Лекция ОПЕРАЦИИ ДЕЛЕГАТЫ СОБЫТИЯ

Операции

Представим, что надо сложить между собой четыре комплексных числа класса Complex. Пусть в классе реализован статичный метод Add, который складывает два объекта типа Complex и возвращает объект типа Complex. Тогда для того, чтобы сложить четыре комплексных числа придется написать следующую строку:

```
Var = Complex.Add( Var1, Complex.Add(Complex.Add ( Var2, Var3) , Var4);
```

Если бы для комплексных чисел (в классе Complex) была реализована операция сложения, код выглядел бы гораздо приятнее:

```
Var =Var1+Var2 +Var3 + Var4;
```

Операции в C# определяются как обычные методы. Компилятор и среда исполнения автоматически переводят выражения с операциями в соответствующую серию вызовов методов.

В C# есть большое количество предопределенных операций. *Для различных типов данных одна и та же операция может реализовывать различные действия. Это свойство называется перегрузкой операций.*

Операции следует определять только тогда, когда они упрощают выражения и когда класс или структура хранят какие-либо данные, для которых действие этой операции понятно. То есть, например, для класса Employee семантически не понятно действие оператора инкремента(emp++).

Все операции – это public static методы, их имена задаются по шаблону operatorop, где op – знак операции. Для операции сложения – operator+. Список параметров и их типы должны быть определены. Операции возвращают объект класса.

```
public static Time operator+(Time t1, Time t2)
{
int newHours = t1.hours + t2.hours;
int newMinutes = t1.minutes + t2.minutes;
return new Time(newHours, newMinutes);
}
```

Перегрузка операций сравнения производится попарно: < и >, <= и =>, == и !=. При перегрузке одной операции из пары, обязательно надо определить и вторую. Кроме того, при перегрузке операций равенства и неравенства необходимо определить виртуальный метод Equals, наследуемый от класса Object, чтобы при сравнении объектов с помощью метода Equals не получить результат, противоположный сравнению операцией равенства (==). Для той же цели перегружается ещё один метод наследуемый от Object – GetHashCode, для равных объектов хеш-код также должен быть одинаков.

Для логических операций && и || нет прямой перегрузки, для этого используются побитовые операторы. Перегрузив операторы &, |, true и false, можно определить логическое И и логическое ИЛИ. Пусть x и y переменные типа T, тогда логические операторы определяются следующим образом:

- x && y – будет выражаться через T.false(x) ? x : T.&(x,y), что означает, что если x в терминах класса T является ложью, то результат равен x, иначе в результате получим побитовое И x и y в терминах класса T.

- x || y – выражается через T.true(x) ? x : T.|(x,y), то есть если x в терминах класса T является истинной, то результат равен x, иначе получим побитовое ИЛИ x и y в терминах класса T.

При задании операторов преобразования типов данных необходимо указать явным или неявным образом будет осуществляться преобразование. Для задания явного преобразования используется ключевое слово **explicit**, для неявного – **implicit**:

```
public static explicit operator Time (int minutes) – явное преобразование типа int в Time;
public static explicit operator Time (float minutes) – явное преобразование типа float к Time;
public static implicit operator int (Time t1) – неявное преобразование типа Time к типу int;
public static explicit operator float (Time t1) – явное преобразование типа Time к типу float;
```

public static implicit operator string (Time t1) – если для класса предусмотрена перегрузка преобразования в строку, то необходимо аналогично перегрузить метод ToString(), унаследованный от класса Object.
Операции можно перегружать несколько раз в зависимости от параметров:

```
public static Time operator+ (Time t1, int hours) {…}

public static Time operator+ (Time t1, float hours) {…}
```

Создание и использование делегатов

Делегаты позволяют вызывать методы неявно, не используя прямое обращение по имени. Фактически, делегаты предназначены для ситуаций, когда нужно передать метод другому методу.
Примером одной из таких ситуаций может быть возникновение события. Например, в программировании графического интерфейса – пользователь нажал на кнопку, сгенерировал событие – необходимо предусмотреть обработку этого события. Написать метод и метод каким-то образом связать с этим событием.
Самый простой способ передать метод в качестве параметра:

```
void Method1() {…} //объявление Method1
MyMethod(Method1); //вызов MyMethod, в качестве параметра, которому передан указатель на Method1
```

Такой способ возможен для некоторых языков, например для C, C++. С точки зрения идеологии платформы .NET Framework, такой прямой подход вызывает проблемы с безопасностью типов и игнорирует тот факт, что в объектно-ориентированном программировании методы редко существуют в изоляции: для вызова метода обычно должен быть создан объект класса. В связи с этим, в C# такой подход синтаксически не допустим. Вместо этого, когда нужно передать метод, подробную информацию о нем следует поместить в специальную оболочку – делегат. Делегат содержит подробную информацию о методе.
Сначала дается определение делегата, который нужно будет использовать. Его определение означает сообщение компилятору, какого рода метод будет представлять делегат. Затем необходимо создать экземпляр этого делегата. Синтаксис объявления делегатов:

```
delegate void VoidOpetation(int x);
```

В данном случае определен делегат, каждый экземпляр которого может содержать ссылку на метод, принимающий один параметр типа int и возвращающий тип void. При определении делегата ему сообщается полная сигнатура метода, который он может представлять. Таким образом, экземпляр делегата может ссылаться на любой метод любого объекта, если сигнатура этого метода совпадает с сигнатурой делегата.

Синтаксис определения делегата подобен определению метода за исключением того, что за таким определением не следует тело метода, и перед определением используется ключевое слово delegate. Объявление делегата можно поместить в любое место, где может находиться объявление класса – то есть внутри какого-либо класса, либо вне любого класса, либо в пространстве имен объекта высшего уровня. Перед делегатом можно использовать любой модификатор доступа.

Пример:

```
private delegate string MyDelegate(int x);
private string MyMethod (int x){
return x.ToString();
}
static void Main(string[] args)
{
MyDelegate myD=new MyDelegate(MyMethod); //создаем делегат и связываем с MyMethod
Console.WriteLine(myD(5)); //строка вызовет метод MyMethod с параметром 5
}
```

События

События позволяют объектам зарегистрироваться на интересующие его изменения в другом объекте. У события есть отправитель – тот, кто генерирует событие, и получатель. Получателем события может выступать любое приложение, объект или компонент, который нуждается в уведомлении, когда что-то происходит.

Отправитель события ничего не знает о том, кто его получает. Где-то внутри получателя есть метод, который отвечает за обработку события. Этот обработчик события должен запускаться всякий раз, когда возникает зарегистрированное для него событие. Как раз здесь и нужны делегаты. Поскольку отправитель не знает, кто будет получателем, между ними не может быть прямых ссылок, нужен посредник. Таким посредником является делегат. Отправитель определяет делегат, который будет использован получателем. Получатель регистрирует обработчик события, привязывает метод обработки к событию.

Пример. В windows приложении разместим на форме кнопку btnOne и создадим для нее обработчик события нажатия на кнопку. Для этого два раза нажмем на кнопку, откроется файл, в котором будет сгенерирован заголовок метода btnOne_Click:

```
private void btnOne_Click(object sender, EventArgs e){…},
```

в который необходимо вписать его реализацию. Также автоматически в коде появится строка

```
btnOne.Click+=new EventHandler(btnOne_Click);
```

Это означает, что когда произойдет событие Click для кнопки btnOne, то должен быть выполнен метод btnOne_Click. **EventHandler** – это делегат, который использует событие для назначения обработчика (btnOne_Click) событию (Click). Для добавления метода к списку делегатов использовался оператор +=.

Событию можно назначить больше одного обработчика. Однако нет никакой гарантии того, в каком порядке будут вызваны методы, с которыми связан делегат. С разными событиями можно связать один и тот же метод. Допустим, на форме есть кнопка btnTwo, свяжем событие Click этой кнопки с методом btnOne_Click:

```
btnTwo.Click+=new EventHandler(btnOne_Click);
```

Делегат EventHandler определен средой .NET и находится в пространстве имен System. Все события, определенные .NET, используют этот делегат. Делегат EventHandler использует два параметра object и EventArgs. Первый параметр – это объект, который сгенерировал событие, в данном примере – это кнопка btnOne или btnTwo. Внутри метода с помощью первого параметра можно узнать, кто именно сгенерировал событие. Второй параметр – это объект, содержащий другую полезную информацию, может быть любого типа, унаследованного от класса EventArgs.

Важно отметить, что обработчики событий всегда возвращают void.

Рассмотрим, как создавать свои собственные события. Для определения события отправитель объявляет делегат и связывает его с событием.

```
public delegate void StartPumpCallback(object sender, CoreOverheatingEventArgs args);
private event StartPumpCallback CoreOverheating;
```

Здесь CoreOverheatingEventArgs – класс, наследник от класса System.EventArgs для передачи дополнительных параметров:

```
public class CoreOverheatingEventArgs : EventArgs
{
private readonly int temperature;
public CoreOverheatingEventArgs(int temperature)
{
this.temperature = temperature;
}
public int GetTemperature()
{
return temperature; }
```

```
}
```
Подписчики определяют метод, который будет вызываться при возникновении события. Если событие еще не создано, то подписчик определяет делегат, ссылающийся на метод при создании события. Если событие уже существует, подписчик добавляет делегат, вызывающий метод при возникновении события.

```
ElectricPumpDriver ed1 = new ElectricPumpDriver();
PneumaticPumpDriver pd1 = new PneumaticPumpDriver();
…
CoreOverheating = new StartPumpCallback(ed1.StartElectricPump); //делегат, который //указывает на метод StartElectricPump объекта ed1, подписывается на событие //CoreOverhating

CoreOverheating += new StartPumpCallback(pd1.SwitchOn); //делегат, который //указывает на метод SwitchOn объекта pd1, также подписывается на событие //CoreOverhating
```

Для уведомления подписчиков необходимо вызывать событие.

```
public void SwitchOnAllPumps()
{
CoreOverheatingEventArgs e=new CoreOverheatingEventArgs(37);
if(CoreOverheating != null) //если на событие кто-то подписан, то генерируем //событие
CoreOverheating(this,e); //событие вызовет два метода //StaertElectricPump и SwithcOn, в которые передаст параметр this и e

}
```

Если событие происходит, то вызываются все делегаты. Заметим, что сначала надо проверить, есть ли хоть один подписчик, так как при отсутствии подписчиков вызов делегата сгенерирует исключение.
Кроме того, что произошло событие, подписчикам бывает интересно, при каких условиях оно произошло. Для этого события могут передавать параметры. Для передачи параметров событиями в C# выделен отдельный класс System.EventsArg.