Классы в C#, методы классов

Цель работы: Изучить описание методов классов, функций-членов в C#, их создание в C# и некоторые алгоритмы их обработки.

Теоретические сведения

Пример 1.1 . Простой класс Time

```
using System;
public class Time
{// открытые методы
public void DisplayCurrentTime()
{ Console.WriteLine(“ заглушка для DisplayCurrentTime”)
}
// закрытые переменные
lnt Year;
int Month;
int Date;
int Hour;
int Minute;
int Second;
}
public class Tester
{
static void Main()
{
Time t = new Time();
DisplayCurrentTime();
}
}
```

Единственный метод, объявленный в определении класса Time, – метод DisplayCurrentTime();. Тело этого метода определено внутри определения класса. C# не требует объявления методов до их описания. В C# все методы встраиваются в определение класса, как метод DisplayCurrentTime();. В определении метода DisplayCurrentTime(); указано, что тип возвращаемого значения – void, то есть он не возвращает значения методу, вызвавшему его. В конце определения класса Time объявляется ряд переменных: lnt Year,int Month, int Date, int Hour, int Minute, int Second.

После закрывающей фигурной скобки определен другой класс, Tester. Он включает в себя метод Main(). В методе Main()создается экземпляр класса Time, и его адрес присваивается объекту t. Поскольку t является экземпляром Time, метод Main() может вызывать метод DisplayCurrentTime(), предоставляемый объектами этого типа, и тем самым выводить на экран текущее время:

Желательно определять переменные класса как закрытые (private). Это гарантирует, что только методы того же класса будут иметь доступ к их значениям. Поскольку уровень доступа private устанавливается по умолчанию, нет необходимости указывать его явно.

Так, в примере 1.1 объявления переменных лучше переписать следующим образом:

```
// закрытые переменные
private int Year;
private int Month:
private ini Date;
onvate int Hour;
private int Minute;
private int Second;
```

Класс Time и метод DisplayCurrentTime() объявлены открытыми, чтобы любой другой класс мог обратиться к ним.

Аргументы метода

Методы могут принимать любое количество параметров. Список параметров указывается в круглых скобках после имени метода, причем каждый параметр предваряется своим типом. Например, в следующем участке кода определяется метод по имени MyMethod, который возвращает void (то есть не возвращает значения). Он имеет два параметра с типами int и button:

```
void MyMethod (int firstParam, button secondParam)
{
// ..
}
```

В теле метода параметры ведут себя как локальные переменные, как будто бы они были объявлены и инициализированы значениями, переданными методу. В примере 1.2 иллюстрируется передача значений методу. В этом случае значения имеют типы int и float.

Пример 1.2. Передача значений методу SomeMethod()

```
using System;
public class MyClass
{
public void SomeMethod(int firstParam, float secondParam)
{ Console.WriteLine( “Получены параметры {0} {1} ”, firstParam, secondParam);
}
} public class Tester
{
static void Main()
{
int howManyPeople = 5;
float pi = 3,14f:
MyClass me = new MyClass();
me.SomeMethod(howManyPeople, pi);
}
}
```

Здесь метод SomeMethod() принимает значения типов int и f loat и выводит их с помощью метода Console.WriteLine(). Параметры, названные firstParam и secondParam, в теле метода SomeMethod() трактуются как локальные переменные.

Создание объектов

Базовые типы C# (int, char и т. д.) являются размерными типами; они хранятся в стеке. Объекты, со своей стороны, имеют ссылочный тип и хранятся в куче, а создаются с помощью ключевого слова new; Здесь переменная t, на самом деле, не содержит значение объекта Time. Вместо этого она содержит адрес объекта (безымянного), созданного в куче. Сама же переменная t является лишь ссылкой на этот объект.

Конструкторы

Обратите внимание на оператор, создающий объект Time в примере 1.1. Внешне он выглядит как вызов метода: И действительно, при создании экземпляра происходит обращение к методу. Он называется *конструктором,* и программист должен либо определить его как часть определения класса, либо положиться в этом на среду CLR, которая сама его предоставит. Задача конструктора – создать объект указанного класса и перевести его в *действующее* состояние. До вызова конструктора объект представляет собой неинициализированную область памяти; по окончании его работы эта память содержит действующий экземпляр данного класса.

Класс Time из примера 1.1 не определяет никакого конструктора. Если конструктор не объявлен, компилятор предоставляет его самостоятельно. Конструктор, вызванный по умолчанию, создает объект, но не предпринимает никаких других действий. Переменные класса инициализируются безвредными значениями (целые – нулем, строки – пустой строкой и т. д.).

Как правило, программисты предпочитают определять собственные конструкторы и снабжать их аргументами, чтобы конструктор мог устанавливать начальное состояние объекта. В примере 1.1 было бы разумно передавать конструктору информацию о текущем годе, месяце, дне месяца и т. д., чтобы объект был заполнен осмысленными данными. При определении конструктора объявляется метод с тем же именем, что и класс, в котором он определяется. У конструкторов нет возвращаемого типа, и они обычно объявляются открытыми. Если планируется передавать конструктору какие-либо аргументы, их список указывается, как для любого другого метода. В примере 1.3 объявляется конструктор для класса Time, который принимает единственный аргумент, объект типа DateTime.

Пример 1.3. Объявление конструктора

```
public class Jim
{   // открытые методы доступа
public void DisplayCurrentTime()
{ System.Console.WriteLine( “{0},{1},{2},{3},{4},{5}”, Month, Date, Year, Hour,
    Minute, Second );
}
// конструктор
public Time(System,DateTime  dt)
{
Year = dt.Year;
Month = dt.Month;
Date = dt.Day;
Hour = dt.Hour;
```

```
Minute = dt.Minute;
Second = dt.Second;
}
// закрытые переменные класса
int Year;
int Month;
int Date:
int Hour;
int Minute;
int Second;
}
public class Tester
{
static void Main()
{
System. DateTime currentTime = System. DateTime. Now;
Time t =new Time(currentTime);
t.DisplayCurrentTime ( ) ;
}
}
```

В этом примере конструктор принимает объект DateTime и инициализирует все переменные класса, беря за основу значения из этого объекта.

После того как конструктор закончит свою работу, в памяти компьютера будет находиться объект Time с инициализированными значениями. Когда метод Main() вызывает метод DisplayCurrentTime(), эти значения выводятся на экран. Закомментируйте какое-нибудь присваивание и снова выполните программу. Окажется, что компилятор инициализирует соответствующую переменную нулем. Целые переменные класса устанавливаются в 0, если не указано другое значение. Помните, что переменные, имеющие размерный тип (например, целочисленные), не могут оставаться *неинициализированными;* если конструктор не получит конкретных указаний, он установит значения по умолчанию.

В примере 1.3 в методе Main() класса Tester создается объект DateTime. Этот объект, поставляемый библиотекой System, предлагает ряд открытых значений: Year, Montn, Day, Hour, Minute и Second, которые в точности соответствуют переменным объекта Time. Кроме того, объект DateTime предоставляет статический метод Now(), который возвращает ссылку на экземпляр объекта DateTime, инициализированный текущим временем.

Рассмотрим выделенную строчку в методе Main(), где объект DateTime создается вызовом статического метода Now( ). Он создает DateTime в куче и возвращает ссылку на него.

Возвращенная ссылка присваивается переменной currentTime, которая объявлена как ссылка на объект DateTime. Затем currentTime передается в качестве параметра конструктору Time. Параметр этого конструктора, dt, тоже представляет собой ссылку на объект DateTime. Теперь ct ссылается на тот же объект, что и переменная currentTime. Таким образом, конструктор Time получает доступ к открытым переменным объекта DateTime, созданного в методе Tester. Main(). Когда объект передается в качестве параметра, он передается *по ссылке,* то есть передается указатель на объект, а копия объекта не создается.

Инициализаторы

Вместо того чтобы инициализировать значения переменных класса в каждом конструкторе, можно сделать это в *инициализаторе.* Инициализатор создается присваиванием начального значения элементу класса:

private int Second = 30; // инициализатор

Предположим, семантика объекта Time такова, что независимо от установленного времени секунды всегда инициализируются значением 30.

Класс Time можно переписать с применением инициализатора. Тогда, какой бы конструктор ни был вызван, переменная Second будет всегда получать начальное значение либо явно, от конструктора, либо неявно, от инициализатора. Это продемонстрировано в примере 1.4.

Пример 1.4. Использование инициализатора

```
puiblic class Time
{
// открытые методы доступа
public void DisplayCurrentTime()
{
System. DateTime now = System.DateTime.Now;
System. Console . WriteLi,ne("\nОтладка\t: {0}/{1}/{2} {3} : {4} : {5}»,now.
      Month, now. Day, now.Year, now.Hour, now. Minute, now. Second);
}
//конструкторы
public Time(System,DateTime dt)
{
Year = dt.Year;
Month = dt.Month;
Date = dt.Day;
Hour = dt.Hour;
Minute = dt.Minute;
Second = dt.Second;//явное присваивание
}
public Time(int Year, int Month, int Date, int Hour, int Minute)
{    this.Year = Year;
this.Month =Month;
this.Date = Day;
this.Hour = Hour;
this.Minute = Minute;
}
// закрьгые переменные класса
private int Year;
private int Month;
private int Date;
private int Hour;
orivate int Mirute;
private int Second = 30; // инициализатор
}
public class Tester
{
static void Main()
{
```

```
System. DateTime currentTime = System. DateTime. Now;
Time t =new Time(currentTime);
t.DisplayCurrentTime ( ) ;
Time t2 =new Time(2007,02,20,17,24);
T2.DisplayCurrentTime ( ) ;
}
}
```

Если инициализатор не указан, конструктор присвоит каждой переменной нулевое начальное значение. Однако в приведенном примера переменная Second получает значение 30:

```
private int Second = 30; // инициализатор
```

Если для переменной Second не будет передано значение, то при создании она будет проинициализирована числом 30:

```
Time t2 =new Time(2007,02,20,17,24);
T2.DisplayCurrentTime ( ) ;
```

Однако если переменной Second значение присвоено, как это делается в конструкторе, принимающем объект DateTime, новое значение замещает первоначальное.

При первом выполнении программы вызывается конструктор, принимающий объект DateTime, причем секунды устанавливаются в значение 24. Если бы у программы не было инициализатора и переменной Second не присваивалось никакого значения, то компилятор установил бы ее в ноль.

Копирующие конструкторы

Копирующий конструктор (copy constructor) создает новый объект, копируя переменные из существующего объекта того же типа. Пусть, например, требуется передать объект Time конструктору Time() так, чтобы новый объект Time содержал те же значения, что и старый. Язык C# не добавляет в класс копирующий конструктор, так что программист должен написать такой конструктор самостоятельно. Подобный конструктор всего лишь копирует элементы исходного объекта во вновь создаваемый:

```
public Time(Time existingTimeObject)
{
Year = existingTimeObject.Year;
Montn = existingTimeObject.Month;
Date = existingTimeObject.Date;
HOLT = existingTimeObject.HOLT;
Minute = existingTimeObject.Minute;
Second = existinqTimeObject.Second;
}
```

Копирующий конструктор вызывается путем создания объекта типа Time и передачи ему имени копируемого объекта Time:

```
Time t3= new Time (t2);
```

Здесь переменная t2 передается в качестве аргумента existingTimeQbject копирующему конструктору, который создаст новый объект Time.

Ключевое слово this

Ключевое слово this является ссылкой на текущий экземпляр объекта. Ссылка This (иногда ее называют указателем) является скрытым указателем на каждый нестатический метод класса. Любой метод может использовать ключевое слово this для доступа к другим нестатическим методам и переменным этого объекта. Существует три типичных случая применения ссылки this. Первый доступ к члену объекта, скрытому параметром. Вторым применением ссылки this является передача ссылки на текущий объект другому методу в качестве аргумента. Например, в следующем коде:

```
public void FirstMetnod(OtherClass otherQbiect)
{
otherQbject.SecondMethod(this);
}
```

используются два класса: один с методом First Method (), а другой, класс OtherClass, с методом SecondMethod(). В теле FirstMethod() вызывается метод SecondMethod(), которому передается текущий объект для дальнейшей обработки. Третье применение ссылки this связано с индексаторами.

Указатель – это переменная, в которой хранится адрес объекта в памяти. C# не использует указатели на управляемые **объекты** (объекты, определенные в управляемом коде).

Статические члены класса

Свойства и методы класса могут быть либо членами *экземпляра (instance members),* либо*статическими членами (static members).* Первые связаны с экземплярами класса, а вторые рассматриваются как часть самого класса. Программист обращается к статическому члену, указывая имя класса, в котором этот член был объявлен. Пусть, например, имеется класс с именем Button. Объекты этого класса названы btrUpdate и btnDelete. Предположим далее, что класс Button имеет статический метод SomeMetnodf). В C# недопустимо обращаться к статическому методу или переменной через экземпляр, и подобные попытки будут пресечены компилятором (программисты, привыкшие к C++, возьмите на заметку!).

В C# нет глобальных методов; существуют только методы класса. Однако аналогичный результат достигается путем определения статических методов в рамках класса. Статические методы подобны глобальным в том смысле, что программист может вызвать их, не имея под рукой конкретный объект. Однако статические методы имеют перед глобальными то преимущество, что область видимости их имени ограничена классом, и глобальное пространство имен не засоряется многочисленными функциями.

Вызов статических методов

Метод Main() является статическим. Про статические методы говорят, что они действуют в классе, а не в экземпляре класса. Они не имеют ссылки this, поскольку отсутствует экземпляр, на который она могла бы указывать.

Статические методы не имеют непосредственного доступа к нестатическим членам. Чтобы метод Main() мог вызвать нестатический метод он должен создать объект. Вспомним пример 1.2,

Пример 1.2. Передача значений методу SomeMethod()

```
using System;
public class MyClass
{
public void SomeMethod(int firstParam, float secondParam)
{ Console.WriteLine( “Получены параметры {0} {1} ”, firstParam, secondParam);
}
}          public class Tester
{
static void Main()
{
int howManyPeople = 5;
float pi = 3,14f;
MyClass me = new MyClass();
me.SomeMethod(howManyPeople, pi);
}
}
```

SomeMetnod() представляет собой нестатический метод класса MyClass. Чтобы обратиться к этому методу, метод Main() должен сначала создать объект типа MyClass, а затем вызвать метод данного объекта.

Применение статических конструкторов

Если в классе объявлен статический конструктор, можно гарантировать, что этот конструктор сработает до создания какого бы то ни было экземпляра этого класса. Программист не в состоянии управлять моментом выполнения статического конструктора, однако можно быть уверенным, что такой конструктор будет выполнен после запуска программы, но до создания первого экземпляра класса. По этой причине нельзя предполагать (и выяснить), создается ли экземпляр в данный момент или нет.

Например, в класс Time можно добавить следующий конструктор:

```
Static Time()
{  Name=” время”;
}
```

Обратите внимание на отсутствие модификатора права доступа (например, public) перед статическим конструктором. Модификаторы права доступа в этом случае не разрешены. Кроме того, поскольку данный метод статический, ему недоступны нестатические переменные класса, и переменная Name должна быть объявлена как статическая переменная класса.

```
Private static string Name;
```

Заключительным штрихом будет добавление строчки в метод DisplayCurrentTime():

```
Public void  DisplayCurrentTime()
{
System. Console.WriteLine( «Имя: {0}”, Name);
System. Console.WriteLine( «{0}/{1}/{2} {3}:{4}:{5}”,  Month,  Date, Year, Hour,
    Minute, Second);      }
```

Программа работает, но для достижения той же цели вовсе не обязательно создавать статический конструктор. Вместо этого можно было бы применить инициализатор:

```
Private static string Name=”время”;
```

делающий практически то же самое. Статические конструкторы подходят в основном для настройки, которую нельзя выполнить с помощью инициализатора, и которая производится только один раз.

Применение закрытых конструкторов

В языке C# нет глобальных методов и констант. Некоторые программисты пишут маленькие вспомогательные классы, единственным предназначением которых является хранение статических элементов.

Использование статических полей

В основном статические переменные класса применяются для отслеживания количества экземпляров класса, существующих в данный момент. Это иллюстрируется примером 1.5.

Пример 1.5. Использование статических полей для подсчета экземпляров

```
using System;
public class Cat
{
public Cat()
{
instances++;
}
public static void HowManyCats()
{
Console. WriteLine(«Кошек  принято: {0} «, instances);
}
private static int instances = 0;
}
public class Tester
{
static void Main()
{
Cat. HowManyCats();
Cat .frisky = new Cat();
Cat.whiskers = new Cat();
Cat. HowManyCats();
}
}
```

Класс Cat содержит самый минимум. В нем создается и инициализируется (нулем) статическая переменная instances. Статический элемент считается частью класса, а не

элементом экземпляра. Поэтому он не может быть инициализирован компилятором в момент создания экземпляра. Явный инициализатор совершенно необходим для статических переменных класса. По мере создания (конструктором) дополнительных экземпляров класса Cat счетчик будет увеличиваться.

Уничтожение объектов

В C# предусмотрена сборка мусора, и поэтому отсутствует необходимость в явном деструкторе. Однако если объект работает с неуправляемыми ресурсами, программисту приходится явно освобождать их по завершении работы. Это делается с помощью *деструктора (destructor),* вызываемого сборщиком мусора при уничтожении объекта.

Деструктор должен лишь освобождать ресурсы, удерживаемые объектом, и не должен ссылаться на другие объекты. Следует заметить, что если используются только управляемые ресурсы, то деструктор не нужен и не должен реализовываться. Поскольку применение деструктора сопряжено с некоторыми затратами, его необходимо применять лишь к нуждающимся в нем методам (а именно к методам, потребляющим ценные неуправляемые ресурсы).

Статические методы доступа к статическим полям

Весьма нежелательно объявлять данные класса открытыми. Это относится и к статическим переменным класса. Тогда как решение об объявлении статических элементов закрытыми, как это сделано в приведенном выше примере (переменная instances), вполне приемлемо. Был создан открытый метод HowManyCats(), обеспечивающий доступ к закрытому члену класса. Поскольку метод HowManyCats() сам является статическим, у него нет проблем с обращением к статической переменной instances, Никогда непосредственно не вызывайте деструктор объекта. Сборщик мусора сделает это сам.

Как работают деструкторы

Сборщик мусора ведет список объектов, имеющих деструкторы. Этот список обновляется каждый раз, когда появляется или уничтожается подобный объект. Когда объект из такого списка попадается сборщику мусора, он помещается в очередь объектов, ожидающих уничтожения. После выполнения деструктора сборщик мусора уничтожает объект, обновляет очередь и список объектов с деструкторами.

Деструктор C#

Синтаксически деструктор в языке C# напоминает деструктор C++, однако работает он совершенно по-другому. Объявляется деструктор с помощью символа «тильда (~)»

В C# этот синтаксис служит лишь сокращенной записью объявления метода Finalize(), связывающей его с базовым классом. Таким образом, запись: преобразуется компилятором C# в следующую:

```
protected override void Finalize()
{
try
{
```

```
// выполнить действия
}
finally
{
base.Firalize();
}}
```

Передача параметров

По умолчанию типы значений передаются методам по значению, когда объект передается методу, внутри метода создается временная копия объекта. По окончании работы метода копия уничтожается. Хотя передача по значению вполне приемлема, бывают ситуации, когда лучше передавать объекты по ссылке. Язык C# предоставляет модификатор параметра ref для передачи переменных методу по ссылке. Кроме того, существует модификатор OJT для тех случаев, когда переменная с модификатором ref должна передаваться без предварительной инициализации. C# поддерживает также модификатор params, позволяющий методу принимать переменное количество параметров.

Передача по ссылке

Методы могут возвращать лишь одно значение (хотя оно может быть коллекцией из нескольких значений). Вернемся к классу Time и добавим метод GetTime (), возвращающий часы, минуты и секунды. Поскольку три значения возвратить невозможно, поступим следующим образом. Передадим методу три аргумента, позволим ему их изменить, а затем проанализируем результат в вызывающем методе. В примере 1.7 демонстрируется первая версия метода.

Пример 1.7. Возвращение результата через параметры

```
public class Time
{
// открытые метода
public void DisplayCurrentTime()
{
System. Console, WriteLine(“{0}/{1}/{2} {3}: { 4 } : {5}»
Month, Date,Year, Hour, Minute, Second);
}
public int GetHour()
{ return Hour;
}
public void GetTime(int h, int m, int s)
{
h = Hour;
m = Minute;
s = Second;
}
//конструктор
public Time( System.DateTime dt)
{ Year =dt.Year;
Month=dt. Month;
Date=dt. Date;
```

```
Hour=dt. Hour;
Minute=dt. Minute;
Second=dt. Second;
}
//закрытые переменные класса
private int Year;
private ini Month;
private int Date;
orivate int Hour;
Drivate irt Minute;
private int Second;
}
public class Tester
{ static void Main()
{
System. DateTime currentTime = System. DateTime. Now;
Time t =new Time(currentTime);
t.DisplayCurrentTime ( ) ;
int theHour = 0;
int tneMinute = 0;
int theSecond = 0;
t.GetTime(theHour, tneMinute. theSecond);
System.Console. WriteLine(«Текущее время: {0}: {1}; {2}",theHour. theMinute,
  theSecond);
}
}
```

Обратите внимание, что в строке «Текущее время» выводится 0:0:0. Очевидно, первая версия работает неправильно. Проблема в передаче аргументов. Методу GetTime() передаются три аргумента, он изменяет их значения, но при обращении к переданным переменным в методе Main() они оказываются неизмененными. Это происходит потому, что int является размерным типом, следовательно, параметры передаются по значению. Иными словами, в методе GetTime() создаются их копии. Отсюда вывод – следует передавать аргументы по ссылке.

Необходимо внести в код два небольших изменения. Во-первых, параметры в методе GetTime() должны быть помечены модификатором ref как ссылочные:

```
public void GetTime(ref int h, ref int m, ref int s)
{ h=Hour;
m= Minute;
s=Second;
}
```

Во-вторых, в вызове метода GetTime() также надо указать, что аргументы передаются по ссылке: t.GetTime(ref Hour, ref Minute, ref Second);

Если не сделать второе изменение и не пометить параметры ключевым словом ref , то компилятор выдаст сообщение о невозможности преобразовать тип int в ref int.

Теперь выводятся корректные результаты. Объявив, что параметры ссылочные, программист инструктирует компилятор передавать их по ссылке. Никакие копии не создаются, а параметр в GetTime() является ссылкой на соответствующую переменную (например,

theHour), созданную в методе Main( )-Изменение значений в методе GetTime() отражается на переменных метода Main( ).

В языке C# действует принцип *явной инициализации,* согласно которому все переменные должны быть инициализированы до первого использования в программе. Если в примере 1.7 не инициализировать переменные TheHour, theMinute и theSecond до передачи их в качестве аргументов методу GetTime(), то компилятор выдаст сообщения об ошибках. В рассматриваемом примере этим переменным присваиваются нулевые значения:

```
int theHour = 0;
int tneMinute = 0;
int theSecond = 0;
t.GetTime( ref Hour,  ref Minute,  ref Second);
```

Такая инициализация выглядит глупо, поскольку переменные тут же передаются методу GetTime(), где они будут изменены. Если же ее не выполнить, компилятор выдаст сообщения.Специально для этого случая в языке C# предусмотрен модификатор параметра out. Он снимает требование на обязательную инициализацию аргумента, передаваемого по ссылке. Аргументы метода GetTime(); не передают ему никакой информации, напротив, они являются средством передачи информации из него. Поэтому, пометив их модификатором out, программист избавляет себя от необходимости инициализировать соответствующие переменные вне метода. Внутри же метода эти формальные параметры должны получить какие-либо значения до

окончания работы метода. Ниже приводится модифицированное объявление параметров метода GetTime( ):

```
public void GetTime(out int h,  out int m, out int s)
{ h=Hour;
m= Minute;
s=Second;
}
```

А вот новый вызов этого метода из Main ( ) :

```
t.GetTime(out Hour,  out Minute,  out Second);
```

Вывод: размерные типы передаются методам по значению. Аргументы с модификатором ref служат для передачи размерных типов по ссылке. Это позволяет возвращать их измененные значения в вызывающий метод. Аргументы с модификатором out применяются только для передачи информации из вызываемого метода. В примере 1.8 приводится программа из примера 1.7, переписанная так, что в ней задействованы все три вида параметров.

Пример 1.8. Входные, выходные и ссылочные параметры

```
public class Time
{
// открытые методы доступа
public void DisplayCurrentTime()
{
```

```
System. Console, WriteLine("{0}/{1}/{2} {3}: {4}: {5}», Month, Date,Year, Hour,
    Minute, Second);
}
public int GetHour()
{  return Hour;
}
public void SetTime(int hr,  out int min  , ref  int sec)
{
// если переданное время >= 30,
// инкрементировать минуты и обнулить секунды,
// в противном случае оставить все как есть
if (sec >= 30)
{
Minute++;
Second = 0;
}
Hour = hr; // присвоить переданное значение
// возвратить минуты и секунды
min = Minute;
sec = Second;
}
//конструктор
public Time( System.DateTime dt)
{      Year =dt.Year;
Month=dt. Month;
Date=dt. Date;
Hour=dt. Hour;
Minute=dt. Minute;
Second=dt. Second;
}
// закрытые переменные класса
private int Year;
private int Month;
private int Date;
private int Hour;
private int Minute;
private int Second;
}
public class Tester
{ static void Main()
{
System. DateTime currentTime = System. DateTime. Now;
Time t =new Time(currentTime);
t.DisplayCurrentTime ( ) ;
int theHour = 3;
int theMinute;
int theSecond = 20:
t.SetTime(tneHour. out theMinute, ref theSecond);
System. Console. Writeline(«Сейчас: {0} минут и {1} секунд», theMinute,
    theSecond);
theSecond = 40;
t.SetTime(theHour, out theMinute, ref theSecond);
System,Console.WriteLine(«Сейчас+ «{0} минут и {1} секунд»,theMinute,
    theSecond);
}
}
```

Метод SetTime () слегка надуманный, но зато иллюстрирует все три вида параметров. Аргумент tneHour передается по значению. Его единственная задача – установить переменную Hour в нужное значение, и никакую выходную информацию он в себе не несет. Передаваемый по ссылке аргумент theSecond используется для установки значения внутри метода. Если этот параметр больше или равен 30, то переменная Second сбрасывается в ноль, а переменная Minute увеличивается на 1.

Наконец, аргумент theMinute передается методу для того, чтобы получить в нем значение переменной Minute и возвратить его. Поэтому он помечен модификатором out.

Перегрузка методов и конструкторов

Нередко требуется, чтобы одно и то же имя было сразу у нескольких функций. Самым распространенным случаем является наличие нескольких конструкторов. В примерах, приводимых до сих пор, конструктор принимал один параметр, объект DateTime. Было бы удобно устанавливать в новых объектах Time произвольное время, передавая им значения года, месяца, числа, часов, минут и секунд. Было бы еще удобнее, если бы одни разработчики могли пользоваться одним конструктором, а другие – другим. Такое понятие, как перегрузка функций, существует именно для этих ситуаций. *Сигнатура (signature)* метода определяется его именем и списком па раметров. Два метода отличаются по сигнатурам, если у них разные имена или списки параметров. Списки параметров могут отличаться по количеству или типам параметров.

Например, в следующем фрагменте программы первый метод отличается от второго количеством параметров, а второй от третьего – их типами.

```
Void mes( int a);
Void mes( int a , int b);
Void mes( int a , srtring b);
```

Класс может иметь любое количество методов, если их сигнатуры отличаются друг от друга. В примере 1.9 демонстрируется класс Time с двумя конструкторами, один из которых принимает объект DateTime, а второй – шесть целых чисел.

Пример 1.9, Перегрузка конструктора

```
public class Time
{
// открытые методы доступа
public void DisplayCurrentTime()
{
System. Console, WriteLine("{0}/{1}/{2} {3}: { 4 } : {5}», Month, Date, Year,
      Hour, Minute, Second);
}
//конструкторы
public Time( System.DateTime dt)
{ Year =dt.Year;
Month=dt. Month;
Date=dt. Date;
Hour=dt. Hour;
```

```
Minute=dt. Minute;
Second=dt. Second;
}
public Time(  int Year, int Month, int .Date, int Hour, int Minute,  int Second)
{  this.Year =Year;
this.Month=Month;
this.Date=Date;
this.Hour=Hour;
this.Minute=Minute;
this.Second=Second;
}
// закрытые переменные класса
private int Year;
private int Month;
private int Date;
private int Hour;
private int Minute;
private int Second;
}
public class Tester
{ static void Main()
{
System. DateTime currentTime = System. DateTime. Now;
Time t =new Time(currentTime);
t.DisplayCurrentTime ( ) ;
Time t2 = new Time(2007,11,18,11,03,30);
t2.DisplayCurrentTime();
}
}
```

Как видно из примера, у класса Time два конструктора. Если бы сигнатура состояла только из имени функции, компилятор не знал бы, какой конструктор вызывать при создании объектов t1 и t2. Однако поскольку сигнатура включает в себя и типы аргументов, компилятор в состоянии сопоставить вызов конструктора для t1 с описанием конструктора, требующего объект DateTime. Аналогичным образом компилятор способен связать вызов конструктора объекта t2 с конструктором, сигнатура которого содержит список из шести аргументов с типом int.

Перегружая метод, программист должен изменить сигнатуру (то есть имя метода, количество параметров или их тип). Можно, к тому же, изменить и тип возвращаемого значения, но это не обязательно. Изменение возвращаемого типа не ведет к перегрузке метода, а наличие двух методов с одинаковыми сигнатурами, но разными возвращаемыми типами вызовет ошибку на этапе компиляции.

Пример 1.10. Изменение возвращаемого типа перегруженных методов

```
public class Tester
{
private int Triple(int val)
{ return 3*val;
}
```

```
private long Triple(long val)
{ return 3* val;
}
public void Test()
{
int x = 5;
int y = Triple(x);
System. Console. WnteLine(“x: {0} y: {1}», x, y);
Long lx=10;
Long ly= Triple(lx);
System. Console. WnteLine(“x: {0} y: {1}», lx, l y);
}
static void Main()
{ Tester t = new Tester();
t.Test();
}
}
```

В этом примере класс Tester перегружает метод Triple (), один раз с целочисленным параметром, а другой – с параметром типа long. Возвращаемые типы у методов Triple () разные. Хотя это не обязательно, я данном случае разница возвращаемых типов очень удобна.

Инкапсуляция данных в свойствах

Свойства обеспечивают пользователю возможность читать состояние класса, как если бы он обращался непосредственно к полям класса, причем фактически этот доступ реализуется с помощью метода класса. Пользователь хочет обладать правом прямого доступа к состоянию объекта, не желая иметь дело с методами. Разработчик класса, со своей стороны, стремится скрыть внутреннее состояние класса в членах класса и предоставляет клиенту косвенный доступ, через методы.

Заставляя клиента применять метод (или свойство), программист может вносить изменения во внутреннюю структуру класса Time, не приводя при этом к неработоспособности использовавших его программ. Свойства удовлетворяют обоим требованиям. Они предоставляют пользователю простой интерфейс, поскольку выглядят как переменные класса. С другой стороны, они реализованы в виде методов и скрывают данные в соответствии с требованиями объектно-ориентированного программирования.

Пример 1.11. Использование свойства

```
public class Time
{
// открытые методы доступа
public void DisplayCurrentTime()
{
System. Console, WriteLine(“{0}/{1}/{2} {3}: { 4 } : {5}»,Month, Date, Year,
      Hour, Minute, Second);
}
//конструктор
public Time( System.DateTime dt)
{ Year =dt.Year;
Month=dt. Month;
Date=dt. Date;
```

```
Hour=dt. Hour;
Minute=dt. Minute;
Second=dt. Second;
}
public Time(  int Year, int Month, int .Date, int Hour, int Minute,  int Second)
{  this.Year =Year;
this.Month=Month;
this.Date=Date;
this.Hour=Hour;
this.Minute=Minute;
this.Second=Second;
}
// создание свойства
public int Hour
{
get
{
return hour;
}
set
{
hour = value;
}
}
// закрытые переменные класса
private int Year;
private int Month;
private int Date;
private int Hour;
private int Minute;
private int Second;
}
public class Tester
{
static void Main()
{
System. DateTime currentTime = System. DateTime. Now;
Time t =new Time(currentTime);
t.DisplayCurrentTime ( ) ;
int theHour = t. Hour;
System. Console. Writeline(«\nПолученное значение nour: {0}\n", theHour);
theHour++:
Hour =theHour;
System. Console. WriteLine(«измененное  значение hour: (0}\n», theHour);
}
}
```

Чтобы объявить свойство, напишите его тип и имя и поставьте пару фигурных скобок. Внутри скобок можно объявить *процедуры доступа (accessor) к* свойству, реализующие чтение и изменение его состояния. У них не должно быть явных параметров, хотя, как будет показано ниже, процедура доступа set() имеет неявный параметр value.

В примере 1.11 приводится свойство Hour. В его объявлении указаны две процедуры доступа: get и set.

```
public int Hour
{
get
{
return hour;
}
set
{
hour = value;
}
}
```

Каждая из этих процедур доступа имеет тело, в котором и выполняется вся работа по чтению или изменению значения свойства.

Поля, предназначенные только для чтения

Предположим, для каких-то целей понадобилась версия класса Time, предоставляющая открытые статические значения текущего времени и даты. В примере 1.12 дано простое решение этой задачи.

Пример 1.12. Использование статических открытых констант

```
public class RightNow
{
{
System. DateTime dt = System. DateTime. Now;
Year = dt.Year:
Month = dt. Month:
Date = dt.Day;
Hour = dt. Hour;
Minute=dt. Minute;
Second=dt. Second;
}
// открытые переменные класса
public static irt Year:
public static int Moith;
public siatic int Date;
public static int Hour;
public static int Minute;
puolic static int Second;
}
public class Tester
{
static void Main()
{
System. Console. Write Line («Год: {O}»,RightNow.Year.ToStnng());
RightNow.Year = 2016;
System. Console. Write Line («Год: {O}»,RightNow.Year.ToString());
}
}
```

Все будет работать хорошо, пока кто-нибудь не придет и не изменит одно из значений. Как видно из примера, значение RightNow.Year можно изменить, например, на 2003. Очевидно, это не то, что требовалось. Чтобы добиться желаемого результата, хотелось бы пометить

статические значения как постоянные, но это невозможно, потому что они не инициализируются до выполнения статического конструктора. Именно для такого случая, в языке C# предусмотрено ключевое слово readonly. Если изменить описания переменных класса на следующие:

```
public static readonly int Year;
public static readonly int Month;
public static readonly int Date;
public static readonly int Hour;
public static readonly int Minute;
public static readonly int Second;
```

а затем закомментировать повторное присваивание в методе Main ():

```
// RightNow.Year = 2002; // ошибка!
```

то программа будет скомпилирована без ошибок и будет работать, как задумано.