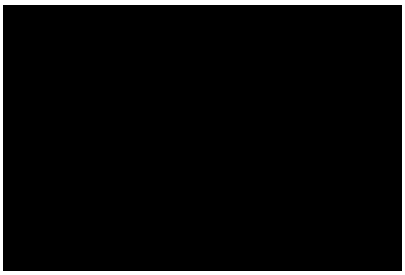
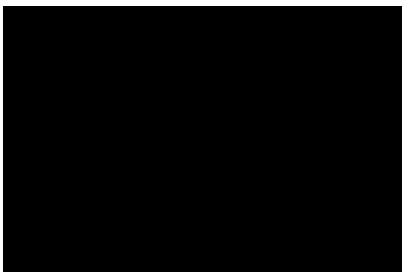
Если бы классы позволяли только описывать свойства объектов, то пользы от них было бы мало. Безусловно, мы хотим, чтобы объекты могли выполнять какие-то действия:

- выводили слова на экран;
- решали задачи;
- копировали данные с веб-узла;
- регулировали яркость фотографии;
- ... и выполняли еще тысячи операций.

А теперь вспомните уроки, на которых вас учили сложению чисел. Вероятней всего, сначала учитель объяснял, **КАК** это делается, описывая, возможно, каждый шаг. Другими словами, в первую очередь рассматривался **МЕТОД** решения задачи определенного типа, то есть набор пошаговых инструкций, поясняющих порядок выполнения действий.



А следуя уже известному вам методу решения задач, вы справитесь с любыми похожими задачами.



Подобным образом, для того чтобы компьютер выполнял нужные нам действия, надо написать код, разъясняющий порядок их выполнения, то есть описать метод, который будет использоваться для выполнения необходимых действий, а затем применять этот метод при решении задачи подобного типа.

В языке C# метод позволяет описать порядок выполнения определенных действий. Описание метода называется программным кодом или просто кодом. В нужный момент метод может быть вызван. Компьютер читает код метода и выполняет именно то, что предписано.

Вот пример простейшего метода:

```
void SayHello()
\{
\}
```

Этот метод ничего не делает, потому что между фигурными скобками ничего не находится. Давайте зададим здесь какое-нибудь действие.

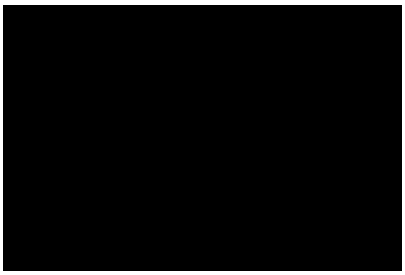
```
void SayHello()
\{
    Console.WriteLine("Hello");
\}
```

(Придадим коду аккуратный вид и при помощи клавиши TAB создадим в методе отступы.)

Если вызвать этот метод, то на экран будет выведено слово "Hello". Задача не слишком сложная, поэтому и метод довольно прост.

1 Что означает VOID?

Странное слово void в приведенном выше примере может несколько озадачить, и вы можете поинтересоваться, зачем вообще его использовали. Вкратце остановимся на этом. Вспомните фильм о космических путешествиях: представьте безграничное пустое пространство во вселенной. Void означает пустоту, "ничего".



Использование слова void перед именем метода означает, что, когда завершается выполнение метода, возвращается пустое значение, то есть по завершении определенных действий, которые выполняет метод, он никаких значений не возвращает. В приведенном выше примере метода "SayHello"нас это вполне устраивает, поскольку после написания слова "Hello"возвращать ничего не нужно. Все необходимое было сделано во время работы метода.

Вы сможете лучше понять смысл этого слова, когда мы будем рассматривать случаи, где метод должен будет возвращать некоторый результат. Подобный пример появится немного позже.

2 Вызов метода

Написав приведенный выше код, мы объяснили компьютеру, КАК выводить приветствие на экран, но не сказали, КОГДА ему нужно воспользоваться приобретен-

ным умением.

Это выглядит так, как если бы на уроке вам рассказали, как складывать числа, после чего вы записали метод, но решить задачку или пример вас не попросили.

Чтобы заставить компьютер выполнить действие, нужно вызвать метод. Для этого надо просто написать имя метода и поставить рядом круглые скобки:

```
SayHello();
```

Когда компьютер встречает эту строку, он понимает, что нужно выполнить команду, но встроенного метода с таким именем у компьютера нет, поэтому он лихорадочно начинает искать написанный вами метод с именем "SayHello". Если найти такой метод удается, он быстро выполняет все предписания, записанные в коде метода.

Затем возвращается к самому началу и, довольный успехом, получает вашу благодарность.

3 Как выполняется метод? Параметры метода

В предыдущем разделе мы познакомились с методами вкратце, но о них можно рассказать гораздо больше. Теперь рассмотрим более подробно, как передавать методу значения и получать значения, созданные при работе метода.

Предположим, вам потребовалось, чтобы компьютер вывел на экран следующий текст:

```
Hello Jo
Hello Sam
Hello You
```

Один из возможных способов — написать отдельный метод для каждого случая:

```
void WriteHelloJo()
\{
    Console.WriteLine("Hello Jo");
\}
void WriteHelloSam()
\{
    Console.WriteLine("Hello Sam");
\}
void WriteHelloYou()
\{
    Console.WriteLine("Hello You");
\}
```

Затем необходимо вызвать их следующим образом:

Строительный блок: Объявлениеблок: Объявление и вызов метода	
Чаще всего в классе присутствует один или несколько методов. Каждый из них выполняет определенное действие. Методами они называются потому, что именно в них описывается метод выполнения действий – пошаговые инструкции, задающие порядок выполнения операций. Строка, начинающаяся с двух символов "слеш"(//), называется комментарием. Комментарии только поясняют код, но не влияют на выполнение программы.Операция "+"определена над строками. Она называется сцеплением строк, или конкатенацией. Результатом операции является приписывание второй строки в конец первой.Операция "+"определена над строками. Она называется сцеплением строк, или конкатенацией. Результатом операции является приписывание второй строки в конец первой. Как уже говорилось, объекты класса Как уже говорилось, объекты класса PersonPerson могут объявляться и создаваться в методах другого класса. Когда встречается вызов метода Когда встречается вызов метода ShowFullNameShowFullName, компьютер находит в классе PersonPerson метод с таким именем и – шаг за шагом – выполняет описанные в нем действия.	class Person // Поля public string firstName; public string lastName; // Метод public void ShowFullName() Console.WriteLine("Name is "+ firstName + + lastName); Person Petr; Petr = new Person(); Petr.firstName = "Petr"; Petr.lastName = "Ivanov"; Petr.ShowFullName();class Person { // Поля public string firstName; public string lastName; // Метод public void ShowFullName() { Console.WriteLine("Name is "+ firstName + + lastName); } } Person Petr; Petr = new Person(); Petr.firstName = "Petr"; Petr.lastName = "Ivanov"; Petr.ShowFullName();

```
WriteHelloJo();
WriteHelloSam();
WriteHelloYou();
```

Но ведь все три метода очень похожи. А что если написать один метод WriteHello, дополнив его соответствующими параметрами, и при каждом вызове просто передавать значение параметра, отличающее один вызов от другого?

Вот как это можно сделать:

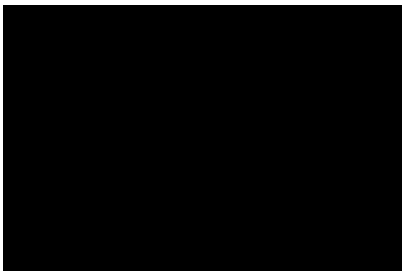
```
void WriteHello(string someName)
\{
    Console.WriteLine("Hello " + someName);
\}
```

и затем вызвать метод следующим образом:

```
WriteHello("Jo");
WriteHello("Sam");
WriteHello("You");
```

Как видим, код позволяет сэкономить и занимаемое пространство, и затраченные усилия. Всегда старайтесь делать код как можно более кратким – чем короче программа, тем умнее программист.

Примечание редактора. Умный программист пишет не только короткий, но и понятный код. Хороший код всегда содержит комментарии умного программиста.



Напишем метод подобным "умным"образом:

```
void WriteHello(string someName)
\{
    Console.WriteLine("Hello " + someName);
\}
```

Фактически мы говорим: "Каждый раз при вызове этого метода я буду подставлять строку символов с каким-либо именем. Любая *подставляемая* строка должна выводиться после слова "Hello".

Код в скобках (string someName) называется параметром. Параметр позволяет подставлять значение в метод при его вызове.

Когда вас обучали сложению, учитель не рассказывал о сложении всех существующих пар чисел, он просто научил методу и затем задавал разные задачи: "Сложите 2 и 5, а теперь 7 и 3". Это похоже на то, как если бы вам излагали метод сложения чисел, используя параметры: неважно, какие значения у параметров, - зная метод, всегда можно найти ответ для заданных значений.

Компьютеру все равно, какое имя вы присвоите параметру, важно, чтобы оно было единым при использовании во всем методе. Например, следующий код будет выполнен правильно:

```
void WriteHello(string x)
\{
    Console.WriteLine("Hello " + x);
\}
```

А этот с ошибкой:

```
void WriteHello(string someName)
\{
    Console.WriteLine("Hello " + someBodiesName);
\}
```

Вы заметили: параметры someName и someBodiesName отличаются — наш "электронный друг"не разберется в этой путанице и "разгневается".

Кроме того, в методе можно использовать не один параметр, а несколько, но нужно обязательно разделить их запятыми:

```
void WriteHello(string firstName, string lastName)
\{
    Console.WriteLine("Hello " + firstName + " " + lastName);
\}
```

А при вызове метода необходимо подставить правильное количество значений:

```
WriteHello("Petr", "Ivanov");
```

В данном случае на экран будет выведен текст "Hello Petr Ivanov".

Ошибки в задании типов параметров

Представьте, что когда вы впервые изучали сложение чисел, учитель неожиданно задал вам такую задачу: "Сложите число 5 и *цветок* ".

Как бы вы поступили? Наверное, ответили бы, что цветок — это не число, и выполнить сложение невозможно. Правильно.

Подобным же образом компьютеру не понравится, если вы укажете значение *неверного типа*. Такая ошибка часто встречается у программистов. Поэтому если что-то не получается, вернитесь к началу и убедитесь, что значения, указанные вами при вызове метода, имеют тип, соответствующий тому, который определен в самом методе.

Примечание редактора. Внимательный читатель спросит: "Складывая строку с числом при вызове метода Console.WriteLine, не делаем ли мы ту же ошибку, как в случае сложения цветка с числом?". Здесь мы полагаемся на то, что компьютер (точнее, компилятор языка C#) умеет справляться с этим – сначала он автоматически преобразовывает число в строку и только потом выполняет операцию сложения — сцепление строк. Приводимые до сих пор методы были void-методами. Они выводили некий текст на экран и затем возвращались назад к месту их вызова, как бы говоря: "Свое дело я сделал и вернулся в исходную точку". Однако иногда необходимо вернуть некоторое значение в точку вызова метода. В этих случаях следует написать метод, который будет возвращать значение, отличное от void.

Приведем пример. Напишем метод, который будет выполнять поиск количества конечностей указанного животного и затем отправлять полученное число туда, откуда этот метод был вызван.

Строительный блок: Параметры	
Чтобы в методе выбиралось нужное значение, необходимо указать соответствующие параметры.Каждый раз при вызове метода мы должны убедиться, что подставляем правильный тип значений в параметры. В приведенном примере мы подставляем два целых числа, так как параметры метода "LuckyNumber"были определены как целые числа.Каждый раз при вызове метода мы должны убедиться, что подставляем правильный тип значений в параметры. В приведенном примере мы подставляем два целых числа, так как параметры метода "LuckyNumber"были определены как целые числа.	class Person // Поля string firstName; string lastName; // Метод public void LuckyNumber(int numberOfTeeth, int age) Console.WriteLine("Счастливое число"+ numberOfTeeth * age); Person Petr; Petr = new Person(); Petr.LuckyNumber(24, 14);class Person { // Поля string firstName; string lastName; // Метод public void LuckyNumber(int numberOfTeeth, int age) { Console.WriteLine("Счастливое число"+ numberOfTeeth * age); } } Person Petr; Petr = new Person(); Petr.LuckyNumber(24, 14);

Следует помнить, что с помощью метода мы показываем компьютеру, КАК выполнять определенное действие. Сначала я напишу то, чего хочу добиться от него, на русском языке, а затем на C#:

- если животное, о котором мы говорим, — слон, то number of legs = 4 ;
- иначе, если животное, о котором мы говорим, — индейка, то number of legs = 2 ;
- иначе, если животное, о котором мы говорим, — устрица, то number of legs = 1 ;
- иначе, если мы говорим о каких-либо других животных, то number of legs = 0.

```
int NumberOfLegs(string animalName)
\{
    if (animalName == "слон") //Если название животного - слон
    \{
        // Возвращаемое значение 4
        return 4;
    \}
    else if (animalName == "индейка") //Иначе, если животное - индейка
    \{
        // Возвращаемое значение 2
        return 2;
    \}
    else if (animalName == "устрица")//Иначе, если животное - устрица
    \{
        // Возвращаемое значение 1
```

```
        return 1;
    \}
    else //Иначе (при всех других условиях)
    \{
        // Возвращаемое значение 0
        return 0;
    \}
\}
```

Теперь мы можем вызвать метод. Давайте сделаем это дважды:

```
int i;
    //Переменная "i" будет хранить значение числа конечностей.
i = NumberOfLegs("индейка");
    //Теперь i = 2, получив значение, возвращенное методом NumberOfLegs
Console.WriteLine("У индейки конечностей - " + i);
i = NumberOfLegs("обезьяна");
    //Теперь i = 0. Догадайтесь, почему!
Console.WriteLine("У обезьяны конечностей - " + i);
```

На экран будут выведены тексты: "У индейки конечностей — 2"и "У обезьяны конечностей — 0". Итак, метод возвращает значение, которое можно принять в точке его вызова.

Мы определили метод именно так:

```
int NumberOfLegs(string animalName)
\{
...
\}
```

А не так:

```
void NumberOfLegs(string animalName)
\{
...
\}
```

И не так:

```
string NumberOfLegs(string animalName)
\{
...
\}
```

Дело в том, что нам необходимо, чтобы в данном случае метод возвращал **целое число** — не "пустое"значение (void), не строку букв, а именно целое число. А для работы с целыми числами используется тип данных Integer, или int в сокращенном варианте.

При написании метода мы всегда указываем тип данных, возвращаемых этим методом. Если возвращать значение не надо, используется void – для возврата пустого значения.

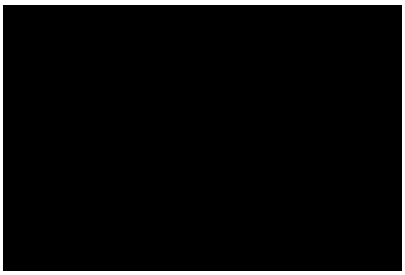
```
void JustWriteSomething(string someThing)
\{
  Console.WriteLine(someThing);
\}
```

И наконец: возможно, вы догадались, что слово return возвращает значение. Когда компьютер встречает это слово, происходит выход из метода и возврат запрашиваемого значения.

Строительный блок: Возвращаемые значения	
Иногда необходимо получить значение из метода. В таком случае вместо указания типа voidvoid, сообщающего, что "никакого значения возвращено не будет мы указываем определенный тип данных, возвращаемых методом.Возвращаемые значения автоматически становятся доступными в любом месте, где мы вызываем метод.Возвращаемые значения автоматически становятся доступными в любом месте, где мы вызываем метод. Например, используя возвращенное значение, мы могли бы сначала сохранить ответ в переменной части выражения и затем использовать значение переменной в отдельном выражении.Например, используя возвращенное значение, мы могли бы сначала сохранить ответ в переменной части выражения и затем использовать значение переменной в отдельном выражении. Или мы могли бы вызвать метод непосредственно в выражении Или мы могли бы вызвать метод непосредственно в выражении WriteLineWriteLine.	class Person // Поля string firstName; string lastName; // Метод int LuckyNumber(int numberOfTeeth, int age) return (numberOfTeeth * age); Person Anna; Anna = new Person(); int num = Anna.LuckyNumber(24, 14); Console.WriteLine("Счастливое число Анны:"+ num); Console.WriteLine("Счастливое число Анны:"+ Anna.LuckyNumber(24, 14));class Person { // Поля string firstName; string lastName; // Метод int LuckyNumber(int numberOfTeeth, int age) { return (numberOfTeeth * age); } } Person Anna; Anna = new Person(); int num = Anna.LuckyNumber(24, 14); Console.WriteLine("Счастливое число Анны:"+ num); Console.WriteLine("Счастливое число Анны:"+ Anna.LuckyNumber(24, 14));

Доступ к методам, аналогично доступу к полям класса, регулируется с помощью ключевых слов. По умолчанию все методы будут рассматриваться как private (закрытые), то есть они применяются только внутри своего класса. Чтобы разрешить их использование для других классов, можно добавить слово public в начало объявления метода.

```
public void JustWriteSomething(string someThing)
\{
    Console.WriteLine(someThing);
\}
```



В реальном мире людям запрещается входить в некоторые помещения без специального разрешения. Например, в ресторанах только повара и официанты могут проходить на кухню – это закрытая зона. В то же время обеденный зал предназначен для свободного доступа, и в нем могут находиться любые лица. Подобным же образом некоторый код закрыт для других классов.

Мы уже рассматривали пример с закрытыми и открытыми полями. Дополним его: введем закрытые (private) и открытые (public) *методы* в класс Animal и затем пытаемся обратиться к ним из класса.

```
class Animal
\{
    //Поля
    public string kindOfAnimal;
    public string name;
    public int numberOfLegs;
    public int height;
    public int length;
    public string color;
    bool hasTail;
    protected bool isMammal;
    private bool spellingCorrect;
    //Методы
    // Открытый метод, получающий информацию о том, чем питается животное
    public string GetFoodInfo()
    \{
    // Представим, что здесь расположен код, выполняющий поиск по базе данных
    ...
    \}

    // Закрытый метод для проверки правильности написания вида животного
    private void SpellingCorrect()
    \{
    // Представим, что здесь расположен код для проверки правописания
    ...
    \}

    // Защищенный метод, определяет существование данного вида животного
    protected bool IsValidAnimalType()
    \{
    //код для проверки существующих видов животных
    ...
    \}
```

```
\}

class Zoo
\{
    Animal a = new Animal ();
    a.name = "Kangaroo";
    string food;
    bool animalExists;

    // Следующий код будет выполнен успешно, поскольку классу "Zoo" разрешено
    // обращаться к открытым методам в классе "Animal"
    food = a.GetFoodInfo(); // Вызов открытого метода

    // Обе следующие строки НЕ будут выполнены, поскольку классу "Zoo"
    // не разрешено обращаться к закрытым или защищенным методам
    a.spellingCorrect(); // Попытка вызова закрытого метода
    animalExists = a.IsValidAnimalType(); // Попытка вызова защищенного метода
\}
```

Очень часто встречаются классы с особым типом метода, называемым **"конструктором"**. С точки зрения синтаксиса (правил языка) его особенность состоит в том, что имя *метода-конструктора совпадает с именем класса* и в объявление конструктора не включается *тип возвращаемого значения*. Содержательная специфика связана с предназначением конструктора — он нужен для создания (конструирования) объекта. Использование этого метода в классах помогает приобрести хороший практический опыт.

Примечание редактора. Классов без конструктора не бывает, поскольку объект класса можно создать только путем вызова конструктора класса. Даже если программист не добавит в класс конструктор, это будет сделано по умолчанию, но параметров такой конструктор не имеет. Полезно иметь в классе конструктор с параметрами, роль которых уже пояснялась. Подобных конструкторов может быть несколько.

```
class Person
\{
  // Поля
  string firstName;
  string lastName;
  // Метод-конструктор для класса Person
  public Person()
  \{
    firstName = "Johnny";
    lastName = "Rocket";
  \}
\}
```

Метод-конструктор вызывается по-особому: при каждом создании экземпляра класса с помощью конструкции new.

Напоминание:

Под "экземпляром класса"мы понимаем *определенный* объект класса. Например, в одном из предыдущих разделов мы выделили "Гориллу Джереми"как опре-

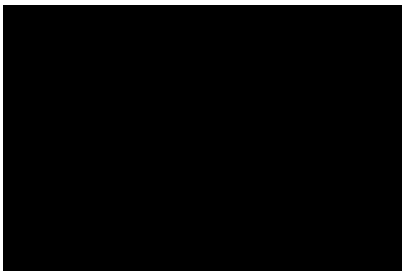
деленный объект, или экземпляр класса Animal.

Итак, если мы выполним следующий код:

```
Person p = new Person();
Console.WriteLine(p.lastName);
```

то в результате на экране появится слово "Rocket". Написав конструкцию new Person(), *мы тем самым дали указание компьютеру вызвать конструктор класса* Person для создания нового объекта этого класса. Он будет связан с переменной p, у которой задано значение "Rocket"для поля lastName.

Приведем аналогичный пример из реальной жизни. В некоторых странах новорожденного регистрируют, согласно закону, еще в родильном доме, чтобы он как можно скорее стал членом общества и получил все гражданские права.



Это действие можно сравнить с методом-конструктором, выполняемым для класса. Прежде чем новый экземпляр класса сможет что-либо сделать, выполняется метод-конструктор. В него можно включить любые планируемые к выполнению действия, прежде чем объект будет считаться "готовым к жизни".

Конструкторы с параметрами

В конструктор можно включить параметры. Приведем пример класса с двумя различными конструкторами:

```
class Person
\{
  // Поля
  string firstName;
  string lastName;
  // Первый метод-конструктор
  public Person()
  \{
    firstName = "Johnny";
    lastName = "Rocket";
  \}
  // Второй метод-конструктор
  public Person(string f, string l)
  \{
    this.firstName = f;
    this.lastName = l;
```

```
  \}
\}
```

Таким образом, мы получили *два разных способа конструирования объекта*. Например, этот:

```
Person p = new Person();
```

В таком случае в поле p.lastName будет автоматически подставлено значение "Rocket". Или этот:

```
Person p = new Person("Petr", "Ivanov");
```

Тогда в поле p.lastName будет подставлено значение "Ivanov".

Слово this относится к **объекту, который мы создаем**, то есть фактически указывается: "подставлять в поле имени и фамилии *этого объекта* любые значения, передаваемые методу-конструктору".

Примечание редактора. В данном случае слово this *можно добавить и в первый конструктор – для уточнения имени полей. Ничего не изменится, если во втором конструкторе имена полей будут указываться без* this. *Но иногда такое уточнение необходимо. Например, если во втором конструкторе имя параметра задавать не одной буквой* f, *а именем* firstName, *отражающим суть параметра, то без уточнения* this *в имени поля не обойтись, иначе компьютер запутается, не понимая, где имя поля, а где имя параметра метода.*

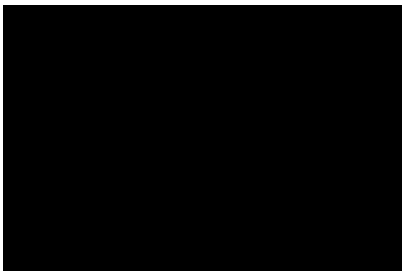
4 События

В реальном мире события происходят непрерывно, причем некоторые *от нас* совсем *не зависят*. Например, восход и закат солнца (хотел бы я посмотреть, как вы пытаетесь заставить солнце вставать и садиться). *Другие* события мы вызываем сами: скажем, заставляем громкоговоритель издавать звуки.

Рассмотрим несколько событий, которые часто встречаются в мире компьютеров:

- нажатие на кнопку, изображенную на экране монитора;
- истечение времени таймера;
- перемещение мыши;
- нажатие клавиши на клавиатуре.

Очевидно, что, нажав на кнопку, мы хотим заставить компьютер выполнить определенное действие. (Если нет – зачем вообще ее трогать?) Но компьютер ждет не только подтверждения, что нажатие этой кнопки имеет для вас какое-то значение, но и указания на действие, которое нужно выполнить.



Рассмотрим подробнее этот пример, поскольку нажатие на кнопку, возможно, является наиболее распространенным событием, и детально разберем порядок работы. Допустим, в вашей программе есть объект — кнопка с именем mrButton, *на которой написано* "Нажми меня".

По ходу обсуждения **попробуйте все делать сами**. Для начала:

- Запустите Visual C# Express.
- Создайте новый проект приложения Windows: в меню File ("Файл") выберите Create project ("Создать проект") и затем тип проекта Windows Application ("Приложение Windows Forms").
- В Visual C# Express откроется несколько файлов, где содержится "скелет"кода программы.
- В окне обозревателя решений справа (в списке всех файлов) удалите файл с именем Form1.cs.
- Дважды щелкните имя файла Program.cs и удалите весь автоматически вставленный "скелет"кода.
- Чтобы создать программу с экземпляром кнопки, наберите следующий код в окне Program.cs так, как показано ниже (написание слов *курсивом или жирным шрифтом можно не учитывать*.

```
using System;
using System.Windows.Forms;
class MyButtonClass: Form
\{
    private Button mrButton;
    // Метод-конструктор
    public MyButtonClass()
    \{
        mrButton = new Button();
        mrButton.Text = "Нажми меня";
        this.Controls.Add(mrButton);
    \}
    // Основной метод
    static void Main()
    \{
        Application.Run(new MyButtonClass());
    \}
\}
```

Выполните программу при помощи клавиши F5 (или щелкните по зеленой кнопке "Выполнить"). Если возникнут сообщения об ошибках, тщательно проверьте, нет ли опечаток в коде. Если программа будет выполнена успешно, вы увидите форму с кнопкой "Нажми меня". Пока при нажатии на кнопку никаких действий происходить не будет. Конечно, вы ожидали другого результата, но все еще впереди.

5 Событие нажатия кнопки. Указание действия в случае события

Теперь мы должны задать метод, выполняющий действие при нажатии на кнопку. Такие методы называются обработчиками событий, поскольку они именно "обрабатывают"событие. В приведенном ниже примере код обработчика события просто изменяет надпись на кнопке, поэтому он совсем короткий:

```
using System;
using System.Windows.Forms;
class MyButtonClass: Form
\{
    private Button mrButton;
    // Метод-конструктор
    public MyButtonClass()
    \{
        mrButton = new Button();
        mrButton.Text = "Нажми меня!";
        this.Controls.Add(mrButton);
    \}
    // Основной метод
    static void Main()
    \{
        Application.Run(new MyButtonClass());
    \}
    // Метод-обработчик событий
    void MyButtonClickEventHandler(object sender, EventArgs e)
    \{
        mrButton.Text = "Вы нажали меня!";
    \}
\}
```

Ваша программа еще выполняется?

- Остановите ее (нажмите на кнопку X в верхнем правом углу окна, в котором открыта форма).
- Добавьте в программу выделенный код и нажмите на клавишу F5 для выполнения измененной программы.
- Попробуйте теперь щелкнуть по кнопке "Нажми меня". И теперь ничего не происходит?!

Поскольку вы уже прочитали те страницы, где мы говорили про методы, то должны узнать основную структуру приведенного выше метода. Слово void означает, что по его завершении ничего не возвращается. Мы назвали этот метод MyButtonClickEventHan

Возможно, то, что вы видите, кажется немного странным. Вы понимаете, что в скобках присутствуют два параметра, но почему у них такие необычные типы (object sender, EventArgs e)? К сожалению, с методами обработчиков событий *нельзя использовать собственные типы параметров*. Когда у кнопки возникает событие "Click она посылает сообщение о нем операционной системе, а та находит и вызывает соответствующий обработчик события. При вызове такого метода система сама определяет типы параметров и передает обработчику их значения. *Это факт, с которым* мы ничего поделать не можем.

Поэтому придется просто смириться и всегда использовать ожидаемые типы параметров с обработчиком событий. Очень часто подставляемые параметры имеют тип object и EventArgs. В приведенном выше примере мы выбрали имена параметров sender и e, но могли бы выбрать любые другие — для компьютера важны имена *типов* этих параметров. Например, следующий код будет работать точно так же, как и код, рассмотренный в предыдущем примере. Можете проверить это сами, изменив имена параметров в вашей программе на x и y.

Примечание редактора. Ничего удивительного: эти параметры вообще не используются в тексте обработчика события. Но в ряде случаев они полезны. Например, при работе с мышкой обработчику события могут быть переданы координаты объекта, на котором нажата кнопка мыши.

```
void MyButtonClickEventHandler(object x, EventArgs y)
\{
  mrButton.Text = "Вы нажали меня!";
\}
```

В первом параметре обычно содержится некоторая информация об объекте, который инициировал событие. Второй параметр относится к данным о самом событии.

Очень важно знать следующее: система всегда подставляет некоторые значения в эти два параметра, но зачастую необходимости в их использовании нет — они отправляются в метод обработчика событий "на всякий случай".

6 Подключение метода обработчика событий к событию

Вы удивляетесь, почему ничего не происходит при нажатии на кнопку? Дело в том, что указанный метод вызывается только тогда, когда мы *свяжем с ним событие нажатия на кнопку*, указав в программе: при нажатии на кнопку необходимо перейти к определенному обработчику событий.

Когда в программе используются разные кнопки и несколько обработчиков событий, без такого уточнения не обойтись, так как компьютер должен знать, какой именно метод следует выполнять при нажатии на определенную кнопку.

Код для связывания события объекта с методом обработчика события выглядит тоже несколько странно.

```
using System;
using System.Windows.Forms;
class MyButtonClass : Form
\{
    private Button mrButton;
    // Метод-конструктор
    public MyButtonClass()
    \{
        mrButton = new Button();
        mrButton.Text = "Нажми меня";
        // Код для связывания события объекта с методом обработчика события
        mrButton.Click += new System.EventHandler(MyButtonClickEventHandler);
        this.Controls.Add(mrButton);
    \}
    // Основной метод
    static void Main()
    \{
        Application.Run(new MyButtonClass());
    \}
    // Метод-обработчик событий
    void MyButtonClickEventHandler(object sender, EventArgs e)
    \{
        mrButton.Text = "Вы нажали меня!";
    \}
\}
```

С компьютерного языка это можно перевести следующим образом:

"Путем нажатия на кнопку mrButton надо связать событие Click с методом обработчика событий, который называется MyButtonClickEventHandler ".

При нажатии на кнопку приведенная выше строка кода позволяет системе вызвать метод обработчика событий, после его выполнения надпись на кнопке меняется на "Вы нажали меня!".

Чтобы использовать этот код, остановите свою программу, добавьте в нее выделенный код и нажмите клавишу F5 для выполнения программы. Нажмите на кнопку, и надпись на ней изменится. Рабочую программу — пример события нажатия на кнопку — можно найти в папке примеров, прилагаемых к курсу (Проект Example5).

Примечание редактора. Программа работает, но можно немного улучшить ее внешний вид. Сейчас положение кнопки на форме и ее размеры установлены по умолчанию. Для изменения этих параметров добавьте в конструктор класса необходимые строчки, и снова запустите программу.

```
public MyButtonClass()
    \{
        mrButton = new Button();
```

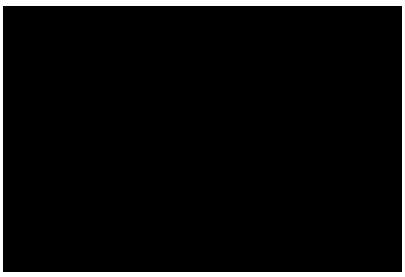
```
        mrButton.Text = "Нажми меня";
        mrButton.Top = 100;
        mrButton.Left = 100;
        mrButton.Height = 50;
        mrButton.Width = 70;

        mrButton.Click += new System.EventHandler(MyButtonClickEventHandler);
        this.Controls.Add(mrButton);
    \}
```

Теперь мы попробуем описать основную структуру метода обработчика событий мыши. Вероятно, в этом случае вы *захотите* использовать информацию, подставляемую в параметр MouseEventArgs, хотя бы для того, чтобы выяснить, какая из кнопок мыши нажимается.

```
public void TheMouseIsDown(object sender, MouseEventArgs e)
\{
    if (e.Button == MouseButtons.Left)
        this.Text = "Нажата левая кнопка мыши";
\}
```

Далее показано, как связать событие с методом. В переводе с компьютерного в тексте написано следующее: "Если при выполнении этой программы нажимается кнопка мыши, надо перейти к методу TheMouseIsDown которому известно, как следует обрабатывать события мыши:

```
this.MouseDown += new MouseEventHandler(TheMouseIsDown);
```



Можно внести некоторые улучшения, чтобы при запуске следующего обработчика код, содержащийся в нем, делал окно более широким или узким в зависимости от нажимаемой кнопки.

```
public void TheMouseWasClicked(object sender, MouseEventArgs e)
\{
    // При нажатии левой кнопки
    if (e.Button == MouseButtons.Left)
        // Расширение текущего окна
        this.Width = this.Width + 100;
    else if (e.Button == MouseButtons.Right)
        // Сужение текущего окна
        this.Width = this.Width - 100;
\}
```

Другой обработчик событий позволяет обнаружить перемещение мыши и рисовать окружность в том месте, где находится курсор:

```
public void TheMouseMoved(object sender, MouseEventArgs e)
\{
    // Подготовка области рисования
    System.Drawing.Graphics g = this.CreateGraphics();

    // Использование красной ручки
    System.Drawing.Pen redPen = new System.Drawing.Pen(System.Drawing.Color.Red,
3);

    // Рисуем окружность как эллипс с равными осями.
    // Окружность рисуется в охватывающем ее квадрате.
    // Координаты X и Y левого верхнего угла квадрата
    // определяются координатами текущего положения мыши.
    g.DrawEllipse(redPen, e.X, e.Y, 40, 40);

    // Очистка
    g.Dispose();
\}
```

На снимке экрана показано, как это выглядит при перемещении мыши: