Объектно-ориентированное программирование

Лекция №5. Структуры, Классы

# Темы:

1. Классы. Методы, Поля, Свойства. Перегружаемые методы.

2. Наследование

3. Виртуальные методы, полиморфизм

# Классы

Класс является типом данных, определяемым пользователем. Он должен представлять собой одну логическую сущность, например, являться моделью реального объекта или процесса. Элементами класса являются данные и функции, предназначенные для их обработки.

Описание класса содержит ключевое слово с 1 ass, за которым следует его имя, а далее в фигурных скобках — тело класса, то есть список его элементов. Кроме того, для класса можно задать его базовые классы (предки) и ряд необязательных атрибутов и спецификаторов, определяющих различные характеристики класса:

[ атрибуты ] [ спецификаторы ] class имя\_класса [ : предки]

телокласса

Как видите, обязательными являются Только ключевое слово class, а также имя и тело класса. Имя класса задается программистом по общим правилам С#. Тело класса — это список описаний его элементов, заключенный в фигурные скобки. Список может быть пустым, если класс не содержит ни одного элемента. Таким образом, простейшее описание класса может выглядеть так:

class Demo {}

Спецификаторы определяют свойства класса, а также доступность класса для других элементов программы. Возможные значения спецификаторов перечислены в табл. 5.1. Класс можно описывать непосредственно внутри пространства имен или внутри другого класса. В последнем случае класс называется вложенным. В зависимости от места описания класса некоторые из этих спецификаторов могут быть запрещены.

Класс является обобщенным понятием, определяющим характеристики и поведение некоторого множества конкретных объектов этого класса, называемых экземплярами, или объектами, класса.

Объекты создаются явным или неявным образом, то есть либо программистом, либо системой. Программист создает экземпляр класса с помощью операции new, например:

Demo а = new Demo(); // создание экземпляра класса Demo

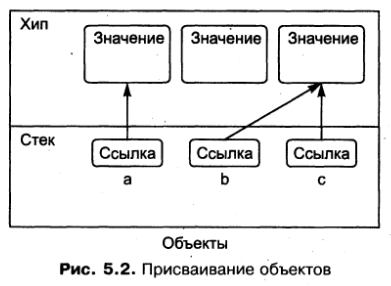
Demo b = new Demo(); // создание другого экземпляра класса Demo

Для каждого объекта при его создании в памяти выделяется отдельная область, в которой хранятся его данные. Кроме того, в классе могут присутствовать статические элементы, которые существуют в единственном экземпляре для всех объектов класса. Часто статические данные называют данными класса, а остальные — данными экземпляра. Функциональные элементы класса не тиражируются, то есть всегда хранятся в единственном экземпляре. Для работы с данными класса используются методы класса (статические методы)у для работы с данными экземпляра — методы экземпляра, или просто методы.

## Присваивание и сравнение объектов

Операция присваивания рассматривалась в разделе «Операции присваивания». Механизм выполнения присваивания один и тот же для величин любого типа, как ссылочного, так и значимого, однако результаты различаются. При присваивании значения копируется значение, а при присваивании ссылки — ссылка, поэтому после присваивания одного объекта другому мы получим две ссылки, указывающие на одну и ту же область памяти (рис. 5.2).

Рисунок иллюстрирует ситуацию, когда было создано три объекта, а, b и с, а затем выполнено присваивание b = с. Старое значение b становится недоступным и очищается сборщиком мусора. Из этого следует, что если изменить значение одной величины ссылочного типа, это может отразиться на другой (в данном случае, если изменить объект через ссылку с, объект b также изменит свое значение).



Аналогичная ситуация с операцией проверки на равенство. Величины значимого типа равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на одни и те же данные (на рисунке объекты b и с равны, но а не равно b даже при равенстве их значений или если они обе равны null).

# Методы

Метод — это функциональный элемент класса, который реализует вычисления или другие действия, выполняемые классом или экземпляром. Методы определяют поведение класса.

Метод представляет собой законченный фрагмент кода, к которому можно обратиться по имени. Он описывается один раз, а вызываться может столько раз, сколько необходимо. Один и тот же метод может обрабатывать различные данные, переданные ему в качестве аргументов.

Синтаксис метода:

[ атрибуты ] [ спецификаторы ] тип имя\_метода ( [ параметры ] )

телометода

Рассмотрим основные элементы описания метода. Первая строка представляет собой заголовок метода. Тело метода, задающее действия, выполняемые методом, чаще всего представляет собой блок — последовательность операторов в фигурных скобках.

Пример простейшего метода:

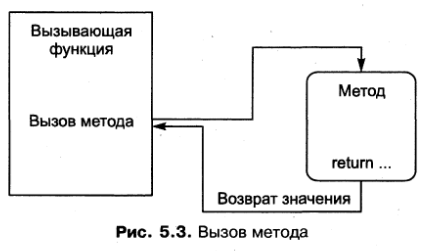
public double Gety()

{

return у;

}

Тип определяет, значение какого типа вычисляется с помощью метода. Часто употребляется термин «метод возвращает значение», поскольку после выполнения метода происходит возврат в то место вызывающей функции, откуда был вызван метод, и передача туда значения выражения, записанного в операторе return (рис. 5.3). Если метод не возвращает никакого значения, в его заголовке задается тип void, а оператор return отсутствует.

Параметры используются для обмена информацией с методом. Параметр представляет собой локальную переменную, которая при вызове метода принимает значение соответствующего аргумента. Область действия параметра — весь метод.

## Свойства

Свойства служат для организации доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки. Синтаксис свойства:

[ атрибуты ] [ спецификаторы ] тип имя\_свойства

{

[ get коддоступа ]

[ set код доступа ]

}

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются как открытые (со спецификатором public), поскольку они входят в интерфейс объекта.

Код доступа представляет собой блоки операторов, которые выполняются при получении (get) или установке (set) свойства. Может отсутствовать либо часть get, либо set, но не обе одновременно.

Если отсутствует часть set, свойство доступно только для чтения (read-only), если отсутствует часть get, свойство доступно только для записи (write-only).

В версии C# 2.0 введена удобная возможность задавать разные уровни доступа для частей get и set. Например, во многих классах возникает потребность обеспечить неограниченный доступ для чтения и ограниченный — для записи.

Спецификаторы доступа для отдельной части должны задавать либо такой же, либо более ограниченный доступ, чем спецификатор доступа для свойства в целом. Например, если свойство описано как public, его части могут иметь любой спецификатор доступа, а если свойство имеет доступ protected internal, его части могут объявляться как internal, protected или private. Синтаксис свойства в версии 2.0 имеет вид

[ атрибуты ] [ спецификаторы ] тип имя\_свойства

{

[ [ атрибуты ] [ спецификаторы ] get коддоступа ]

[ [ атрибуты ] [ спецификаторы ] set код доступа ]

}

Пример описания свойств:

public class Button: Control

{

private string caption: // закрытое поле, с которым связано свойство

public string Caption {

// свойство

get {

// способ получения свойства

return caption:

}

set {

// способ установки свойства

if (caption != value) {

caption = value;

}

}

}

}

## Рекомендации по программированию

При создании класса, то есть нового типа данных, следует хорошо продумать его интерфейс — средства работы с классом, доступные использующим его программистам. Интерфейс хорошо спроектированного класса интуитивно ясен, непротиворечив и обозрим. Как правило, он не должен включать поля данных.

Поля предпочтительнее делать закрытыми (private). Это дает возможность впоследствии изменить реализацию класса без изменений в его интерфейсе, а также регулировать доступ к полям класса с помощью набора предоставляемых пользователю свойств и методов. Важно помнить, что поля класса вводятся только для того, чтобы реализовать характеристики класса, представленные в его интерфейсе с помощью свойств и методов.

Не нужно расширять интерфейс класса без необходимости, «на всякий случай», поскольку увеличение количества методов затрудняет понимание класса пользователем. В идеале интерфейс должен быть полным, то есть предоставлять возможность выполнять любые разумные действия с классом, и одновременно минимально необходимым — без дублирования и пересечения возможностей методов.

Методы определяют поведение класса. Каждый метод класса должен решатьтолько одну задачу (не надо объединять два коротких независимых фрагмента кода в один метод). Размер метода может варьироваться в широких пределах, все зависит от того, какие функции он выполняет. Желательно, чтобы тело метода помещалось на 1-2 экрана: одинаково сложно разбираться в программе, содержащей несколько необъятных функций, и в россыпи из сотен единиц по несколько строк каждая.

Если метод реализует сложные действия, следует разбить его на последовательность шагов и каждый шаг оформить в виде вспомогательной функции (метода со спецификатором private). Если некоторые действия встречаются в коде хотя бы дважды, их также нужно оформить в виде отдельной функции.

Создание любой функции следует начинать с ее интерфейса, то есть заголовка. Необходимо четко представлять себе, какие параметры функция должна получать и какие результаты формировать. Входные параметры обычно перечисляют в начале списка параметров. Предпочтительнее, чтобы каждая функция вычисляла ровно один результат, однако это не всегда оправдано. Если величина вычисляется внутри функции и возвращается из нее через список параметров, необходимо использовать перед соответствующим параметром ключевое слово out. Если параметр значимого типа может изменить свою величину внутри функции, его предваряют ключевым словом ref. Величины ссылочного типа всегда передаются по адресу и, следовательно, могут изменить внутри функции свое значение.

Необходимо стремиться к максимальному сокращению области действия каждой переменной, то есть к реализации принципа инкапсуляции. Это упрощает отладку программы, поскольку ограничивает область поиска ошибки. Следовательно, величины, используемые только в функции, следует описывать внутри нее как локальные переменные.

Поля, характеризующие класс в целом, то есть имеющие одно и то же значение для всех экземпляров, следует описывать как статические. Все литералы, связанные с классом (числовые и строковые константы), описываются как поля-константы с именами, отражающими их смысл.

# Перегрузка методов

Часто бывает удобно, чтобы методы, реализующие один и тот же алгоритм для различных типов данных, имели одно и то же имя. Если имя метода является осмысленным и несет нужную информацию, это делает программу более понятной, поскольку для каждого действия требуется помнить только одно имя. Использование нескольких методов с одним и тем же именем, но различными типами параметров называется перегрузкой методов.

Компилятор определяет, какой именно метод требуется вызвать, по типу фактических параметров. Этот процесс называется разрешением (resolution) перегрузки. Тип возвращаемого методом значения в разрешении не участвует1. Механизм разрешения основан на достаточно сложном наборе правил, смысл которых сводится к тому, чтобы использовать метод с наиболее подходящими аргументами и выдать сообщение, если такой не найдется. Допустим, имеется четыре варианта метода, определяющего наибольшее значение:

// Возвращает наибольшее из двух целых:

int max( int a, int b )

// Возвращает наибольшее из трех целых:

int max( int a, int b, int c )

// Возвращает наибольшее из первого параметра и длины второго:

int max ( int a, string b )

// Возвращает наибольшее из второго параметра и длины первого:

int max ( string b, int a )

Console.WriteLine( max( 1/2 ) );

Console.WriteLine( max( 1. 2. 3 ) );

Console.WriteLine( max( 1, "2" ) ):

Console.WriteLine( max( "1". 2 ) );

При вызове метода max компилятор выбирает вариант метода, соответствующий типу передаваемых в метод аргументов (в приведенном примере будут последовательно вызваны все четыре варианта метода).

Если точного соответствия не найдено, выполняются неявные преобразования типов в соответствии с общими правилами, например, bool и char в int, float в double и т. п. Если преобразование невозможно, выдается сообщение об ошибке. Если соответствие на одном и том же этапе может быть получено более чем одним способом, выбирается «лучший» из вариантов, то есть вариант, содержащий меньшие количество и длину преобразований в соответствии с правилами, описанными в разделе «Преобразования встроенных арифметических типов-значений» (см. с. 45). Если существует несколько вариантов, из которых невозможно выбрать лучший, выдается сообщение об ошибке.

Вам уже известно, что все методы класса должны различаться сигнатурами. Это понятие было введено в разделе «Методы». Перегруженные методы имеют одно имя, но должны различаться параметрами, точнее, их типами и способами передачи (out или ref). Например, методы, заголовки которых приведены ниже, имеют различные сигнатуры и считаются перегруженными:

int max( int a, int b )

int max( int a, ref int b )

Перегрузка методов является проявлением полиморфизма, одного из основных свойств ООП. Программисту гораздо удобнее помнить одно имя метода и использовать его для работы с различными типами данных, а решение о том, какой вариант метода вызвать, возложить на компилятор. Этот принцип широко используется в классах библиотеки .NET. Например, в стандартном классе Console метод WriteLine перегружен 19 раз для вывода величин разных типов.

# Иерархии классов

Управлять большим количеством разрозненных классов довольно сложно. С этой проблемой можно справиться путем упорядочивания и ранжирования классов, то есть объединяя общие для нескольких классов свойства в одном классе и используя его в качестве базового.

Эту возможность предоставляет механизм наследования, который является мощнейшим инструментом ООП. Он позволяет строить иерархии, в которых классы-потомки получают свойства классов-предков и могут дополнять их или изменять. Таким образом, наследование обеспечивает важную возможность многократного использования кода. Написав и отладив код базового класса, можно, не изменяя его, за счет наследования приспособить класс для работы в различных ситуациях.

Это экономит время разработки и повышает надежность программ.

Классы, расположенные ближе к началу иерархии, объединяют в себе общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных особенностей.

Итак, наследование применяется для следующих взаимосвязанных целей:

□ исключения из программы повторяющихся фрагментов кода;

□ упрощения модификации программы;

□ упрощения создания новых программ на основе существующих.

Кроме того, наследование является единственной возможностью использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения.

# Наследование

Класс в C# может иметь произвольное количество потомков и только одного предка. При описании класса имя его предка записывается в заголовке класса после двоеточия. Если имя предка не указано, предком считается базовый класс всей иерархии System.Object:

[ атрибуты ] [ спецификаторы ] class ииякласса [ : предки ]

тело класса

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными далее правилами:

* Если в конструкторе производного класса явный вызов конструктора базового класса автоматически вызывается базового класса безотсутствует, конструктор параметров.
* Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса. Таким образом, каждый конструктор инициализирует свою часть объекта.
* Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации. Вызов выполняется с помощью ключевого слова base. Вызывается та версия конструктора, список параметров которой соответствует списку аргументов, указанных после слова base.

## Виртуальные методы

При раннем связывании программа, готовая для выполнения, представляет собой структуру, логика выполнения которой жестко определена. Если же требуется, чтобы решение о том, какой из одноименных методов разных объектов иерархии использовать, принималось в зависимости от конкретного объекта, для которого выполняется вызов, то заранее жестко связывать эти методы с остальной частью кода нельзя.

Следовательно, надо каким-то образом дать знать компилятору, что эти методы будут обрабатываться по-другому. Для этого в C# существует ключевое слово virtual. Оно записывается в заголовке метода базового класса, например:

virtual public void Passport()

Слово virtual в переводе с английского значит «фактический». Объявление метода виртуальным означает, что все ссылки на этот метод будут разрешаться по факту его вызова, то есть не на стадии компиляции, а во время выполнения программы. Этот механизм называется поздним связыванием.

Для его реализации необходимо, чтобы адреса виртуальных методов хранились там, где ими можно будет в любой момент воспользоваться, поэтому компилятор формирует для этих методов таблицу виртуальных методов (Virtual Method Table, VMT). В нее записываются адреса виртуальных методов (в том числе унаследованных) в порядке описания в классе. Для каждого класса создается одна таблица.

Каждый объект во время выполнения должен иметь доступ к VMT. Обеспечение этой связи нельзя поручить компилятору, так как она должна устанавливаться во время выполнения программы при создании объекта. Поэтому связь экземпляра объекта с VMT устанавливается с помощью специального кода, автоматически помещаемого компилятором в конструктор объекта.

Если в производном классе требуется переопределить виртуальный метод, используется ключевое слово override.

Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса. Это требование вполне естественно, если учесть, что одноименные методы, относящиеся к разным классам, могут вызываться из одной и той же точки программы.

Виртуальные методы базового класса определяют интерфейс всей иерархии. Этот интерфейс может расширяться в потомках за счет добавления новых виртуальных методов. Переопределять виртуальный метод в каждом из потомков необязательно: если он выполняет устраивающие потомка действия, метод наследуется.

Вызов виртуального метода выполняется так: из объекта берется адрес его таблицы VMT, из VMT выбирается адрес метода, а затем управление передается этому методу. Таким образом, при использовании виртуальных методов из всех одноименных методов иерархии всегда выбирается тот, который соответствует фактическому типу вызвавшего его объекта.

С помощью виртуальных методов реализуется один из основных принципов объектно-ориентированного программирования — полиморфизм.. Это слово в переводе с греческого означает «много форм», что в данном случае означает «один вызов — много методов». Применение виртуальных методов обеспечивает гибкость и возможность расширения функциональности класса.

Виртуальные методы незаменимы и при передаче объектов в методы в качестве параметров. В параметрах метода описывается объект базового типа, а при вызове в него передается объект производного класса. В этом случае виртуальные методы, вызываемые для объекта из метода, будут соответствовать типу аргумента, а не параметра.

При описании классов рекомендуется определять в качестве виртуальных те методы, которые в производных классах должны реализовываться по-другому. Если во всех классах иерархии метод будет выполняться одинаково, его лучше определить как обычный метод.

## Абстрактные классы

При создании иерархии объектов для исключения повторяющегося кода часто бывает логично выделить их общие свойства в один родительский класс. При этом может оказаться, что создавать экземпляры такого класса не имеет смысла, потому что никакие реальные объекты им не соответствуют. Такие классы называют абстрактными.

Абстрактный класс служит только для порождения потомков. Как правило, в нем задается набор методов, которые каждый из потомков будет реализовывать по-своему. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс задает интерфейс для всей иерархии, при этом методам класса может не соответствовать никаких конкретных действий. В этом случае методы имеют пустое тело и объявляются со спецификатором abstract.

Абстрактные классы используются при работе со структурами данных, предназначенными для хранения объектов одной иерархии, и в качестве параметров методов. Если класс, производный от абстрактного, не переопределяет все абстрактные методы, он также должен описываться как абстрактный.

Можно создать метод, параметром которого является абстрактный класс. На место этого параметра при выполнении программы может передаваться объект любого производного класса. Это позволяет создавать полиморфные методы, работающие с объектом любого типа в пределах одной иерархии. Полиморфизм в различных формах является мощным и широко применяемым инструментом ООП.