChild elem)
{
    return (ClassParent)elem;
}
ClassChild Meth41(ClassChild elem)
{
    return elem;
}
ClassParent Meth42(ClassParent elem)
{
    return elem;
}
ClassChild Meth43(ClassParent elem)
{
    return new ClassChild();
}
```

Встроенные делегаты

Встроенные делегаты

Встроенные делегаты

В C# существуют встроенные делегаты. Самые часто используемые из них:

- Action – принимает до 16 параметров и не возвращает значение.

```
public delegate void Action();
public delegate void Action<in T>(T obj);
public delegate void Action<in T1, in T2,....in T16>;
```

- Func – аналог Action, только еще возвращает значение.

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, in T2,...in T16, out TResult>();
```

- Predicate – используется для сравнения объекта T по определенному условию. Возвращается bool значение.

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

Встроенные делегаты

Встроенные делегаты

```
Action act = Meth1;
act += Meth1;
act += Meth11;

Func<ClassChild, ClassParent> func = Meth40;
func += Meth41;
func += Meth42;
func += Meth43;
```

```
void Meth1()
{
    Console.WriteLine("Call Meth1");
}
void Meth11()
{
    Console.WriteLine("Call Meth11");
}
void Meth111()
{
    Console.WriteLine("Call Meth111");
}
```

```
ClassParent Meth40(ClassChild elem)
{
    return (ClassParent)elem;
}
ClassChild Meth41(ClassChild elem)
{
    return elem;
}
ClassParent Meth42(ClassParent elem)
{
    return elem;
}
ClassChild Meth43(ClassParent elem)
{
    return new ClassChild();
}
```

Событийное программирование

Событийное программирование

Событийное программирование

Событийно-ориентированное программирование – парадигма программирования, в которой выполнение программы определяется **событиями** – действиями пользователя (клавиатура, мышь, сенсорный экран), сообщениями других программ и потоков, событиями операционной системы (например, поступлением сетевого пакета).
Событийно-ориентированное программирование, как правило, применяется в трёх случаях:

- при построении пользовательских интерфейсов (в том числе графических);
- при создании серверных приложений в случае, если по тем или иным причинам нежелательно порождение обслуживающих процессов;
- при программировании игр, в которых осуществляется управление множеством объектов.

Событийное программирование

Событийное программирование

Разные языки программирования поддерживают СОП в разной степени. Наиболее полной поддержкой событий обладают следующие языки (неполный список):

- Perl (события и демоны DAEMON, и их приоритеты PRIO),
- Delphi,
- ActionScript 3.0,
- C# (события event),
- JavaScript (действия пользователя).

Остальные языки, в большей их части, поддерживают события как обработку исключительных ситуаций.

Событийное программирование

Событие

Событие в объектно-ориентированном программировании – это сообщение, которое возникает в различных точках исполняемого кода при выполнении определённых условий. События предназначены для того, чтобы иметь возможность предусмотреть реакцию программного обеспечения. Для решения поставленной задачи создаются обработчики событий: как только программа попадает в заданное состояние, происходит событие, посылается сообщение, а обработчик перехватывает это сообщение.

В общем случае в обработчик не передаётся ничего, либо передаётся ссылка на объект, инициировавший (породивший) обрабатываемое событие.

В особых случаях в обработчик передаются значения некоторых переменных или ссылки на какие-то другие объекты, чтобы обработка данного события могла учесть контекст возникновения события.

Событийное программирование

Событие

Элемент «Событие» в C# это особый тип многоадресных делегатов, которые можно создавать только в классах или структурах.
События позволяют классу или объекту **уведомлять** другие классы или объекты о возникновении каких-либо ситуаций.
Если на событие подписаны другие классы или структуры, их методы обработчиков событий будут вызваны когда класс издателя инициирует событие.
Класс, отправляющий (или порождающий) событие, называется **издателем**, а классы, принимающие (или обрабатывающие) событие, называются **подписчиками**.

<мод. доступа> event <делегат> <имя поля-события>;

Событийное программирование

Событие

```
namespace Lection05
{
    internal class ClassParent
    {
        protected event Action<string> ev1;

        public void AddCaller(Action<string> del)
        {
            ev1 += del;
        }

        public void DoSomething()
        {
            ev1?.Invoke("Start progress");
            //....
            ev1?.Invoke("Finish progress");
        }
    }
}
```

```
void Message(string s)
{
    Console.WriteLine(s);
}
```

```
namespace Lection05
{
    internal class ClassListner
    {
        public void Method1(string s)
        {
            Console.WriteLine($"Send by email: {s}");
        }

        public void Method2(string s)
        {
            Console.WriteLine($"Write to file: {s}");
        }
    }
}
```

```
ClassListner l = new ClassListner();

ClassParent p = new ClassParent();
p.AddCaller(Message);
Console.Write("Choose message mode:");
int n = Convert.ToInt32(Console.ReadLine());
switch(n)
{
    case 1:
        p.AddCaller(l.Method1);
        break;
    case 2:
        p.AddCaller(l.Method2);
        break;
}
p.DoSomething();
```

```
U:\Teaching\RPP\FirstSemest
Hello, World!
Choose message mode:0
Start progress
Finish progress
```

```
U:\Teaching\RPP\FirstSemester\FirstSemesterLec
Hello, World!
Choose message mode:1
Start progress
Send by email: Start progress
Finish progress
Send by email: Finish progress
```

```
U:\Teaching\RPP\FirstSemester\FirstSemester
Hello, World!
Choose message mode:2
Start progress
Write to file: Start progress
Finish progress
Write to file: Finish progress
```

Событийное программирование

Акссесоры

С помощью специальных акссесоров add/remove можно управлять добавлением и удалением обработчиков.
Аксессор add вызывается при добавлении обработчика, то есть при операции +=. Добавляемый обработчик доступен через ключевое слово value. В этом блоке можно получить информацию об обработчике и определить некоторую логику.
Блок remove вызывается при удалении обработчика. Аналогично в этом блоке можно задать некоторую дополнительную логику.

```
мод. доступа> event <делегат> <имя события>
{
    add { // логика... }
    remove { // логика... }
}
```

Событийное программирование

Акссесоры

```
ClassChild ch = new ClassChild();
ch.Ev2 += Meth33;
ch.Ev2 -= Meth33;
```

```
int Meth33(string s)
{
    return int.Parse(s);
}
```

```
namespace Lection05
{
    internal class ClassChild : ClassParent
    {
        private event Func<string, int> ev2;

        public event Func<string, int> Ev2
        {
            add
            {
                if (value != null)
                {
                    ev2 += value;
                }
            }
            remove
            {
                ev2 -= value;
            }
        }
    }
}
```

Событийное программирование

ООП

СОП

Событийное программирование

ООП

вам нужно иметь записную книжку с номерами **всех** тех, кого вы хотите оповестить о каком-то событии. И **каждому** нужно еще **позвонить** и сказать об этом.

СОП

вы просто «**постите**» новость в социальной сети и все, кто на вас «**подписан**» видят вашу новость.

Событийное программирование

ООП

Необходимо иметь объекты от всех классов, кто должен знать и реагировать на изменения внутри класса и при изменениях внутри класса вызывать определенные методы классов этих объектов.

СОП

Создается поле-«событие» в классе и метод, с помощью которого любой другой класс может «подписаться» на это событие (указать метод, который следует вызывать при совершении события). Далее, если в классе происходят изменения, просто вызывается это событие и все классы, кто на него подписаны реагируют на это.

Событийное программирование

ООП vs. СОП

```
namespace Lection05
{
    internal class ClassOldWay1
    {
        private ClassListnerOld11 _classListnerOld11;
        private ClassListnerOld12 _classListnerOld12;

        public ClassOldWay1(ClassListnerOld11 classListnerOld11, ClassListnerOld12 classListnerOld12)
        {
            _classListnerOld11 = classListnerOld11;
            _classListnerOld12 = classListnerOld12;
        }

        public void AddClassListnerOld11(ClassListnerOld11 classListnerOld11)
        {
            _classListnerOld11 = classListnerOld11;
        }

        public void AddClassListnerOld12(ClassListnerOld12 classListnerOld12)
        {
            _classListnerOld12 = classListnerOld12;
        }

        public void DoSomething()
        {
            if (_classListnerOld11 != null)
            {
                _classListnerOld11.Listen("start", this);
            }
            if (_classListnerOld12 != null)
            {
                _classListnerOld12.Listen("start");
            }
        }
```

```
namespace Lection05
{
    internal class ClassListnerOld12
    {
        public void Listen(string s)
        {
        }
    }
}
```

```
namespace Lection05
{
    internal class ClassListnerOld11
    {
        public void Listen(string s, object o)
        {
        }
    }
}
```

```
namespace Lection05
{
    internal class ClassOldWay2
    {
        private List<InterfaceListner> _listners;

        public ClassOldWay2()
        {
            _listners = new List<InterfaceListner>();
        }

        public void AddListner(InterfaceListner listner)
        {
            _listners.Add(listner);
        }

        public void DoSomething()
        {
            foreach (var elem in _listners)
            {
                if (elem != null)
                {
                    elem.Listen("start", this);
                }
            }
        }
```

```
namespace Lection05
{
    internal interface InterfaceListner
    {
        void Listen(string s, object o);
    }
}
```

```
namespace Lection05
{
    internal class ClassListnerOld21 : InterfaceListner
    {
        public void Listen(string s, object o)
        {
            throw new NotImplementedException();
        }
    }
}
```

```
namespace Lection05
{
    internal class ClassListnerOld22 : InterfaceListner
    {
        public void Listen(string s, object o)
        {
            throw new NotImplementedException();
        }
    }
}
```

```
namespace Lection05
{
    internal class ClassChild : ClassParent
    {
        private event Func<string, int> ev2;

        public event Func<string, int> Ev2
        {
            add
            {
                if (value != null)
                {
                    ev2 += value;
                }
            }
            remove
            {
                ev2 -= value;
            }
        }
    }
}
```

```
namespace Lection05
{
    internal class ClassParent
    {
        protected event Action<string> ev1;

        public void AddCaller(Action<string> del)
        {
            ev1 += del;
        }

        public void DoSomething()
        {
            ev1?.Invoke("Start progress");
            //....
            ev1?.Invoke("Finish progress");
        }
    }
}
```

Анонимные методы

Анонимные методы

Анонимные методы

Анонимный метод – это блок кода, у которого нет имени и он используется для инициализации экземпляра делегата.
Создание анонимных методов является, по существу, способом передачи блока кода в качестве параметра делегата.
Использование анонимных методов позволяет сократить издержки на кодирование при создании делегатов, поскольку **не требуется** создавать отдельный метод.
Например, указание блока кода вместо делегата может быть целесообразно в ситуации, когда создание метода может показаться ненужным действием.

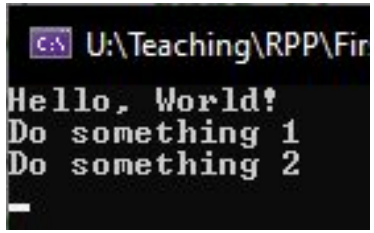
<делегат> = delegate(<параметры>) { // логика};

Анонимные методы

Анонимные методы

```
Del1 del = delegate
{
    Console.WriteLine("Do something 1");
    Console.WriteLine("Do something 2");
};

del();
```



Анонимные методы

Передача параметров

Если анонимный метод **использует** параметры, то они должны **соответствовать** параметрам делегата.
Если для анонимного метода **не требуется** параметров, то скобки с параметрами опускаются. В таком случае он соответствует любому делегату, который имеет тот же тип возвращаемого значения.
При этом даже если делегат принимает несколько параметров, то в анонимном методе можно вовсе **опустить** параметры.

Анонимные методы

Передача параметров

```
Del1 del = delegate
{
    Console.WriteLine("Do something 1");
    Console.WriteLine("Do something 2");
};

del();
```

```
Del3<int, string> del31 = delegate (int x) { return x.ToString(); };
Del3<int, string> del32 = delegate { return "sdfdfdfg"; };
```

Лямбда-выражения

Лямбда-выражения

Лямбда-выражения

Лямбда-выражение – это анонимная функция, с помощью которой можно создавать упрощенную запись анонимных методов для делегатов или строить деревья выражений.
Чтобы создать лямбда-выражение, необходимо указать входные параметры (если они есть) с левой стороны лямбда-оператора =>, и поместить блок выражений или операторов с другой стороны.

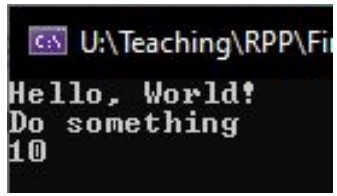
(input parameters) => expression

Лямбда-выражения

Лямбда-выражения

```
Del1 del = () => { Console.WriteLine("Do something"); };
del();

Action<int> act = (x) => { Console.WriteLine(x.ToString()); };
act(10);
```



Лямбда-выражения

Передаваемые параметры

Если лямбда имеет только **один** входной параметр и для него не требуется определять тип, то скобки можно **не ставить**, во всех остальных случаях они **обязательны**. Два и более входных параметра разделяются запятыми и заключаются в скобки.
При определении списка параметров допускается не указывать для них тип данных. Однако, могут быть случаи, когда компилятору бывает трудно или даже невозможно опредедлить типы входных параметров. В этом случае типы необходимо указывать в явном виде.
Отсутствие входных параметров задаётся **пустыми скобками**.

Лямбда-выражения

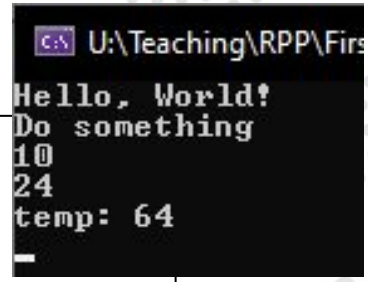
Передаваемые параметры

```
Del1 del = () => { Console.WriteLine("Do something"); };
del();

Action<int> act = (x) => { Console.WriteLine(x.ToString()); };
act(10);

Action<int> act2 = x => { Console.WriteLine(x.ToString()); };
act2(24);

Action<int, string> act3 = (f, t) => { f += 40; Console.WriteLine($"{t}: {f}"); };
act3(24, "temp");
```



Лямбда-выражения

Тело выражения

Тело лямбда-выражения заключается в фигурные скобки.
Если лямбда-выражение состоит из одного оператора, то скобки можно **опустить**.
Лямбда-выражение может возвращать результат. Возвращаемый результат можно указать после лямбда-оператора (=>), если в теле выражения прописан только один оператор. Если лямбда-выражение содержит несколько операторов, тогда нужно использовать оператор return, как в обычных методах.
Тело лямбды оператора может состоять из **любого** количества операторов, однако на практике обычно используется не более **двух-трех**.

Лямбда-выражения

Тело выражения

```
Action<int> act = (x) => Console.WriteLine(x.ToString());
Func<int, int> func1 = x => x * x;
Func<int, int> func2 = x => { return x * x; };
Del1 del = () =>
{
    Console.WriteLine("Do something 1");
    Console.WriteLine("Do something 2");
};
```

Деревья выражений

Деревья выражений

Деревья выражений

Деревья выражений представляют код в виде древовидной структуры, где каждый узел является выражением, например, вызовом метода или двоичной операцией, такой как x < y.
Существуют 2 основных способа создания деревьев выражений:

- через статические методы класса Expression;
- через лямбда выражения, компилирующееся в Expression.

Деревья выражений

Деревья выражений

Деревья выражений должны быть **неизменными**. Это означает, что если требуется изменить дерево выражений, следует создать новое дерево выражений путем копирования существующего и заменить узлы в нем. Для прохода по существующему дереву выражений можно использовать другое дерево выражений (паттерн посетитель).
Тип Expression<TDelegate> предоставляет метод Compile, который компилирует код, представляемый деревом выражений, в исполняемый делегат.

Деревья выражений

Класс Expression

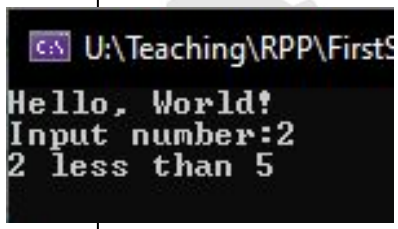
Для создания деревьев выражений с помощью API-интерфейса используется класс Expression. Этот класс содержит статические методы фабрики, позволяющие создать узлы дерева выражения конкретного типа, например, ParameterExpression, который представляет переменную или параметр, или MethodCallExpression, который представляет вызов метода. ParameterExpression, MethodCallExpression и другие зависящие от выражения типы также определяются в пространстве имен System.Linq.Expressions. Эти типы являются производными от абстрактного типа Expression.

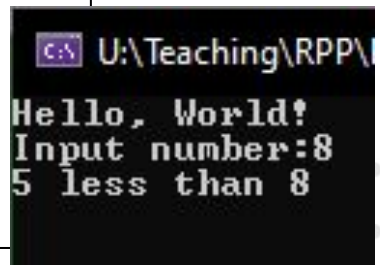
Деревья выражений

Класс Expression

```
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });

Func<int, bool> func1 = lambda1.Compile();
Console.Write("Input number:");
int n = Convert.ToInt32(Console.ReadLine());
if (func1(n))
{
    Console.WriteLine($"{n} less than 5");
}
else {
    Console.WriteLine($"5 less than {n}");
}
```





Деревья выражений

Лямбда-функции

Компилятор C# может создавать деревья выражений только на основе лямбда-выражений (или однострочных лямбда-функций). Они не могут анализировать лямбды операторов (или многострочные лямбды).
Когда лямбда-выражение назначается переменной с типом Expression<TDelegate>, компилятор выдает код для создания дерева выражений, представляющего лямбда-выражение.
Существуют ограничения на лямбда-выражения, которые могут быть преобразованы в деревья решения:

- Содержащие оператор присваивания
- Содержащие оператор dynamic
- Асинхронные
- С телом (фигурные скобки)

Деревья выражений

Лямбда-функции

```
Expression<Func<int, bool>> lambda = num => num < 5;
Expression<Func<int, bool>> lambda2 = num => { return num < 5; };
Expression<Action<int>> lambda3 = num => num = 45;
```

Спасибо за внимание!

Рады видеть Вас на наших мероприятиях!

Ульяновский государственный
технический университет

2021