

Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

Т. Л. Романькова

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

КУРС ЛЕКЦИЙ

**по одноименной дисциплине для студентов
специальности 1-40 01 02 «Информационные
системы и технологии (по направлениям)»
дневной формы обучения**

Электронный аналог печатного издания

Гомель 2010

УДК 004.42(075.8)
ББК 32.973-018я73
Р69

*Рекомендовано к изданию научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(протокол № 4 от 14.12.2009 г.)*

Рецензент: доц. каф. «Электроснабжение» ГГТУ им. П. О. Сухого *Т. В. Алферова*

Романькова, Т. Л.

Р69

Объектно-ориентированное программирование : курс лекций по одной дисциплине для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» днев. формы обучения / Т. Л. Романькова. – Гомель : ГГТУ им. П. О. Сухого, 2010. – 97 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: <http://lib.gstu.local>. – Загл. с титул. экрана.

ISBN 978-985-420-967-8.

Даны основные понятия и принципы объектно-ориентированного программирования. Рассмотрены базовые понятия современного языка программирования C#, возможности создания классов и использования стандартных классов библиотеки .Net. Особое внимание уделено механизму наследования и реализации интерфейсов.

Для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» дневной формы обучения.

УДК 004.42(075.8)
ББК 32.973-018я73

ISBN 978-985-420-967-8

© Романькова Т. Л., 2010
© Учреждение образования «Гомельский
государственный технический университет
имени П. О. Сухого», 2010

Введение

Основные задачи курса лекций по дисциплине «Объектно-ориентированное программирование» для студентов специальности «Информационные системы и технологии»:

- познакомиться с концепцией объектно-ориентированного подхода;
- изучить синтаксис объектно-ориентированного языка C# и методы использования основных объектов и конструкций языка;
- познакомиться с основными технологическими приемами разработки объектно-ориентированных программных приложений;
- рассмотреть технологию организации классов, иерархии классов, предопределенных и создаваемых классов;
- освоить использование абстрактных и виртуальных объектов;
- освоить методы и приемы обработки исключительных ситуаций;
- показать применение объектно-ориентированного языка для решения прикладных задач.

Одной из наиболее перспективных технологий на настоящий момент является объектно-ориентированное программирование (ООП). Применение ООП позволяет разрабатывать программное обеспечение повышенной сложности за счет улучшения его технологичности (лучших механизмов разделения данных, увеличения повторяемости кодов, использования стандартизованных интерфейсов пользователя и т. д.).

В настоящей работе будут рассмотрены основные принципы ООП, а также относительно новый объектно-ориентированный язык C#.

Язык программирования C# – один из языков, предназначенных для написания приложений для архитектуры Microsoft .NET (одной из основных платформ разработки, развертывания и исполнения распределенных приложений). Разработан в конце 1990-х годов компанией Microsoft. Язык C# можно использовать для создания консольных приложений, Windows-приложений, Web-приложений.

Тема 1. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

1.1. Основные направления в программировании

Рассмотрим эволюцию языков программирования и основные подходы к разработке программных систем:

- ранние *неструктурные* подходы;
- *структурный*, или *модульный*, подход (задача разбивается на подзадачи, затем на алгоритмы, составляются их структурные схемы и осуществляется реализация);
- *функциональный* подход;
- *логический* подход;
- *объектно-ориентированный* подход;
- *смешанный* подход (некоторые подходы можно комбинировать).

Первые языки программирования были довольно примитивны и ориентированы в основном на численные расчеты. Программы, написанные на ранних языках программирования, представляли собой линейные последовательности элементарных операций с регистрами, в которых хранились данные.

Ранние языки программирования существенно зависели от аппаратной архитектуры конкретного компьютера и приблизительно соответствовали современным машинным кодам или языкам ассемблера.

В языках программирования так называемого «высокого уровня» одна инструкция (или оператор) соответствует последовательности из нескольких низкоуровневых инструкций или команд. То есть программа по сути представляет собой набор директив, обращенных к компьютеру. Такой подход называется императивным.

Еще одна особенность языков высокого уровня – возможность повторного использования ранее написанных программных блоков посредством их идентификации и последующего обращения к ним. Такие блоки получили название *функций* или *процедур*. Примеры: ALGOL, COBOL, Pascal, Basic, Fortran, C.

В 60-х годах возникает новый подход к программированию – декларативный подход. Суть подхода состоит в том, что программа представляет собой не набор команд, а описание действий, которые необходимо осуществить. Примеры декларативных языков программирования: SML, Haskell, Prolog, LISP.

Одним из путей развития декларативного стиля программирования стал *функциональный* подход.

Особенностью данного подхода является то, что любая программа может интерпретироваться как функция с одним или несколькими аргументами.

В 70-х годах возникла еще одна ветвь языков декларативного программирования, связанная с проектами в области искусственного интеллекта, – *языки логического программирования*.

Языки логического программирования базируются на классической логике и применимы для систем логического вывода, в частности, для так называемых экспертных систем. Согласно логическому подходу программа представляет собой совокупность правил или логических высказываний. В качестве примеров языков логического программирования можно привести Prolog (название возникло от слов PROgramming in LOGic) и Mercury.

1.2. Понятие и основные принципы объектно-ориентированного программирования

Объектно-ориентированное программирование – основная парадигма программирования 80–90-х годов, которая, судя по всему, сохранится и в течение последующего десятилетия.

Объектно-ориентированное программирование (ООП) – это методика разработки программ, в основе которой лежит понятие «объект».

ООП – это стиль, который фиксирует поведение реального мира так, что детали его составных частей скрыты, и тот, кто решает задачу, мыслит в терминах, присущих самой задаче, а не языку программирования. Объект – это некоторая структура, соответствующая объекту реального мира, его поведению. Каждый объект – робот, автомобиль, человек – обладает определенными характеристиками. Ими могут служить вес, рост, максимальная скорость, фамилия, грузоподъемность и т. д. Объект способен производить какие-то действия: перемещаться в пространстве, поворачиваться, поднимать, расти, уменьшаться или увеличиваться, есть, пить, изменяя свои первоначальные характеристики. Таким образом, при объектном подходе интеграция данных и процедур их обработки определяется структурой предметной области, т. е. набором моделируемых объектов, их взаимосвязью или взаимодействием в рамках решаемой задачи. В программе может быть образовано любое количество однотипных объек-

тов, а в отличие от данных объекты обладают собственными процедурами их обработки.

Основные принципы ООП:

- наследование;
- инкапсуляция;
- полиморфизм.

Основные *достоинства* ООП:

- использование более естественных понятий предметной области;
- простота введения новых понятий на основе существующих;
- отображение в библиотеке классов наиболее общих свойств и отношений между объектами моделируемой предметной области;
- естественность отображения пространства решаемой задачи в пространство объектов программы;
- простота внесения изменений в определения классов и программу в целом;
- упрощение составления и понимания программы благодаря полиморфизму;
- упрощение структуры программы вследствие инкапсуляции свойств и поведения объекта.

Недостатки ООП:

- снижение быстродействия программ, особенно при необходимости позднего связывания;
- большие затраты на разработку библиотеки классов, поэтому ООП целесообразно применять при создании больших программных систем, а не при написании маленьких единичных программ;
- необходимость анализа всей иерархии классов для правильного использования свойств базовых классов.

Инкапсуляция («заклучение в оболочку») представляет собой локализацию в рамках объекта всех данных об объекте, которые характеризуют его внутреннюю структуру и поведение с запрещением непосредственного доступа к тем данным, которые нецелесообразно (или даже опасно) предоставлять в распоряжение пользователя. Так, например, объект класса *счетчик* может содержать атрибут целого типа *текущее_значение*, и, по всей видимости, было бы опасно дать пользователю возможность менять это значение произвольным образом, добавляя к нему любое целое значение или вычитая его.

Связывая код и описания данных вместе, объект помогает упорядочить их. Если требуется получение значения одного из атрибутов

объекта, то вызывается метод, принадлежащий этому объекту, который возвращает значение нужного атрибута. Разумеется, это возможно только тогда, когда такой метод представлен в описании объектного типа. Аналогично, чтобы задать значение атрибута, вызывается метод, который присваивает этому атрибуту новое значение.

Разные объектно-ориентированные языки программирования решают проблему инкапсуляции по-разному. Так, например, в Сmolтоке все поля объекта (внутренние переменные) скрыты от пользователя и доступны только в телах методов данного класса, в то же время все методы объекта открыты пользователю. Более гибкая схема реализована в С++, Яве и С#. В этих языках каждый компонент объекта (поле или метод) может помечаться одним из спецификаторов доступа **private**, **protected** или **public**. Компонент, помеченный как **private** (скрытый), может использоваться только в телах методов данного класса, компонент, помеченный как **protected** (защищенный), может дополнительно использоваться в наследниках данного класса (т. е. в телах их методов), а компонент, помеченный как **public** (открытый), может использоваться в любом месте программы. Такое гибкое разграничение доступа к компонентам объекта позволяет избежать нежелательных искажений свойств объекта и обеспечить эффективный доступ к ним, когда это необходимо (прямой доступ к элементу данных объекта обычно более эффективен, чем доступ к нему посредством метода).

Скрытие внутренней структуры объекта и снабжение его осмысленными операциями, доступными извне, позволяет в максимальной степени изолировать содержание объекта от внешнего мира, т. е. ограничить и контролировать доступ к компонентам объекта. В результате замена или модификация компонентов, как правило, не влечет за собой плохо контролируемые последствия для программы в целом.

Итак, к основным свойствам инкапсуляции относятся следующие возможности:

- совместное хранение данных и функций (т. е. свойств и методов) внутри объекта;
- сокрытие внутренней информации от пользователя (что обеспечивает большую безопасность приложения);
- изоляция пользователя от особенностей реализации (что обеспечивает независимость от машины и потенциально дружественный интерфейс приложений).

Наследование – это свойство классов порождать своих потомков и наследовать элементы своих родителей. Класс-потомок автоматически наследует от родителя все свойства и методы, а также может

вводить новые элементы и даже заменять (переопределять) операции родительского класса. Наследование в ООП позволяет адекватно отражать родственные отношения между объектами предметной области. Если класс **В** обладает всеми свойствами класса **А** и еще имеет дополнительные свойства, то класс **А** называется *базовым* (родительским), а класс **В** – *порожденным* (наследником) класса **А**.

Механизм наследования упрощает процесс построения иерархии классов, что сокращает затраты на написание новых классов на основе существующих и обеспечивает исключительную гибкость системы классов. Программист обычно определяет базовый класс, обладающий наиболее общими свойствами, а затем создает последовательность потомков, которые обладают своими специфическими свойствами. В результате получается иерархия наследования свойств классов. Пример иерархии классов: **фигура** -> **точка** -> **круг** -> **цилиндр**.

Полиморфизм в буквальном переводе с греческого означает «много форм». *Полиморфизмом имени* называется свойство какого-либо имени программы одновременно обозначать различные сущности. Но совсем необязательно связывать его только с именами. Можно говорить о полиморфных знаках операций. Например, знак операции «+» может обозначать операцию сложения целых, сложения вещественных чисел или сцепления строк. Полиморфизм можно также определить как способность объектов-родственников по-разному осуществлять однотипные действия, т. е. однотипные действия у множества классов, связанных отношением иерархии, имеют разные формы проявления. Например, метод «нарисовать на экране» должен реализовываться по-разному для классов **точка**, **круг** и **цилиндр**.

1.3. Объекты и классы

В объектно-ориентированном подходе все или большинство данных рассматриваются как активные объекты, обладающие всем множеством операций (методов) своего класса. Неважно, что все определения операций, как правило, сосредоточены в определении класса объектов, и в реализации все операции одного класса будут оторваны от объектов (трудно представить себе реализацию, в которой каждый объект сопровождался бы индивидуальной копией всего множества своих методов). Главное, чтобы программист составлял программу в терминах взаимодействующих объектов и рассматривал как активные элементы именно объекты, а не использующие их процедуры и функции.

Класс объектов (или просто **класс**) – это образец, из которого создаются объекты с указанными свойствами. Объекты одного класса имеют одну и ту же структуру и обладают одними и теми же операциями и вследствие этого – одним и тем же поведением. Определение класса задает программную структуру, в которой данные и операции представляют единое целое и отражают свойства и поведение этого целого в рамках моделируемой предметной области. В отличие от модуля, в котором на состав данных и процедур накладывается меньше смысловых ограничений, в классе объектов вводятся только те данные и операции, которые необходимы для описания свойств и поведения объектов этого класса.

Классы объектов выделяются в процессе анализа предметной области с использованием идей абстрагирования от несущественных деталей и классификации родственных объектов. Например, на основании анализа авиационной техники можно выделить класс объектов **самолет**. При этом мы абстрагировались от таких свойств как форма крыльев, длина фюзеляжа, используемые материалы конструкций, расположение крыльев и т. п. К числу основных свойств этого класса можно отнести скорость полета, высоту полета, грузоподъемность и т. п. Классы объектов могут образовывать иерархию, отражающую иерархию понятий предметной области.

Итак, в программе **класс** – это специальный тип данных, включающий описание данных и описание процедур и функций (методов), которые могут быть выполнены над представителем класса – объектом.

Класс объектов характеризуется уникальным набором свойств и обозначается уникальным именем, как и любой тип данных. **Объект** – это конкретный экземпляр класса. Создаваемые объекты данного класса различаются значениями (степенью проявления) своих свойств и снабжаются уникальными внутренними идентификаторами.

Тема 2. БАЗОВЫЕ КОНСТРУКЦИИ ЯЗЫКА C#

2.1. Характеристика языка

C# – относительно новый язык программирования, который обладает следующими преимуществами:

1. Он спроектирован и разработан специально для применения с Microsoft Net Framework (развитой платформой разработки, развертывания и исполнения распределенных приложений).

2. Это язык, основанный на современной объектно-ориентированной методологии проектирования, при разработке которого специалисты Microsoft опирались на опыт создания других языков, построенных в соответствии объектно-ориентированными принципами, которые были впервые предложены около 20 лет назад.

3. Язык C# является одним из компонентов платформы .NET. Платформа .NET включает в себя:

- языки программирования VB.NET, J#.NET, C++.NET и др.;
- среду разработки приложений Visual Studio.NET для нескольких языков программирования;
- компиляторы, переводящие программу с соответствующего языка в промежуточный язык Microsoft Intermediate Language;
- общезыковую среду выполнения Common Language Runtime (CLR) или виртуальную машину CLI;
- библиотеку классов.

Все классы библиотеки .NET, а также создаваемые программистом в среде .NET классы имеют одного предка – класс **object**. Все классы сгруппированы в *пространства имен*. Любая создаваемая в .NET программа использует пространство имен **System**, в котором определены классы, обеспечивающие базовую функциональность.

2.2. Идентификаторы

Идентификатор – имя любого объекта программы (переменной, константы, процедуры и др.).

Идентификатор может включать буквы в кодировке **Unicode**, цифры и символ подчеркивания. Идентификатор не может начинаться с цифры. Прописные и строчные буквы в идентификаторах различаются. Например, `Vasja1`, `VASJA1` и `VaSjA1` – это разные идентификаторы. Длина идентификатора не ограничена.

Для использования ключевого слова в качестве идентификатора его нужно предварить символом @.

Например, `@if`.

В C# для имен используют обычно две нотации: Паскаля и Camel.

В нотации Паскаля каждое слово в имени начинается с прописной буквы: `MaxDlina`.

В нотации Camel каждое слово в имени, кроме первого, начинается с прописной буквы: `maxDlina`, `myBestFriend`.

2.3. Классификация типов данных

Под типом данных понимается множество допустимых значений этих данных, а также совокупность операций над ними. В табл. 2.1 приводятся основные виды типов данных в C#.

Таблица 2.1

Основные виды типов данных

По строению элемента	По «создателю»	По выделению памяти	По способу хранения элементов
Простые	Встроенные	Статические	Значимые
Структурированные	Определяемые программистом	Динамические	Ссылочные

На рис. 2.1 отражен один из способов распределения типов данных на группы.



Рис. 2.1. Классификация типов данных в C#

Концептуальная разница между ссылочными и значимыми типами состоит в том, что тип значений хранит данные непосредственно, а ссылочный тип хранит ссылку на значение. Эти типы хранятся в разных местах памяти: типы значений сохраняются в области, известной как стек, а ссылочные типы – в динамической области памяти, называемой *управляемой кучей*.

2.4. Встроенные типы данных

Встроенные типы C# однозначно соответствуют стандартным классам библиотеки .NET, определенным в пространстве имен System. Язык C# поддерживает восемь предопределенных целочисленных типов (табл. 2.2).

Таблица 2.2

Целые типы данных

Имя	Тип CTS	Описание	Диапазон
sbyte	System.Sbyte	8-битное целое значение	-128:127
short	System.Int16	16-битное целое значение	-32768:32767
int	System.Int32	32-битное целое значение	-2147483648:2147483647
long	System.Int64	64-битное целое значение	-9223372036854775808:9223372036854775807
byte	System.Byte	8-битное целое без знака	0:255
ushort	System.UInt16	16-битное целое без знака	0:65535
uint	System.UInt32	32-битное целое без знака	0:4294967295
ulong	System.UInt64	64-битное целое без знака	0:18446744073709551615

Язык C# поддерживает три типа с плавающей точкой (табл. 2.3).

Таблица 2.3

Вещественные типы данных

Имя	Тип CTS	Описание	Количество знаков	Диапазон
float	System.Single	32-битное с плавающей точкой одинарной точности	7	$\pm 1,5 \times 10^{-45}$: $\pm 3,4 \times 10^{38}$
double	System.Double	32-битное с плавающей точкой двойной точности	15/16	$\pm 5,0 \times 10^{-324}$: $\pm 1,7 \times 10^{308}$
decimal	System.Decimal	128-битное с плавающей точкой в десятичной нотации	28	$\pm 1,0 \times 10^{-28}$: $\pm 7,9 \times 10^{28}$

Тип `bool` в C# используется для хранения булевских значений – `true` или `false` (табл. 2.4).

Таблица 2.4

Логический тип данных

Имя	Тип CTS	Описание	Значение
<code>bool</code>	<code>System.Boolean</code>	32-битное с плавающей точкой одинарной точности	<code>true</code> или <code>false</code>

Для хранения одиночных символов C# поддерживает тип данных `char` (табл. 2.5).

Таблица 2.5

Символический тип данных

Имя	Тип CTS	Значение
<code>Char</code>	<code>System.Char</code>	Представляет отдельный 16-битовый символ

Язык C# содержит два predefined ссылочных типа, описанных в табл. 2.6.

Таблица 2.6

Предопределенные ссылочные типы

Имя	Тип CTS	Описание
<code>object</code>	<code>System.Object</code>	Корневой тип, от которого наследуются все типы CTS
<code>string</code>	<code>System.String</code>	Строка символов Unicode

Имя встроенного типа можно заменить именем соответствующего класса библиотеки. Тип `object` – первичный родительский тип, от которого наследуются все внутренние и пользовательские типы. У встроенных типов данных есть методы и поля, к которым можно обращаться. Например, `int.MaxValue` (`System.Int32.MaxValue`) – максимальное значение типа `int`.

2.5. Приведение типов

В одном выражении могут быть сгруппированы операнды различных типов. Однако возможность подобного «смешения» при определении значения выражения приводит к необходимости применения дополнительных усилий по приведению значений операндов к «общему типу». Иногда приведение значения к другому типу происходит автоматически. Такие преобразования называются *неявными*.

Но в ряде случаев программист должен явным образом указывать необходимость преобразования, используя выражения приведения типа или обращаясь к специальным методам преобразования, определенным в классе **System.Convert**, которые обеспечивают преобразование значения одного типа к значению другого (в том числе значения строкового типа к значениям базовых типов).

Преобразование типа создает значение нового типа, эквивалентное значению старого типа, однако при этом не обязательно сохраняется идентичность (или точные значения) двух объектов.

Если операнды, входящие в выражение, одного типа и операция для этого типа определена, то результат выражения будет иметь тот же тип. Если операнды разного типа или операция для этого типа не определена, перед вычислениями автоматически выполняется преобразование типа по правилам неявного преобразования (рис. 2.2).

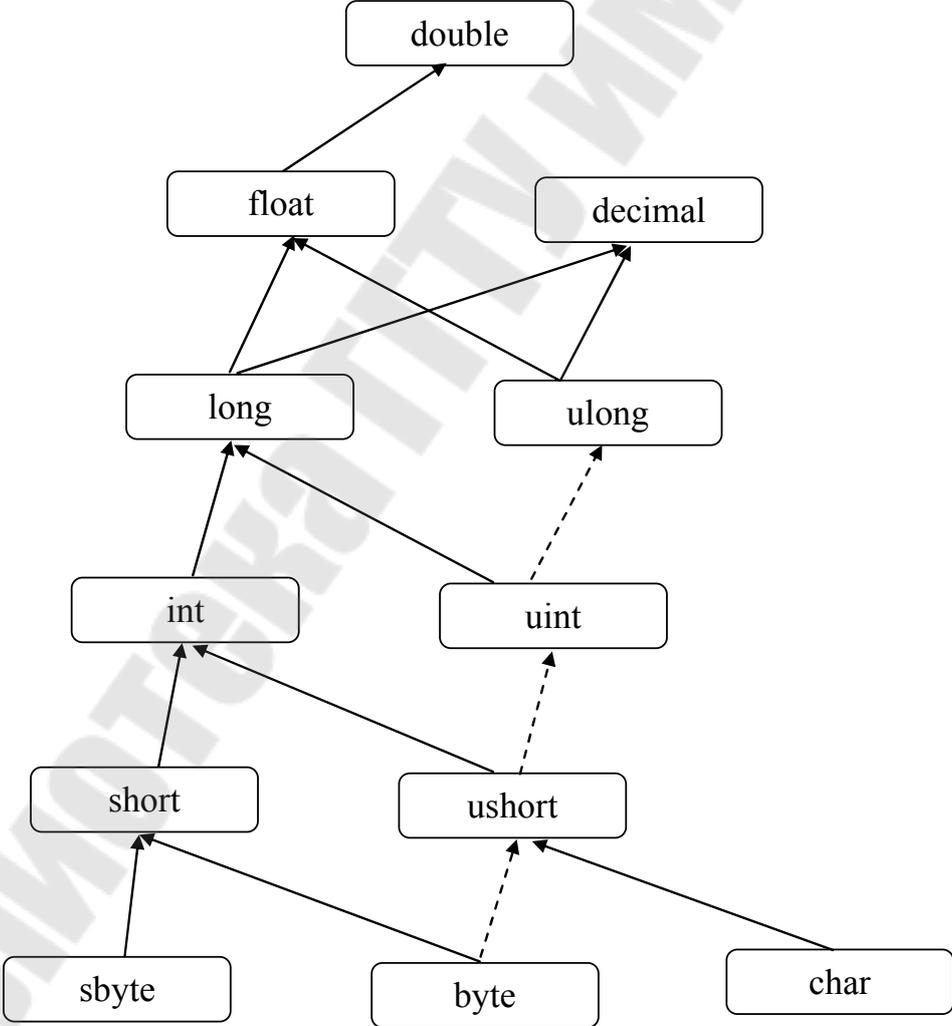


Рис. 2.2. Правила неявного преобразования типов

Преобразование выполняется непосредственно из исходного типа в результирующий.

Если неявного преобразования из одного типа в другой не существует, можно задать явное преобразование с помощью операции (тип) *x*. За результатом должен следить программист.

Пример

```
long y = 500;  
int x = (int) y;  
byte z = (byte) x;
```

В последнем случае данные будут потеряны.

Различаются расширяющее и сужающее преобразования.

Расширяющее преобразование – значение одного типа преобразуется к значению другого типа, которое имеет такой же или больший размер. Расширяющее преобразование считается безопасным, поскольку исходная информация при таком преобразовании не искажается.

Сужающее преобразование – значение одного типа преобразуется к значению другого типа, которое имеет меньший размер (из 64-разрядного в 32-разрядное). Такое преобразование потенциально опасно потерей значения. Сужающие преобразования могут приводить к потере информации. Если сужающее преобразование обеспечивается методами класса **System.Convert**, то потеря информации сопровождается генерацией исключения.

2.6. Переменные и константы

Все переменные, используемые в программе должны быть описаны явным способом.

При описании задаются ее имя и тип:

```
тип имя1,[имя2, имя3, ...];
```

Например:

```
int dlna; float b;
```

При описании можно инициализировать переменные:

```
int x = 1, b;
```

```
float y = 0.1f, n = y * 2 + 2.1;
```

При описании инициализировать переменные не обязательно, но обязательно их инициализировать перед вычислениями. Область действия переменной начинается в точке ее описания и распространяется до конца блока, внутри которого она описана. **Блок** – это код, заклю-

ченный в фигурные скобки. Имя переменной должно быть уникальным в области ее действия.

Предваряя переменную ключевым словом **const** при ее объявлении или инициализации, вы объявляете ее как константу:

const int a=100;

Константы должны инициализироваться при объявлении, и однажды присвоенные им значения не должны никогда меняться, значение константы должно быть вычислено при компиляции.

Использование в программе констант различных типов представлено в табл. 2.7.

Таблица 2.7

Типы констант

Константа	Описание	Примеры
Логическая	true или false	false
Целая	Десятичная: последняя цифра с суффиксом или без Шестнадцатеричная	5u 0 2256L 0xA 0X00FFL
Вещественная	С фиксированной точкой С порядком	3.7 3.7F 5e-3 .11E-6D
Символьная	Символ в апострофах Управляющая последовательность	'A' '*' '\n' '\a'
Строковая	Последовательность символов, заключенная в кавычки	"Привет! " "\tпривет\nКак дела? " @"D:\program\readme.txt "
null	Значение по умолчанию для величин ссылочных типов	null

2.7. Структура консольной программы

Программа на C# состоит из взаимодействующих между собой классов.

При создании консольного приложения средой автоматически создается следующая заготовка:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ConsoleApplication1
```

```

{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}

```

Директива **using System;** разрешает использовать имена классов из пространства имен **System** без указания имени пространства.

Ключевое слово **namespace** создает для проекта собственное пространство имен. Таким образом, программным объектам можно давать имена, не волнуясь о том, что они могут совпасть с именами в других пространствах имен.

Пока программа содержит только один класс **Program**, содержащий только один элемент – метод **Main**.

В каждом приложении **обязательно** должен присутствовать метод **Main**. **Именно с него начинается выполнение программы.**

2.8. Ввод-вывод в консольном приложении

Для работы с консолью в C# применяется класс **Console**, определенный в пространстве имен **System**. В классе **Console** существует несколько вариантов методов с именами **Write** и **WriteLine**, предназначенных для вывода значений различных типов. Методы вывода перегружены для всех встроенных типов данных. Для доступа к методу используется операция доступа (точка):

Console.WriteLine(параметр);

Пример

```
Console.WriteLine("Здравствуй, я ваша тетя!");
```

или

```
Console.Write ("Здравствуй, я ваша тетя!");
```

Если метод **WriteLine (Write)** вызван с одним параметром, то он может быть любого встроенного типа. Если требуется вывести в одной строке несколько величин различного типа, перед передачей для вывода их нужно «склеить» в одну строку с помощью операции «+».

Пример

```
int x = 3;
```

```
Console.WriteLine("x="+x);
```

В случае форматного вывода используется другой вариант метода с несколькими параметрами.

Первым параметром передается строковая константа, содержащая обычные символы, которые выводятся на экран, а также управляющие последовательности и параметры в фигурных скобках.

Эти параметры представляют собой номера переменных в списке вывода и при выводе заменяются на значения соответствующих переменных (*Нумерация с нуля*).

Пример

```
int x = 3;
float y=0.6f;
Console.WriteLine("x={0}\t y={1}",x,y );
Console.WriteLine("x="+x+" y="+y);
```

Для каждого параметра можно задать ширину поля вывода. Например:

```
Console.WriteLine("x={0,5}\ty={1,4}",x,y );
```

Для арифметических типов можно использовать спецификаторы формата, которые указываются после двоеточия:

```
Console.WriteLine("x={0,5:f3}\ty={1,4}",x,y );
```

Основные спецификаторы:

C (c) – вывод значения в денежном формате. После спецификатора можно задать длину дробной части.

Например:

```
double x = 3.5672;
Console.WriteLine("{0,5:c}",x);
```

Результат **3,57р.**

При выполнении оператора

```
Console.WriteLine("{0,5:c3}",x);
```

Результат **3,567р.**

D (d) – вывод целых значений. После спецификатора можно указать ширину поля вывода, при этом недостающие места слева заполняются нулями.

Например, после выполнения

```
int x = 25;
Console.WriteLine("{0,6:d4}",x);
```

Результат **0025**.

F (f) – вывод значений с фиксированной точностью. После спецификатора можно задать длину дробной части.

Например, **Console.WriteLine("{0,6:f3}",x);**

Результат **25,000**

P (p) – вывод числа в процентном формате.

Например:

```
float x = 0.25f;
```

```
Console.WriteLine("{0,6:p2}", x);
```

Результат **25,00%**

Для выравнивания значения по левому краю поля вывода перед шириной поля вывода нужно поставить минус.

Например, **Console.WriteLine("{0,-10:p2}***",x);**

В классе **Console** для ввода строки можно использовать метод **ReadLine()**. Например:

```
Console.Write("Как вас зовут?\n");
```

```
string name = Console.ReadLine();
```

Для ввода символа можно использовать метод **Read()**, который возвращает код символа типа **int** или **-1**, если символ не был введен. Поэтому требуется явное преобразование в тип **char**.

```
Console.Write("Введите символ\n");
```

```
char name = (char) Console.Read();
```

Ввод чисел с клавиатуры можно осуществить в два этапа:

1) ввести число в виде строки символов;

2) преобразовать строку в переменную соответствующего типа.

Преобразование выполняется с помощью класса **Convert** из пространства имен **System** с использованием соответствующих методов класса **ToInt32(s)**, **ToDouble(s)**, **ToByte(s)** и т. д.

Здесь **s** – параметр типа **string**.

Пример

```
string s;
```

```
s = Console.ReadLine();
```

```
byte h = Convert.ToByte(s);
```

Преобразование можно также выполнить с помощью метода **Parse**, который есть в каждом стандартном арифметическом классе.

Пример

```
byte w = byte.Parse(s);
```

2.9. Методы класса Math

В классе **Math** пространства имен **System** реализованы всевозможные математические функции (табл. 2.8).

Таблица 2.8

Методы класса Math

Функция в математике	Метод класса Math
$ x $	Abs(x)
Arccos x	Acos(x)
Arcsin x	Asin(x)
Arctan x	Atan(x)
Округление до большего целого	Ceiling(x)
Натуральный логарифм	Log(x)
Десятичный логарифм	Log10(x)
Число e	E
Число π	PI
Округление	Round(x)
e^x	Exp(x)
tg x	Tan(x)
x^y	Pow(x,y)
\sqrt{x}	Sqrt(x)
Остаток от деления	IEEERemainder(x,y)
Округление до меньшего целого	Floor(x)
Знак числа	Sign(x)

Тема 3. КЛАССЫ, ОСНОВНЫЕ ЭЛЕМЕНТЫ КЛАССА

3.1. Основные понятия и описание классов

Класс – это ссылочный тип данных, включающий описание данных и описание функций, которые могут быть выполнены над представителем класса – объектом.

В класс могут входить следующие элементы:

- *константы*, которые хранят неизменяемые значения, связанные с классом;
- *поля*, содержащие данные класса;
- *методы*, реализующие действия, выполняемые классом или экземпляром;
- *свойства*, которые определяют характеристики класса в совокупности со способами их задания и получения (методами записи и чтения);

- *конструкторы*, выполняющие действия по инициализации экземпляров или класса в целом;
- *деструкторы*, определяющие действия, выполняемые перед тем как объект будет уничтожен;
- *индексаторы*, обеспечивающие возможность доступа к элементам класса по их порядковому номеру;
- *операции*, задающие действия с объектами с помощью знаков операций;
- *события*, на которые может реагировать объект, определяют уведомления, которые может генерировать класс;
- *типы* – внутренние по отношению к классу типы данных.

Класс описывается следующим образом:

```
[ спецификаторы] class Имя_Класса [ : предки]
{
    тело класса
}
```

Спецификаторы определяют свойства класса и доступность для других элементов программы. Допустимо использование следующих спецификаторов:

public – доступ не ограничен;

protected – для вложенных классов доступ только из элементов данного и производных классов;

internal – доступ только из данной программы;

private – для вложенных классов доступ только из элементов класса, внутри которого описан данный класс.

Для невложенных классов используются только спецификаторы **public** и **internal**.

Объекты (экземпляры) класса создаются явным или неявным образом (программистом или системой). Для явного создания экземпляра используется операция **new**.

Формат операции: **new** Имя_класса ([аргументы]).

Например, пусть имеется следующее описание класса:

```
class Primer1 {}
```

Тогда для создания объектов **p1** и **p2** в программе следует набрать

```
Primer1 p1 = new Primer1();
```

```
Primer1 p2 = new Primer1();
```

3.2. Поля и константы класса

Данные, инкапсулированные в классе, представлены полями и константами.

Переменные, описанные в классе, называются *полями класса*.

Синтаксис описания элемента данных:

[спецификаторы] [const] тип имя [= начальное_значение];

Спецификаторы полей и констант:

public – доступ к элементу не ограничен;

protected – доступ только из данного и производных классов;

internal – доступ только из данной сборки;

private – доступ только из данного класса;

static – одно поле для всех экземпляров класса;

readonly – поле доступно только для чтения.

По умолчанию все элементы класса считаются закрытыми (**private**).

Желательно определять переменные класса как закрытые, чтобы только методы того же класса имели доступ к их значениям.

Поля, характеризующие класс в целом, т. е. имеющие одно и то же значение для всех экземпляров класса, следует описывать как *статические*.

При создании каждого объекта (экземпляра) класса в памяти выделяется отдельная область, в которой хранятся его данные. Статические поля класса и константы существуют в *единственном* экземпляре для всех объектов класса.

Обращение к статическому полю класса:

Имя_класса. имя_поля

Обращение к константе класса:

Имя_класса. имя_константы

Обращение к обычному полю (полю экземпляра) класса:

Имя_объекта. имя_поля

Например, пусть в приложении описан класс **Primer1**:

```
class Primer1
{
    public float x, y = 2.5f; //поля данных с неограниченным
    доступом
    public const int f = 1, z = 5; // константы
    public static char h = 'A', h1; //статические поля данных
    double b; // закрытое поле данных
}
```

Обратиться к полям этого класса можно следующим образом:

```
class Program
{
    static void Main(string[] args)
    {
        Primer1 P1 = new Primer1();
        Primer1 P2 = new Primer1();
        Console.WriteLine(P1.x); Console.WriteLine(P1.y);
        Console.WriteLine(P2.x); Console.WriteLine(P2.y);
        Console.WriteLine(Primer1.f);
        Console.WriteLine(Primer1.z);
        Console.WriteLine(Primer1.h);
        Console.WriteLine(Primer1.h1);
    }
}
```

Оператор **Console.WriteLine(P1.b);** будет ошибочным, т. к. обращаться к закрытому полю данных из другого класса нельзя. Доступ к такому полю можно получить только посредством метода. Результат выполнения программы:

```
0
2,5
0
2,5
1
5
A
пустая строка
```

Все поля автоматически инициализируются (статические – при инициализации классов, обычные – при создании объектов). Числовые получают значение 0, символьные и строковые – пустая строка. После этого полям присваиваются значения, указанные при явной инициализации (если таковая имеется).

3.3. Методы

Метод – это оформленный особым образом поименованный фрагмент кода, который реализует вычисления или другие действия, выполняемые классом или объектом. Метод описывается один раз, а вызываться может необходимое количество раз.

Синтаксис метода:

```
[ спецификаторы ] тип имя ( [параметры])  
{  
    тело метода  
}
```

Спецификаторы метода:

public – доступ не ограничен;

protected – доступ только из данного и производных классов;

internal – доступ только из данной сборки;

private – доступ только из данного класса (по умолчанию);

static – статический метод.

Для работы со статическими данными класса используются статические методы (**static**), для работы с данными экземпляра – методы экземпляра (просто методы).

Тип в заголовке метода определяет тип результата работы метода.

Для передачи значения выражения в качестве результата работы метода используется оператор

```
return выражение;
```

Пример

```
class Primer2  
{ string s; // закрытое поле класса  
  public string vvods() // метод для ввода значения поля s  
  {  
    s = Console.ReadLine();  
    return s;  
  }  
}
```

Если метод не возвращает никакого значения, в заголовке указывается тип **void**, а оператор `return` отсутствует.

Пример

```
public void vvods()  
{ Console.WriteLine(s); }
```

Параметры в заголовке используются для обмена информацией с методом и определяют множество значений аргументов, которые можно передавать в метод. Для каждого параметра указывается тип.

Пример

```
static int maximum(int x, int y) // метод, возвращающий
```

```

// максимальное из двух целых чисел
{
    if (x > y) return x;
    else     return y;
}

```

Обращение к статическому методу класса:

Имя_класса. имя_метода([аргументы])

Обращение к нестатическому методу класса:

Имя_объекта. имя_метода([аргументы])

Если метод возвращает значение, то он вызывается отдельным оператором, а если не возвращает – то в составе выражения в правой части оператора присваивания.

Пример

```

Primer2 S1 = new Primer2(); // создание объекта класса Primer2
string g = S1.vvods();
S1.vyvods();

```

При вызове метода из другого метода того же класса имя класса или экземпляра можно не указывать.

При вызове метода с параметрами количество аргументов должно совпадать с количеством параметров в заголовке метода. Кроме того, должно существовать неявное преобразование типа аргумента к типу соответствующего параметра.

Например, если статический метод **maximum** является методом класса **Primer2**, то из метода **Main** можно обратиться к нему следующим образом:

```

int max = Primer2.maximum(5, 8); // обращение к статическому
                                // методу по имени класса

```

или

```

int max = Primer2.maximum(c1, c2+6);

```

Если метод **maximum** является методом класса **Program**, то из метода **Main** можно обратиться к нему следующим образом:

```

int max = maximum(5, 8); // не указывается имя класса.

```

В С# предусмотрены следующие виды параметров:

- параметры-значения;
- параметры-ссылки;
- выходные параметры;

– параметры-массивы.

При описании параметра-значения в заголовке метода указывается только тип. Параметр-значение представляет собой локальную переменную, которая получает в качестве своего значения копию значения аргумента. При вызове метода в качестве соответствующего аргумента на месте параметра-значения может находиться выражение, в том числе переменная и константа. Например, описанный выше метод **maximum** имеет два параметра-значения.

Если в методе требуется изменить значение передаваемых в качестве параметров величин, используют *параметры-ссылки*. При описании параметра-ссылки в заголовке метода перед указанием типа помещают ключевое слово **ref**. При вызове метода в область параметров копируется не значение аргумента, а его адрес, т. е. метод работает непосредственно с переменной из вызывающей функции и может ее изменить. При вызове метода в качестве аргумента на месте параметра-ссылки может находиться **только ссылка на инициализированную переменную точно того же типа со словом ref перед аргументом**.

Пример

```
class Program
{
    static void Udvoenie(ref int x, ref float y)
    { x = 2 * x; y *= 2;
    }
    static void Main(string[] args)
    {
        int a = 3; float b = 3.5f;
        Udvoenie(ref a, ref b);
        Console.WriteLine("a=" + a + " b=" + b);
    }
}
```

Если нет необходимости инициализировать переменную-аргумент до вызова метода, можно использовать выходные параметры. При описании выходного параметра в заголовке метода перед указанием типа помещают ключевое слово **out**.

Выходному параметру **обязательно** должно быть присвоено значение внутри метода, а в вызывающем коде переменную достаточно описать.

При вызове метода перед соответствующим аргументом тоже указывается слово **out**.

Пример

```
class Program
{
    static void Vvod(out int x, out float y)
    {
        string s;
        Console.WriteLine("Введите целое число");
        s = Console.ReadLine(); x = Convert.ToInt32(s);
        Console.WriteLine("Введите вещественное число");
        s = Console.ReadLine(); y = Convert.ToSingle(s);
    }
    static void Main(string[] args)
    {
        int a; float b;
        Vvod(out a, out b);
        Console.WriteLine("a=" + a + " b=" + b);
        Console.ReadKey();
    }
}
```

В любой нестатический метод автоматически передается *скрытый* параметр **this**, в котором хранится ссылка на вызвавший этот метод объект. Этот параметр применяется, например, когда требуется использовать имя поля, совпадающее с именем параметра метода.

Пример

```
class Primer3
{
    int x;
    public void set_x(int x)
    { this.x = x; } // полю x объекта класса Primer3 будет
                  // присвоено значение, переданное в метод set_x
                  // в качестве аргумента
}
```

3.4. Конструкторы

Конструктор объекта класса – это метод, предназначенный для инициализации объекта, автоматически вызываемый при создании объекта с помощью операции **new**. Имя конструктора должно совпадать с именем класса.

Описание:

```
[спецификатор] Имя_класса([параметры])  
    { тело конструктора}
```

Обычно используется спецификатор **public**.

Конструктор не возвращает значение.

Класс может иметь несколько конструкторов для инициализации объектов разными способами.

Если конструктор в классе не определен, C# автоматически предоставляет конструктор по умолчанию, который инициализирует все поля нулями. Но если в классе определить хотя бы один конструктор, конструктор по умолчанию не используется.

Для инициализации статических данных класса можно создать статический конструктор.

Статический конструктор вызывается автоматически до вызова конструктора объекта. Он должен быть закрытым.

Описание статического конструктора:

```
static Имя_класса()  
    { тело конструктора}
```

Приведенный на рис. 3.1 пример демонстрирует создание объектов класса с помощью различных конструкторов.

```

class Cilindr
{
    double radius_osnovania, vysota; // закрытые поля :
                                   // радиус основания и высота

    // Конструктор без параметров:
    public Cilindr()
    { radius_osnovania = 10; vysota = 10; }

    // Конструктор с параметрами:
    public Cilindr(double r, double h)
    { radius_osnovania = r; vysota = h; }

// методы получения значений радиуса и высоты
    public double get_radius()
        {return radius_osnovania; }
    public double get_vysota() { return vysota; }

    //метод для вывода полей класса
    public void vyvod()
    {
        Console.WriteLine("Радиус= {0,5:f3} Высота= {1,5:f3}",
                           radius_osnovania, vysota);
    }
}

// метод для вычисления площади боковой поверхности:
    public double Ploschad_bokovoy_pov()
    { return 2 * Math.PI * radius_osnovania * vysota; }

    // метод для вычисления объема цилиндра:
    public double Objem()
    { return Math.PI * Math.Pow(radius_osnovania, 2) *
        vysota; }

// методы установки значений радиуса и высоты
    public void set_radius(double r)
    { radius_osnovania = r; }
    public void set_vysota(double h)
    { vysota = h; }

```

Рис. 3.1. Пример использования конструкторов класса
(1-й фрагмент; продолжение и окончание см. на с. 30 и 31)

```

class Program
{
    static void Main(string[] args)
    {
        // создание объекта класса Cilindr (первого цилиндра)
        // с вызовом конструктора без параметров
        Cilindr C1=new Cilindr();

        //вывод параметров первого цилиндра:
        Console.WriteLine(
            "Площадь боковой поверхности первого цилиндра:{0,5:f3}",
                C1.Ploschad_bokovoy_pov());
        Console.WriteLine(
            "Объем первого цилиндра:{0,5:f3}",C1.Objem());
        Console.WriteLine("Радиус= {0,5:f3}  Высота= {1,5:f3}",
            C1.get radius(),C1.get vysota());

        // создание второго цилиндра с радиусом 20 и высотой 5
        // с использованием второго конструктора с параметрами
        Cilindr C2 = new Cilindr(20,5);

        // вывод параметров второго цилиндра
        Console.WriteLine(
            "\nПлощадь боковой поверхности второго цилиндра:{0,5:f3}",
                C2.Ploschad_bokovoy_pov());
        Console.WriteLine(
            "Объем второго цилиндра:{0,5:f3}", C2.Objem());

        // создание третьего цилиндра с использованием второго
        // конструктора
        // и вводом значений радиуса и высоты с клавиатуры:
        Console.WriteLine(
            "\nВведите радиус основания и высоту третьего цилиндра");
        Cilindr C3 = new Cilindr(Convert.ToDouble(Console.ReadLine()),
            Convert.ToDouble(Console.ReadLine()));
        // вывод радиуса и высоты третьего цилиндра с помощью метода
        // экземпляра C3:
        C3.vyvod();
    }
}

```

Рис. 3.1. Продолжение (начало см. на с. 29, окончание – на с. 31)

```

// сравнительный анализ объемов второго и третьего цилиндров:
    if (C2.Objem()>C3.Objem())
        Console.WriteLine(
"Объем второго цилиндра больше объема третьего цилиндра");
    else
        if (C2.Objem()<C3.Objem())
            Console.WriteLine(
"Объем третьего цилиндра больше объема второго цилиндра");
        else Console.WriteLine(
"Объемы второго и третьего цилиндра равны");

// изменение параметров третьего цилиндра
// с помощью методов установки значений полей
    C3.set_radius(10); C3.set_vysota(10);
    Console.WriteLine("Новые параметры третьего цилиндра:");
    C3.vyvod();
    Console.ReadKey();
    }
}

```

Рис. 3.1. Окончание (начало см. на с. 29 и 30)

3.5. Сбор мусора, деструкторы

Каждому объекту класса при создании выделяется память в динамической области памяти (*хипе*). В С# имеется система сбора мусора, которая автоматически возвращает память для повторного использования. Эта система действует незаметно для программиста, активизируется только по необходимости и точно невозможно узнать, когда происходит сбор мусора.

Деструктор – это специальный метод, который вызывается сборщиком мусора непосредственно перед удалением объекта из памяти. Деструктор не имеет параметров и не возвращает значение. Точно не известно, когда вызывается деструктор, но все деструкторы будут выполнены перед окончанием программы.

Синтаксис деструктора:

```

~ Имя_класса ( )
{ тело деструктора}

```

3.6. Свойства класса

Свойство – это элемент класса, предоставляющий доступ к его полям. Обычно связано с закрытым полем класса.

```
[спецификаторы] тип имя_свойства
{
    [get код аксессуора чтения поля]
    [set код аксессуора записи поля]
}
```

Оба аксессуора **не могут отсутствовать**.

Имя свойства можно использовать как обычную переменную в операторах присваивания и выражениях.

При обращении к свойству автоматически вызываются аксессуоры чтения и установки.

Обращение к свойству:

имя_объекта.имя_свойства

Аксессуор **get** должен содержать оператор **return**. Эта часть кода выполняется при использовании свойства в выражении и возвращает значение, указанное в операторе **return**.

В аксессуоре **set** используется стандартный параметр **value**, который содержит устанавливаемое для поля значение. Эта часть кода выполняется при использовании свойства в левой части оператора присваивания. Значение выражения, стоящего в правой части оператора присваивания, передается в аксессуор **set** через входной параметр **value**.

Например, в классе **Cilindr** из примера в параграфе 3.4 методы **set_radius** и **get_radius** можно заменить на свойство:

```
public double Radius
    { get { return radius_osnovania; }
      set { radius_osnovania = value; } }
```

Обращение к свойству может быть такое:

```
Console.WriteLine("Радиус= {0,5:f3} ", C1.Radius);
```

```
// В этом случае выполняется аксессуор get
```

```
C3.Radius = 100;
```

```
// В этом случае выполняется аксессуор set
```

Аксессуор **set** можно дополнить проверкой значения на положительность:

```
public double Radius
    {
        get { return radius_osnovania; }
        set { if (value>0) radius_osnovania = value; }
    }
```

Тема 4. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

4.1. Понятие исключения, основные стандартные исключения

Все исключения являются подклассами класса **Exception** пространства имен **System**. Исключения генерирует среда программирования или программист. В табл. 4.1 представлены наиболее часто используемые исключения.

Таблица 4.1

Стандартные исключения

Исключение	Значение
<code>ArrayTypeMismatchException</code>	Тип сохраняемого значения несовместим с типом массива
<code>DivideByZeroException</code>	Попытка деления на ноль
<code>IndexOutOfRangeException</code>	Индекс массива оказался вне диапазона
<code>OutOfMemoryException</code>	Обращение к оператору <code>new</code> оказалось неудачным из-за недостаточного объема свободной памяти
<code>OverflowException</code>	Имеет место арифметическое переполнение
<code>FormatException</code>	Попытка передать в метод аргумент неверного формата
<code>InvalidCastException</code>	Ошибка преобразования типа

Свойства класса `Exception`:

Message – текстовое описание ошибки;

TargetSite – метод, выбросивший исключение.

4.2. Оператор `try`

Исключения перехватываются и обрабатываются оператором `try`.

Пример

try

{контролируемый блок}

catch (тип1 [имя1]) *{ обработчик исключения1 }*

catch (тип2 [имя2]) *{ обработчик исключения2 }*

...

catch *{ обработчик исключения }*

finally *{ блок завершения }*

В контролируемый блок включаются операторы, выполнение которых может привести к ошибке.

С try-блоком можно связать не одну, а несколько catch-инструкций. Однако все catch-инструкции должны перехватывать исключения различного типа.

При возникновении ошибки при выполнении операторов контролируемого блока генерируется исключение.

Выполнение текущего блока прекращается, находится обработчик исключения соответствующего типа, которому и передается выполнение.

После выполнения обработчика выполняется блок **finally**. Блок **finally** будет выполнен после выхода из try/catch-блока, независимо от условий его выполнения.

Если подходящий обработчик не найден, вызывается стандартный обработчик, который обычно выводит сообщение и останавливает работу программы.

Форма обработчика

catch (тип) { обработчик исключения }

используется, если важен только тип исключения, а свойства исключения не используются.

Пример

```
try
    int y=a/b
    catch (DivideByZeroException)
    { Console.WriteLine(" Деление на 0"); }
finally
    {Console.ReadKey();}
```

Форма обработчика

catch (тип имя) { обработчик исключения }

используется, когда имя параметра используется в теле обработчика.

Пример

```
try
    int y=a/b
    catch (DivideByZeroException f)
    { Console.WriteLine(f.Message+": Деление на 0"); }
```

При попытке деления на 0 выведется сообщение
Attempted to divide by zero. Деление на 0
Форма обработчика

```
catch { обработчик исключения }
```

применяется для перехвата всех исключений, независимо от их типа.

Он может быть только один в операторе try и должен быть помещен после остальных catch-блоков.

Пример

```
try
```

```
{ int v = Convert.ToInt32(Console.ReadLine()); }
```

```
catch { Console.WriteLine("Ошибка!!!"); }
```

В этом примере и в случае ввода очень большого числа, и в случае ввода недопустимых в целых константах символов выводится сообщение "Ошибка!!!".

4.3. Генерирование исключений вручную

Исключение можно сгенерировать вручную, используя инструкцию **throw**.

Формат инструкции:

```
throw [параметр];
```

Параметр – это объект класса исключений, производного от класса Exception.

Пример

```
double x;
```

```
if (x == 0) throw new DivideByZeroException();
```

Исключение, перехваченное одной catch-инструкцией, можно регенерировать, чтобы обеспечить возможность его перехвата другой (внешней) catch-инструкцией. Чтобы повторно сгенерировать исключение, достаточно использовать ключевое слово **throw**, не указывая параметра (рис. 4.1):

```
throw ;
```

```

try
{
    try
    {
        int v = Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("v=" + v);

    }

    catch (FormatException)
        { Console.WriteLine("Неверный ввод"); throw; }
}
catch (FormatException)
    { Console.WriteLine("Это очень плохо"); }

```

Рис. 4.1. Пример генерирования исключений вручную

Один try-блок можно вложить в другой.

Исключение, сгенерированное во внутреннем try-блоке и не перехваченное catch-инструкцией, которая связана с этим try-блоком, передается во внешний try-блок. Часто внешний try-блок используют для перехвата самых серьезных ошибок, позволяя внутренним try-блокам обрабатывать менее опасные.

Тема 5. ИСПОЛЬЗОВАНИЕ МАССИВОВ В C#

5.1. Основные понятия

Массив – это структурированный тип данных, представляющий собой последовательность *однотипных* элементов, имеющих общее имя и снабженных индексами (порядковыми номерами). Нумерация начинается с нуля.

Массив – это ссылочный тип данных: в стеке размещается ссылка на массив, а сами элементы массива располагаются в динамической области памяти – *хипе*. Поэтому его нужно создавать с помощью операции `new`.

При создании всем элементам массива присваиваются по умолчанию нулевые значения.

Все массивы в C# построены на основе класса **Array** из пространства имен **System**, а значит наследуют некоторые его элементы или для них можно использовать его методы.

5.2. Одномерный массив

Одномерный массив можно описать одним из описанных ниже способов.

тип [] *имя_массива*;

Пример

double [] *y, z*;

В этом случае память под элементы массивов не выделена.

тип [] *имя_массива* = **new** *тип* [*размерность*];

Здесь *размерность* – выражение, тип которого имеет неявное преобразование к `int`, `long`, `ulong`, `uint`.

Пример

int [] *a* = **new int** [20], *b* = **new int** [100];

В этом случае в памяти создаются массивы из 20 и 100 элементов соответственно, всем элементам присваивается значение 0.

тип [] *имя_массива* = **new** *тип* [] { *список_инициализаторов* };

Пример

int [] *x* = **new int** [] { 2, -5, 0, 9 };

В этом случае *размерность* массива явно не указана и определяется по количеству элементов в списке инициализаторов.

тип [] *имя_массива* = { *список_инициализаторов* };

В этом случае **new** подразумевается.

тип [] *имя_массива* = **new** *тип* [*размерность*] { *список_инициализаторов* };

Пример

int [] *x* = **new int** [4] { 2, -5, 0, 9 };

В этом случае *размерность* массива все равно бы определилась, т. е. имеем избыточное описание.

Обращение к элементу массива:

имя массива [*индекс*]

Например, `x[3]`, `MyArray[10]`.

В приведенном ниже примере показан ввод массива с клавиатуры в консольном приложении.

Пример

```
Console.WriteLine("Введите количество элементов");  
int n=Convert.ToInt32(Console.ReadLine());
```

```

double[ ] x = new double[n];
for (int i = 0; i < n; ++i)
{
    Console.WriteLine("x[" + i + "]=");
    x[i] = Convert.ToDouble(Console.ReadLine());
}

```

Для просмотра всех элементов из некоторой группы данных (массива, списка и др.) существует удобный оператор цикла **foreach**.

Синтаксис:

```

foreach (тип имя_переменной in имя_массива)
    тело цикла;

```

Имя_переменной задает имя локальной переменной, которая будет по очереди принимать все значения из массива, имя которого указано в операторе после слова **in**. Ее тип должен соответствовать типу элементов массива.

С помощью оператора **foreach** нельзя изменять значение переменной цикла. Например, **нельзя** написать

```

foreach (double xt in x)
{
    xt = Convert.ToDouble(Console.ReadLine());
}

```

Ниже приведен фрагмент программы, демонстрирующий вывод одномерного массива с использованием оператора **foreach**.

Пример

```

foreach (double xt in x)
    Console.WriteLine(xt);

```

5.3. Свойства и методы класса **Array** для обработки одномерных массивов

Элементы класса **Array** для работы с одномерными массивами представлены в табл. 5.1–5.3.

Таблица 5.1

Свойства класса **System.Array**

Свойство	Описание
Length	Количество элементов массива
Rank	Количество размерностей массива

Статические методы класса **System.Array**

Метод	Описание
Clear(x, j, n)	Присваивает <i>n</i> элементам массива <i>x</i> , начиная с <i>j</i> -го, значения по умолчанию. Например: Array.Clear(x, 1, 2);
BinarySearch(x, xx)	Ищет в отсортированном массиве <i>x</i> номер элемента со значением <i>xx</i> . Например, Array.BinarySearch(a, 9)
Sort(x)	Упорядочивает массив <i>x</i> в порядке возрастания значений элементов
Sort(x, j, n)	Упорядочивает часть массива <i>x</i> из <i>n</i> элементов, начиная с <i>j</i> -го
Reverse(x)	Изменяет порядок следования элементов массива <i>x</i> на обратный
Reverse(x, j, n)	Изменяет порядок следования <i>n</i> элементов массива <i>x</i> на обратный, начиная с <i>j</i> -го элемента
Copy(x, z, n)	Копирует <i>n</i> элементов массива <i>x</i> в массив <i>z</i> (<i>n</i> должно быть не больше размерности <i>z</i> и <i>x</i>)
Copy(x, j, z, k, n)	Копирует <i>n</i> элементов массива <i>x</i> , начиная с <i>j</i> -го, в массив <i>z</i> с <i>k</i> -й позиции
IndexOf(x, xx)	Ищет в массиве <i>x</i> номер первого элемента со значением <i>xx</i>
LastIndexOf(x, xx)	Ищет в массиве <i>x</i> номер последнего элемента со значением <i>xx</i>

Нестатические методы класса **System.Array**

Метод	Описание
CopyTo(x, j)	Копирование всех элементов массива, для которого вызван метод, в массив <i>x</i> , начиная с <i>j</i> -го элемента. Например, x.CopyTo(z, 1);
GetValue(j)	Получение значения элемента массива с номером <i>j</i> (a.GetValue(3))
SetValue(xx, j)	Установка значения <i>xx</i> для <i>j</i> -го элемента массива z.SetValue(8, 1);

5.4. Многомерные массивы

5.4.1. Виды многомерных массивов

Многомерным называется массив, который характеризуется двумя или более измерениями, а доступ к отдельному элементу осуществляется посредством двух или более индексов. Существуют два типа многомерных массивов: *прямоугольные* и *ступенчатые* (*разреженные, рваные, зубчатые*).

Многомерный прямоугольный массив объявляется следующим образом:

```
тип[ , ... , ] имя = new тип[размер_1, ..., размер_N] ;
```

Пример

```
int [ , ] y = new int [ 4 , 3, 3];
```

Так создается трехмерный целочисленный массив размером 4 x 3 x 3.

5.4.2. Двумерные прямоугольные массивы

Двумерный массив можно описать одним из следующих способов.

1. **тип**[,] *имя_массива*;

Пример

```
double[ , ] y, z;
```

В этом случае память под элементы массивов не выделена.

2. **тип**[,] *имя_массива* = **new тип**[*разм_1*, *разм_2*];

Пример

```
int[ , ] a = new int[5,5], b = new int[10,4];
```

В этом случае в памяти создаются массивы из 25 и 40 элементов соответственно, всем элементам присваивается значение 0.

3. **тип**[,] *имя_массива* = **new тип**[,] {*список_инициализаторов*};

В списке инициализаторов значения сгруппированы в фигурных скобках по строкам.

Пример

```
int[,] x = new int[,] {{2, -5, 0, 9},  
                      {3, 2, -5, 5},  
                      {2, 4, 6, -1}};
```

В этом случае размерность массива явно не указана и определяется по количеству элементов в списке инициализаторов.

4. `тип[,] имя_массива = {список_инициализаторов};`
Операция **new** подразумевается.

5. `тип[,] имя_массива = new тип[разм1, разм2]`
`{список_инициализаторов};`

Пример

```
int[, ] x = new int[2,2] {{2, -5}, { 0, 9}};
```

В этом случае размерность массива все равно бы определилась, т. е. имеем избыточное описание.

Обращение к элементу матрицы:

имя_массива [индекс1, индекс2]

Например, `x[3,4]`, `MyArray[1,0]`.

```
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < m; ++j)
        Console.WriteLine("{0, 5:f2}", x[i, j]);
    Console.WriteLine();
}
```

Рис. 5.1. Фрагмент кода, демонстрирующий вывод матрицы на экран в консольном режиме

Для просмотра двумерных прямоугольных массивов также можно применять оператор **foreach**. Повторение оператора **foreach** начинается с элемента, все индексы которого равны нулю, и повторяется через все возможные комбинации индексов с приращением крайнего правого индекса первым. Когда правый индекс достигает верхней границы, он становится равным нулю, а индекс слева от него увеличивается на единицу.

Например, пусть требуется найти максимальный элемент матрицы.

```
double max = x[0, 0];
foreach (double xt in x)
    if (xt > max) max = xt;
Console.WriteLine("Максимум: "+max);
```

В табл. 5.4 приводятся нестатические методы класса **Array** для работы с двумерными массивами.

Нестатические методы класса `Array` для работы с двумерными массивами

Метод	Описание
<code>GetLength(n)</code>	Возвращает длину заданной размерности. Например, если размерность матрицы $A 3 \times 5$, то <code>A.GetLength(1)</code> будет равно 5
<code>GetValue(i, j)</code>	Возвращает значение элемента вызывающего массива с индексами $[i, j]$.
<code>SetValue(xx, i, j)</code>	Устанавливает в вызывающем массиве элемент с индексами $[i, j]$, равным значению <code>xx</code>

5.4.3. Ступенчатые массивы

Ступенчатые массивы – это массивы, в которых количество элементов в разных строках может быть различным. Поэтому такой массив можно использовать для создания таблицы со строками разной длины.

Представляет собой массив массивов, в памяти хранится в виде нескольких внутренних массивов, каждый из которых имеет свою длину, а для хранения ссылки на каждый из них выделяется отдельная область памяти.

Описание:

```
тип [ ] [ ] имя = new тип[размер] [ ];
```

Например, `int [] [] z = new [10] [];`

Здесь *размер* означает количество строк в массиве.

Для самих строк память выделяется индивидуально. Под каждый из внутренних массивов память требуется выделять явным образом.

Пример

```
int [ ] [ ] x = new int [ 3 ] [ ] ;
    x[0] = new int [ 4 ] ;
    x[1] = new int [ 3 ] ;
    x[2] = new int [ 5 ] ;
```

В данном случае `x.Length` равно 3, `x[0].Length` равно 4, `x[1].Length` равно 3, `x[2].Length` равно 5.

Другой способ:

```
тип [ ] [ ] имя = {создание_массива1, создание_массива2, ..., создание_массиваN};
```

Пример

```
int [ ] [ ] x = { new int [ 4 ], new int [ 3 ], new int [ 5 ] };
```

Доступ к элементу осуществляется посредством задания каждого индекса внутри своих квадратных скобок:

имя[индекс1] [индекс2]

Например, **x[2] [9]** или **a[i] [j]**

Пример

Определить средний балл в группах студентов, вывести списки двоечников в каждой группе, назначить студентам стипендию, которой они достойны.

Консольное приложение для решения данной задачи приведено ниже.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Student
    {
        int[ ] ocenki; // закрытые поля класса: массив оценок,
        string fam, grupa; // фамилия, группа,
        static double mrot; // минимальны размер оплаты труда.

        public Student(string fam, string grupa, int n) // конструктор
        { this.fam = fam; this.grupa = grupa;
          ocenki = new int[n]; }

        public string Fam // свойство доступа к полю Фамилия
        {
            set { fam = value; }
            get { return fam; }
        }
        public string Grupa // свойство доступа к полю Группа
        {
            set { grupa = value; }
            get { return grupa; }
        }
        static Student() // статический конструктор класса
        { mrot = 200000; }
```

```

public void vvod_oc() //метод для вывода оценок
{
    Console.WriteLine("Введите оценки студента " +
                      fam + " за сессию");
    for (int i = 0; i < ocenki.Length; ++i )
        {ocenki[i]= int.Parse(Console.ReadLine()); }
}

public double Sr_b // свойство только для чтения,
                  // возвращающее средний балл студента
{
    get { double S = 0;
        foreach (int x in ocenki) S = S + x;
        return S / ocenki.Length;}
}

public double stip // свойство только для чтения,
                  // возвращающее размер стипендии
{
    get { if (Dvoechnik) return 0;
        else return
            (Sr_b > 8) ? (2 * mrot) :
            (Sr_b > 6 ? 1.8 * mrot : 1.25 * mrot);
        }
}

public bool Dvoechnik //свойство только для чтения,
                    // возвращающее true, если у студента есть двойки
{ get
    {if (Array.IndexOf(ocenki, 2)>=0)
        return true;
    else
        return false;
    } } }

class Program
{
    static void Main(string[ ] args)
    { Console.WriteLine("Сколько групп?");
      int m=int.Parse(Console.ReadLine());
    // создание ступенчатого массива объектов класса Student

```

```

Student[ ][ ] fakultet = new Student[m][ ];
for (int i = 0; i < m; ++i)
{
    Console.WriteLine("Какая группа?");
    string grp = Console.ReadLine();
    Console.WriteLine("Сколько в группе студентов ?");
    int ks = Convert.ToInt32(Console.ReadLine());
    fakultet[i]=new Student[ks];
    for (int j = 0; j < ks; ++j)
        { Console.WriteLine("Фамилия студента ? ");
fakultet[i][j] = new Student(Console.ReadLine( ), grp, 4);
        fakultet[i][j].vvod_oc( );
        }
    }
for (int i = 0; i < m; ++i)
{ double Sb = 0;
    foreach (Student xs in fakultet[i])
        { Sb = Sb + xs.Sr_b; }

    Console.WriteLine(fakultet[i][0].Gruppa +
        " "+Sb/fakultet[i].Length);
}
for (int i = 0; i < m; ++i)
{
    Console.WriteLine("\n Двоечники группы "
        + fakultet[i][0].Gruppa);
    foreach (Student xs in fakultet[i])
        {
            if (xs.Dvoechnik)
                Console.WriteLine(xs.Fam);
        }
}
foreach (Student[ ] xss in fakultet)
{
    Console.WriteLine(
        "\nСтипендия студентов группы "+ xss[0].Gruppa);
    foreach (Student xs in xss)
        {
            Console.WriteLine(xs.Fam+" "+xs.stip);
        }
}

```

```

    }
    }
    } Console.ReadKey();
}
}
}

```

Тема 6. ИНДЕКСАТОРЫ И ОПЕРАЦИИ КЛАССА

6.1. Одномерные индексы

Если в классе есть скрытое поле, представляющее собой набор элементов (например, массив), то в нем можно определить индексатор, обеспечивающий индексированный доступ к элементам поля с использованием имени объекта и номера элемента в квадратных скобках.

Индексаторы могут характеризоваться одной или несколькими размерностями.

Одномерные индексаторы описываются следующим образом:

```

спецификатор тип this [тип_индекса индекс]
{
    get {код аксессуара для получения данных}
    set {код аксессуара для установки данных}
}

```

Спецификаторы аналогичны спецификаторам свойств и методов. Нельзя использовать **static**. Чаще всего используют **public**.

Тип – это тип элемента, к которому предоставляется доступ посредством индексатора. Обычно он соответствует базовому типу элементов индексированного поля, например, типу элементов массива. Но индексаторы могут возвращать значение другого типа, отличающегося от типа данных в списке элементов.

Тип_индекса не обязательно должен быть **int**, но поскольку индексаторы обычно используются для обеспечения индексации массивов, целочисленный тип – наиболее подходящий. Но в качестве индекса можно использовать даже строки.

При использовании индексатора аксессоры вызываются автоматически, и в качестве параметра оба аксессуара принимают **индекс**. Если индексатор стоит слева от оператора присваивания, вызывается аксессуар **set** и устанавливается элемент, заданный параметром **индекс**. В противном случае вызывается аксессуар **get** и возвращается значение, соответствующее параметру **индекс**.

Например, пусть в классе `Massiv` определены закрытое поле

```
double [ ] x;
```

и индексатор

```
public double this [int i]  
{ get { return x[i];}  
  set { x[i]=value;}  
}
```

И пусть Y – объект класса `Massiv`.

Тогда к элементам поля x объекта Y можно обращаться следующим образом:

```
Y[1] = 5.3;
```

```
Console.WriteLine(Y[i]);
```

Одно из достоинств индексатора состоит в том, что он позволяет точно управлять характером доступа к массиву, предотвращая попытки некорректного доступа.

```
public double this[int i]  
{  
  get  
  {  
    if (i >= 0 && i < a.Length) return a[i];  
    else throw new Exception("Плохой индекс " + i);  
  }  
  set  
  {  
    if (i >= 0 && i < a.Length) a[i] = value;  
    else throw new Exception("Плохой индекс " + i);  
  }  
}
```

Индексатор может быть совсем не связан с каким либо полем класса:

```
class Pow2  
{  
  public ulong this[int i]  
  {  
    get  
    { if (i >= 0 && i <= 100) return (ulong) Math.Pow(2, i);  
      else throw new Exception("Ошибка!!!");  
    }  
  }  
}
```

```

    }
}
class Program
{
    static void Main(string[] args)
    {
        Pow2 pow2 = new Pow2();
        for (int i = 5; i <= 10; ++i)
            Console.WriteLine("2 в степени "+i+" = "+pow2[i]);
        Console.ReadKey();
    }
}

```

6.2. Многомерные индексы

Многомерные индексы описываются следующим образом:

спецификатор min this [$min1$ индекс1, $min2$ индекс2, ..., $minN$ индексN]

```

{
    get {код аксессуара для получения данных}
    set {код аксессуара для установки данных}
}

```

Например, если в классе объявлен массив

```
int[ , ] x;
```

то простейший индекс, обеспечивающий доступ к массиву x , может выглядеть так:

```

public int this[int i, int j]
{
    get {return x[i, j];}
    set {x[i, j] = value;}
}

```

6.3. Операции класса

В C# операции определены для стандартных типов данных. Язык C# позволяет определить значение операции для объектов созданного программистом класса. Такое создание новой операции со стандартным именем называется перегрузкой операций.

Например, стандартная операция сложения определена для типов **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal**. Можно создать еще одну

\\ ссылается операнд.

```
public static Tochka operator -- (Tochka tchk)
{
    tchk.x = tchk.x - 1; tchk.y = tchk.y - 1;
    return tchk;
}
}
```

Тогда для объектов класса **Tochka** можно использовать перегруженные в классе операции:

```
Tochka t1 = new Tochka(2, 8);
Console.WriteLine((-t1).X + ", " + (-t1).Y);
Tochka t2 = -t1;
Console.WriteLine(t1.X + ", " + t1.Y);
--t1;
Console.WriteLine(t1.X + ", " + t1.Y);
```

Префиксный и постфиксный инкременты и декременты не различаются.

Перегруженные версии ключевых слов **true** и **false** позволяют по новому определить понятия **ИСТИНА** и **ЛОЖЬ** в отношении создаваемых классов.

Например, можно определить, что для объектов класса **Gruppa** истина, если в группе больше 5 человек, т. е. объект этого класса, в котором поле, содержащее количество студентов меньше 6, является ложным (иначе, имеет значение **false**).

Операции **true** и **false** должны быть определены в паре. Нельзя перегружать только одну из них. При этом тип результата обычно **bool**.

Формат этих операций:

```
public static bool operator true (тип_операнда операнд)
{
    тело операции с возвратом значения true или false.
}
}
```

```
public static bool operator false (тип_операнда операнд)
{
    тело операции с возвратом значения true или false.
}
}
```

Например, определим класс **Treugolnic**, объект которого считается истинным, если он существует:

```

class Treugolnic
{
    double a, b, c; \\\стороны треугольника
public Treugolnic(double a1, double b1, double c1)
    { a = a1; b = b1; c = c1; }
//Операция true
public static bool operator true(Treugolnic p)
    {
        if (p.a + p.b > p.c && p.b + p.c > p.a && p.a + p.c > p.b)
            return true;
        else return false;
    }
//Операция false
public static bool operator false(Treugolnic p)
    {
        if (p.a + p.b > p.c && p.b + p.c > p.a && p.a + p.c > p.b)
            return false;
        else return true;
    } }

```

Тогда, если **T1** – объект класса **Treugolnic**, допустим такой оператор:

```
if (T1) Console.WriteLine(" Треугольник существует");
```

При этом будет вызвана операция **true**.

Перегруженная операция **!** также, как правило, возвращает результат типа **bool**.

Дополним класс **Treugolnic** операцией **!**:

```

public static bool operator !(Treugolnic p)
    {
        if (p) return false;
        else return true;
    }

```

Тогда в методе Main ее можно использовать следующим образом: `bool t;`
`t = !T1;`

6.3.2. Бинарные операции

В классе можно определять следующие бинарные операции:

```

+   -   *   /   %
&   |   ^   <<  >>
==  !=  >   <   >=  <=

```

При соблюдении определенных правил можно использовать операторы `&&` и `||`, действующие по сокращенной схеме: если значения первого операнда достаточно, чтобы определить результат операции, второй операнд не вычисляется.

Синтаксис операции:

```
public static тип_результата operator знак_операции  
    (тип_операнда1 операнд1, тип_операнда2 операнд2)  
    {тело_операции}
```

Тип хотя бы одного из операндов должен совпадать с классом, для которого определена операция.

Операции присваивания (включая составные, например, "+=") перегружать нельзя.

Операции, перегрузка которых также запрещена:

```
[] ( ) new is sizeof typeof ?
```

Пример

Создадим класс **Massiv** с операциями сложения массивов и операцией сложения массива с числом.

```
class Massiv  
{  
    double[] a;  
    public Massiv(int n)  
    {  
        a = new double[n];  
    }  
    public void vyvod(string zagolovok)  
    {  
        Console.WriteLine(zagolovok);  
        foreach (double x in a)  
            Console.WriteLine(x);  
    }  
    public double this[int i]  
    {  
        get  
        {  
            if (i >= 0 && i < a.Length) return a[i];  
            else throw new Exception("Плохой индекс " + i);  
        }  
        set  
        {
```

```

        if (i >= 0 && i < a.Length) a[i] = value;
        else throw new Exception("Плохой индекс " + i);
    }
}
public int Length
    { get { return a.Length; } }

public static Massiv operator +(Massiv a1, Massiv a2)
    {
        Massiv rez = new Massiv(a1.Length+a2.Length);
        for (int i = 0; i < a1.Length; ++i) rez[i] = a1[i];
        for (int i = 0; i < a2.Length; ++i) rez[a1.Length+i] = a2[i];
        return rez;
    }
public static Massiv operator +(Massiv a1, int a2)
    {
        Massiv rez = new Massiv(a1.Length);
        for (int i = 0; i < a1.Length; ++i) rez[i] = a1[i] + a2;
        return rez;
    }
public static Massiv operator +(int a2, Massiv a1)
    {
        Massiv rez = new Massiv(a1.Length);
        for (int i = 0; i < a1.Length; ++i) rez[i] = a1[i] + a2;
        return rez;
    } }

```

Тогда, если A , B – объекты класса **Massiv**, допустимы следующие операторы:

Massiv C = A + B; C.vyvod("A+B");

Massiv Z = A + 2; Z.vyvod("A+2");

Massiv Z1 = 3 + B; Z1.vyvod("3+B");

Операции отношения следует перегружать парами. Например, определяя в классе операцию "<", нужно определить операцию ">", и наоборот.

Пары операций отношения:

<= >=

< >

== !=

Перегружая операции `==` и `!=`, следует перегрузить также методы `Object.Equals()` и `Object.GetHashCode()`.

Обычно операции отношения возвращают логическое значение.

Например, дополним класс `Massiv` операциями `<` и `>`:

```
public static bool operator <(Massiv a1, Massiv a2)
{
    if (a1.Length < a2.Length) return true;
    else return false;
}
public static bool operator >(Massiv a1, Massiv a2)
{
    if (a1.Length < a2.Length) return true;
    else return false;
}
```

Использовать можно так:

```
if (A > B) Console.WriteLine("A больше B");
```

Если при определении в классе логических операций не планируется использование логических операций, работающих по сокращенной схеме, можно перегружать операторы `&` и `|` по своему усмотрению.

Эти операции могут возвращать результат типа `bool`.

Чтобы иметь возможность использовать операции `&&` и `||`, необходимо соблюдать следующие правила:

- в классе должны быть определены операции `true` и `false`;
- класс должен перегружать операции `&` и `|`, которые должны возвращать объект этого класса.
- каждый параметр должен представлять собой ссылку на объект класса, в котором определены операции.

6.3.3. Операции преобразования типа

Эти операции преобразуют объект некоторого класса в значение другого типа. Фактически операция преобразования перегружает операцию приведения типов.

Существуют две формы операторов преобразования: явная и неявная.

Явная форма:

```
public static explicit operator тип_результата
    (исходный_тип параметр)
{return значение;}
```

Операция выполняет преобразование из типа параметра в тип результата.

Одним из этих типов должен быть класс, для которого определяется операция.

Нельзя определять преобразование к типу object и наоборот.

Явное преобразование выполняется при использовании операции приведения типа.

Например, определим в классе **Student** операцию преобразования студента (т. е. объекта класса) к строковому типу:

```
public static explicit operator string(Student s)
    { return s.fam; }
```

Применение:

```
string f = (string) st1; Console.WriteLine( f );
Console.WriteLine((string) st2);
```

Определим также в классе **Student** операцию преобразования вещественного числа в объекта класса с соответствующим средним баллом:

```
public static explicit operator Student(double sb)
    { return new Student("", sb); }
```

Применение:

```
Student st3 = (Student) 7.5; st3.Fam = "Иванов";
```

Неявная форма:

```
public static implicit operator тип_результата
                               (исходный_тип параметр)
    {return значение;}
```

Преобразование выполняется автоматически в следующих случаях:

- 1) при присваивании объекта переменной, тип которой совпадает с типом результата;
- 2) при использовании объекта в выражении, содержащем переменные, тип которых совпадает с типом результата;
- 3) при передаче объекта в метод на место параметра с типом результата;
- 4) при явном приведении типа.

Для одной и той же пары типов, участвующих в преобразовании, **нельзя** определить одновременно обе формы операции преобразования.

Например, дополним класс **Student** операцией преобразования строки в студента в неявной форме.

```
public static implicit operator Student(string ff)  
    { return new Student(ff, 0); }
```

Применение:

```
Student st4 = "Коньков";  
Console.WriteLine("Средний балл студента с фамилией " +  
    st4.Fam + " :" + st4.Sb);
```

Тема 7. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ МЕТОДОВ

7.1. Методы с переменным количеством аргументов

Чтобы метод мог принимать произвольное число аргументов, необходимо в списке параметров использовать параметр-массив неопределенной длины, помеченный ключевым словом **params**.

Этот параметр может быть только один и должен размещаться в списке последним.

Например:

```
public int min(int x, int y, params int[] z) { }
```

В этот метод могут быть переданы два и более аргументов.

Параметр с ключевым словом **params** может принять любое количество аргументов, даже нулевое.

Например, определим в классе **Program** метод, вычисляющий сумму нескольких чисел, первое из которых целое, а остальные вещественные:

```
class Program  
    { public static double sum(int x, params double[] y)  
        { double s = x;  
            foreach (double yy in y) s = s + yy;  
            return s;  
        }  
    }  
static void Main(string[] args)  
    {  
        Console.WriteLine("3+5=" + sum(3, 5));  
        Console.WriteLine("3+5+2.5=" + sum(3, 5,2.5));  
        double[] z= {2,4,6};  
        Console.WriteLine("3+ сумма элементов z=" + sum(3,z));  
    }
```

```
        Console.ReadKey();
    }
```

В методе список параметров может состоять только из params-параметра. В этом случае нужно предусматривать возможность вызова метода без параметров, чтобы не возникали ошибки.

Например, определим метод для нахождения максимального из заданных вещественных чисел:

```
public static double  max(params double[] y)
{
    if (y.Length == 0)
        { Console.WriteLine("Нет аргументов "); return 0; }
    else
    {
        double s = y[0];
        foreach (double yy in y) if (yy >= s) s = yy;
        return s;
    }
}
static void Main(string[] args)
{
    Console.WriteLine("max(2,8)= " + max(2, 8));
    Console.WriteLine(" max(3,5,2.5)= " + max(3, 5, 2.5));
    double[] z= {2,4,6,10};
    Console.WriteLine("максимум в z= " + max(z));
    Console.WriteLine("max( )= " + max( ));
    Console.ReadKey();
}
```

7.2. Перегрузка методов

Определение нескольких методов с одним и тем же именем, но различными параметрами называется *перегрузкой методов*, а сами методы – *перегруженными* (overloaded).

Все перегруженные методы должны иметь списки параметров, которые отличаются по типу и/или количеству, а также по способу передачи.

Перегруженные методы *могут* отличаться и типами возвращаемых значений.

Компилятор определяет, какой метод требуется вызвать по типу и количеству фактических параметров, т. е. осуществляет *разрешение перегрузки*.

При разрешении не учитываются тип возвращаемого значения и параметр с модификатором **params**.

При вызове перегруженного метода компилятор выбирает вариант метода, соответствующий типу и количеству передаваемых в метод аргументов.

Если *точное соответствие не найдено*, выполняется неявное преобразование типов в соответствии с общими правилами. Если преобразование невозможно, возникает ошибка.

Если существует несколько подходящих вариантов метода, выбирается лучший из них: содержащий меньшее количество и длину преобразований.

Если выбрать лучший не удастся, выдается сообщение об ошибке.

Пример

Определим в классе Student перегруженные методы vvod и rez.

```
class Student
{
    int[] ocenki ;
    string fam;
    public double Sr_b
    {
        get {
            double s = 0;
            foreach (int x in ocenki) s = s + x;
            return s / ocenki.Length;
        }
    }
    public void vvod()
    {
        Console.WriteLine("Фамилия ?");
        fam = Console.ReadLine();
    }
    public void vvod(int n)
    {
        Console.WriteLine("введите "+n+" оценок");
        ocenki = new int[n];
        for (int i = 0; i < n; i++)
            ocenki[i]=Convert.ToInt32(Console.ReadLine());
    }
    public void rez()
}
```

```

        {
            Console.WriteLine("Успеваемость студента " + fam);
            foreach(int x in ocenki) Console.Write(x+" ");
            Console.WriteLine();
        }
public void rez(int k)
    {
        Console.Write("Количество оценок " +k+" студента "+ fam+ ":
");
        int m = 0;
        foreach (int x in ocenki) if (x == k) m = m + 1;
        Console.WriteLine(m);
    }
public void rez(int k,out int m)
    {
        Console.Write("Количество оценок ниже " + k + " сту-
дента " + fam + ": ");
        m = 0;
        foreach (int x in ocenki) if (x <k) m = m + 1;
        Console.WriteLine(m);
    }
public void rez(Student st)
    {
        if (Sr_b>st.Sr_b)
            Console.WriteLine(fam+" учится лучше чем "+st.fam);
        else Console.WriteLine(st.fam + " учится лучше чем " +
fam);
    }
}
}

```

В методе Main :

```

Student st1 = new Student();
st1.vvod( ); st1.vvod(5);
st1.rez( );
st1.rez(9);
int mm; st1.rez(4, out mm);
if (mm > 0) Console.WriteLine("Этот студент – двоечник" );
Student st2 = new Student( ); st2.vvod( ); st2.vvod(4);
st1.rez(st2);

```

Тема 8. СТРОКИ, СИМВОЛЫ И РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

8.1. Символы

Тип **char** – это 16-разрядный тип данных без знака.

Для представления символов в C# используется Unicode, стандарт кодировки символов, позволяющий представлять алфавиты всех существующих в мире языков, а также любые знаки, используемые в мире.

Стандартный набор символов ASCII составляет подмножество Unicode.

Символьная константа может быть записана в одной из четырех форм:

- символ, имеющий графическое представление (кроме апострофа): **'F' '+' 'ы'**
- управляющая последовательность: **'\n' '\'**;
- символ в виде escape – последовательности Unicode: **'\u0021'**;
- символ, представленный шестнадцатеричным кодом: **'\x5B'**.

В стандарте Unicode все символы разделены на категории: **DecimalNumber**, **LetterNumber** и др.

Символьный тип соответствует стандартному классу **Char** библиотеки **.NET** из пространства имен **System**.

Статические методы класса **Char** представлены в табл. 8.1.

Таблица 8.1

Статические методы класса Char

Метод	Описание
GetNumericValue(ch)	Возвращает числовое значение символа ch , если он цифра, и -1 в противном случае Возвращает результат типа double . Например: Char.GetNumericValue('B');
GetUnicodeCategory(ch)	Возвращает Unicode - категорию символа ch Например: Char.GetUnicodeCategory('B') возвращает UppercaseLetter
Parse(s)	Преобразует строку s , состоящую из одного символа, в символ Например: Char.Parse("B")
IsControl(ch)	Возвращает true , если символ ch является управляющим Например: Char.IsControl('\n')

Метод	Описание
IsDigit(ch)	Возвращает true , если символ ch является десятичной цифрой. Например: Char.IsDigit('\n')
IsLetter(ch)	Возвращает true , если символ ch является буквой. Например: Char.IsLetter('F')
IsLetterOrDigit(ch)	Возвращает true , если символ ch является буквой или цифрой
IsLower(ch)	Возвращает true , если символ ch задан в нижнем регистре
IsNumber(ch)	Возвращает true , если символ ch является числом (входит в одну из категорий: <code>DecimalDigitNumber</code> , <code>LetterNumber</code> , или <code>OtherNumber</code>) Например: Char.IsNumber('\u0660')
IsPunctuation(ch)	Возвращает true , если символ ch является знаком препинания
IsSeparator(ch)	Возвращает true , если символ ch является разделителем
IsUpper(ch)	Возвращает true , если символ ch задан в верхнем регистре
IsWhiteSpace(ch)	Возвращает true , если символ ch является пробелом, символом перевода строки, возврата каретки, табуляции и др.
ToUpper(ch)	Преобразует символ ch в верхний регистр
ToLower(ch)	Преобразует символ ch в нижний регистр
ToString(ch)	Преобразует символ ch к строковому типу

Свойства:

MaxValue, **MinValue** – возвращают символы с максимальным и минимальным кодами (не имеют видимого представления).

Для того чтобы узнать код символа, можно использовать явное преобразование к целому типу (**int**) **a**.

8.2. Строки типа string

8.2.1. Создание строки

Типу **string** соответствует базовый класс **System.String** библиотеки **.NET**, предназначенный для работы со строками символов в кодировке **Unicode**.

В классе **String** определено несколько конструкторов, которые позволяют создавать строки различными способами.

Строковый литерал создает строковый объект автоматически. Поэтому строковый объект часто инициализируется присваиванием ему строковой константы.

```
string <имя> = <строковая константа>;
```

Например:

```
string S = "Сегодня кина не будет!!!";
```

Еще один способ:

```
string <имя> = new string(<символ>,  
                        <количество повторений символа>);
```

Например: **string S1 = new string('ы', 10);**

Создать строку можно из символьного массива:

```
string <имя>=new string(<массив символов>);
```

Например,

```
char[ ] h = { 'B','a','a','y','!' };
```

```
string S2 = new string(h);
```

Еще один способ:

```
string <имя>=new string(<массив символов>,  
                        <нач. индекс>,<количество символов>);
```

Создаваемая таким образом строка будет состоять из *заданного количества* символов, взятых из массива, начиная с символа, индекс которого задан вторым параметром.

Например, **string S3 = new string(h, 1, 3);**

Обратиться к символу строки можно по индексу, как к элементу массива, нумерация начинается с нуля:

S[2], S3[j].

Но так как в классе **String** индексатор определен только для чтения, изменять элементы строки используя обращение по индексу **нельзя**:

S3[0] = 'd'; – это неправильно!!!

Для определения количества символов в строке используется свойство **Length** класса **String**.

Например: **S3.Length** – длина строки S3.

8.2.2. Конкатенация строк

Можно для этого использовать оператор "+". Например, **S1+"Вася"**. Или один из вариантов статического метода **Concat()**.

Параметрами этого метода могут быть либо строки, либо данные типа **object**, причем метод принимает произвольное число аргументов. Если аргумент имеет тип **object**, то в конкатенации участвует текстовое представление объекта (результат метода **ToString()**).

Например, **SS = string.Concat(10, 20.5, "Привет");**

8.2.3. Сравнение строк

В классе **String** реализована перегрузка двух операторов:

== и **!=**.

При использовании этих операций учитывается регистр и пробелы в начале и в конце строки.

Пример

```
string ss1 = "вася"; string ss2 = "  вася";  
if (ss1 == ss2) Console.WriteLine(ss1+" равно "+ss2);  
else Console.WriteLine(ss1 + " не равно " + ss2);
```

Результат: вася не равно вася

В классе **String** есть ряд методов сравнения строк.

Статический метод Compare(s1, s2) производит лингвистическое сравнение.

Возвращает положительное целое число, если строка s1 больше s2, отрицательное число, если s1 меньше s2, и нуль, если строки s1 и s2 равны. Тип результата **int**. Например,

```
ss1 = "вася "; ss2 = "ВАСЯ";  
Console.WriteLine(string.Compare(ss1, ss2));
```

Результат 1

```
ss1 = "вася"; ss2 = "ВАСЯ";  
Console.WriteLine(string.Compare(ss1, ss2));
```

Результат -1

Compare(s1, s2, u)

Если логический параметр *u* равен значению **true**, при сравнении не учитываются различия между прописным и строчным вариан-

тами букв. В противном случае эти различия учитываются. В остальном работает так же, как и предыдущий вариант:

```
ss1 = "вася"; ss2 = "ВАСЯ";  
Console.WriteLine(string.Compare(ss1, ss2,true));
```

Результат 0

Compare(s1, i1, s2, i2, n, u) сравнивает части строк s1 и s2 из n символов, начиная со строковых элементов s1[i1] и s2[i2].

Метод возвращает положительное число, если часть строки s1 больше части строки s2, отрицательное число, если часть строки s1 меньше части строки s2, и нуль, если сравниваемые части строк равны. Назначение параметра u такое же, как в предыдущем варианте метода.

Статический метод **CompareOrdinal(s1, s2)** сравнивает строку s1 со строкой s2 независимо от языка, диалекта или территориального образования.

Возвращает положительное целое число, если строка s1 больше s2, отрицательное число, если s1 меньше s2, и нуль, если строки s1 и s2 равны. Тип результата int. Длина не учитывается.

Сравнение осуществляется по числовым кодам символов, входящих в каждую строку. Поэтому 'f' > 'F'.

CompareOrdinal(s1, i1, s2, i2, n) сравнивает части строк s1 и s2 из n символов, начиная со строковых элементов s1[i1] и s2[i2].

Нестатический метод **CompareTo(s)** возвращает положительное целое число, если вызывающая строка больше строки s, отрицательное число, если вызывающая строка меньше s, и нуль, если строки равны. Тип результата int:

```
ss1 = " вася"; ss2 = "ВАСЯ";  
Console.WriteLine(ss1.CompareTo(ss2));
```

Результат -1

```
ss1 = "вася "; ss2 = "ВАСЯ";  
Console.WriteLine(ss1.CompareTo(ss2));
```

Результат 1

Нестатический метод **Equals(s)** возвращает true, если вызывающая строка равна s.

Например, **ss1.Equals(ss2)**.

Статический метод **Equals(s1,s2)** возвращает true, если строка s1 равна s2.

Например, **string.Equals(ss1,ss2)**.

8.2.4. Поиск в строке

Нестатический метод **IndexOf(s)** возвращает индекс первого вхождения *символа* или *строки s* в вызывающей строке. Если символа нет, возвращает -1 .

Пусть `s = "ABCD"`;

Результат `ss1.IndexOf('B')` 1

Результат `ss1.IndexOf("CD")` 2

Нестатический метод **LastIndexOf(s)** возвращает индекс последнего вхождения *символа* или *строки s* в вызывающей строке. Если символа нет, возвращает -1 .

Результат `ss1.IndexOf('B')` 1.

Нестатические методы **IndexOfAny(a)** и **LastIndexOfAny(a)**.

Здесь **a** – массив символов, возвращают индекс первого (последнего) вхождения любого символа из массива **a**, который обнаружится в вызывающей строке.

Нестатический метод **StartsWith (s)** возвращает значение `true`, если вызывающая строка начинается с подстроки **s**, и значение `false` в противном случае.

Нестатический метод **EndsWith (s)** возвращает значение `true`, если вызывающая строка оканчивается подстрокой **s**.

8.2.5. Обработка строк

Нестатический метод **Split(r)** разбивает вызывающую строку на подстроки, которые возвращаются методом в виде строкового массива `string[]`.

r – это массив символов, содержащий разделители подстрок.

Если параметр **r** отсутствует, в качестве разделителя подстрок используется пробел.

Нестатический метод **Split(r, n)** разбивает вызывающую строку на подстроки, которые возвращаются методом в виде строкового массива из **n** подстрок.

r – это массив символов, содержащий разделители подстрок.

Пример

```
string str = "Какое слово ты скажешь, такое в ответ и услышишь.";
```

```
char[ ] seps = {',', ' ', ' '};
```

```
string[ ] parts = str.Split(seps);
```

```
Console.WriteLine("Результат разбиения строки на части: " );  
for(int i=0; i < parts.Length; i++) Console.WriteLine(parts[i]);
```

Можно `str.Split()`; или `str.Split(seps, 3)`;

Статический метод **Join(s, ss)** возвращает строку, которая содержит объединенные строки, переданные в массиве строк **ss**, разделенные строкой **s**.

Пример

```
string S = String.Join("yx", parts);
```

Результат:

Какое ух слово ух ты ух скажешь ух ух такое ух в ух ответ ух и ух услышишь ух

Статический метод **Join(s, ss, i, n)** возвращает строку, которая содержит **n** объединенных строк, переданных в массиве строк **ss**, начиная с **i**-й, разделенных строкой **s**.

Нестатический метод **Trim()** возвращает строку с удаленными из вызывающей строки начальными и конечными пробелами.

Например, `ss1 = ss1.Trim()`;

Нестатический метод **TrimEnd()** возвращает строку с удаленными из вызывающей строки конечными пробелами.

Нестатический метод **TrimStart()** возвращает строку с удаленными из вызывающей строки начальными пробелами.

Нестатический метод **PadLeft(n)** возвращает строку с добавленными в вызывающую строку начальными пробелами в таком количестве, чтобы общая длина результирующей строки стала равной **n**.

Например, `ss1 = "fffff"; ss1 = ss1.PadLeft(8)`;

Будут добавлены слева 3 пробела.

Нестатический метод **PadRight(n)** возвращает строку с добавленными справа в вызывающую строку пробелами в таком количестве, чтобы общая длина результирующей строки стала равной **n**.

Есть еще варианты этих методов:

Trim(ch) PadLeft(n, ch) PadRight(n, ch)

Нестатический метод **Insert(i, s)** возвращает строку, которая является результатом вставки строки **s** в вызывающую строку, начиная с **i**-й позиции.

Например:

```
ss1 = "Иванов двоечник!"; ss1 = ss1.Insert(6, " не");
```

Нестатический метод **Remove(i, n)** возвращает строку, которая является результатом удаления из вызывающей строки **n** символов, начиная с **i**-го:

```
ss1 = ss1.Remove(6, 3);
```

Нестатический метод **Replace(s1, s2)** возвращает строку, равную вызывающей, в которой все символы или строки **s1** заменены на символ или строку **s2**.

```
Например: ss1 = ss1.Replace('o', 'a');  
           ss1 = ss1.Replace("o", "fff");
```

Нестатический метод **Substring(i, n)** возвращает подстроку вызывающей строки из **n** символов, начиная с **i**-й позиции.

ToLower() возвращает строку из строчных букв.

ToUpper() возвращает строку из прописных букв.

8.2.6. Форматирование строк

Статический метод **Format(s,v1,v2,...,vn)** форматирует объекты **v1,v2,...,vn** согласно соответствующим командам форматирования, содержащимся в строке **s**. Возвращает копию строки **s**, в которой команды форматирования заменены форматированными данными:

```
string ss3=  
    string.Format(" Стипендия: {0,3:d}$ Средний балл:  
{1,5:f1}", 200, 8.68);
```

Результат:

```
Стипендия: 200$ Средний балл: 8,7
```

Нестатический метод **ToString(fmt)** возвращает строковое представление вызывающего объекта в соответствии с заданным спецификатором формата, переданным в параметре **fmt**.

```
Например, 3.99.ToString("f5")      x.ToString("c")
```

8.3. Строки типа **StringBuilder**

8.3.1. Создание строк

Класс **StringBuilder** определен в пространстве имен **System.Text**.

Существуют различные способы создания строки:

```
StringBuilder <имя строки> = new StringBuilder( );
```

```
Например, StringBuilder z = new StringBuilder( );
```

В этом случае создается пустая строка с объемом памяти 16 байт. Длина ее при этом равна нулю:

```
StringBuilder <имя строки> =  
    new StringBuilder(<объем памяти> );
```

Например, `StringBuilder z = new StringBuilder(50);`

```
StringBuilder <имя строки> =  
    new StringBuilder(<инициализирующая строка> );
```

Например, `StringBuilder z = new StringBuilder("Привет");`

```
StringBuilder <имя строки> =  
new StringBuilder(<инициализирующая строка>,  
    <объем памяти> );
```

Например, `StringBuilder z = new StringBuilder("Привет",50);`

```
StringBuilder <имя > =  
new StringBuilder(<иниц. строка>, <нач. индекс>,  
    <кол-во символов>, <объем памяти> );
```

Например, `StringBuilder z = new StringBuilder("Привет",2,4, 50);`

Обратиться к символу строки можно по индексу `z[6]`.

В классе **StringBuilder** можно изменять элементы строки используя обращение по индексу.

Например, `z[3] = 'D';`

8.3.2. Свойства и методы класса *StringBuilder*

Свойства:

Length используется для определения количества символов в строке класса.

Например, `z.Length` – длина строки `z`.

MaxCapacity максимальный объем памяти. Только для чтения.

Capacity используется для получения и установки объема памяти, отводимой под строку.

При установке значения этого свойства меньше текущей длины строки или больше максимального значения генерируется исключение **ArgumentOutOfRangeException**.

Методы:

Нестатический метод **Append** используется для добавления в конец строки: перегружен 18 раз.

Append(s) – добавление объекта **s** в конец вызывающей строки.
s – это может быть число, символ, строка, массив символов, **true** или **false**.

```
z.Append("ggg"); z.Append(2.5); z.Append('s');  
char[ ] h = { 'B','a','a','y','!' }; z.Append(h);
```

Append(s, n) – добавление символа **s** в конец вызывающей строки **n** раз.

```
z.Append('f',3);
```

Append(s, i, n) – добавление в конец вызывающей строки части строки или массива символов **s** из **n** символов, начиная с **i**-го:

```
z.Append("fff",1,2); z.Append(h,1,2);
```

Нестатический метод **Insert(i, s)** вставляет объект **s** в вызывающую строку, начиная с **i**-й позиции.

s – это может быть число, символ, строка, массив символов, **true** или **false**.

```
Например, z.Insert(1,"hhh"); z.Insert(1,'m'); z.Insert(1,2.4);
```

Методы **Replace** и **Remove** аналогичны одноименным методам класса **String**, только они не создают новую строку, а изменяют вызывающую.

AppendFormat(s,v1,v2,...,vn) добавляет к вызывающей строке строку **s**, в которой команды форматирования заменены форматированными данными **v1,v2,...,vn**:

```
z.AppendFormat(" добавили {0,5:f2} и еще {1}", 2.5, 'v');
```

AppendLine(s) добавляет к вызывающей строке строку **s** и символ конца строки:

```
z.AppendLine("fff"); Console.Write(z);
```

Нестатический метод **ToString()** преобразует вызывающую строку к типу **string**:

```
string sss = z.ToString();
```

Нестатический метод **ToString(i,n)** преобразует подстроку вызывающей строки из **n** символов, начиная с **i**-го, к типу **string**:

```
string sss1 = z.ToString(2, 10);
```

8.4. Регулярные выражения

Регулярные выражения – это особым образом отформатированные строки, используемые для поиска шаблонов в тексте, для подтверждения корректности формата данных, редактирования строк и т. д.

Для использования регулярных выражений в библиотеке .NET есть классы, которые объединены в пространство имен **System.Text.RegularExpressions**.

Это пространство имен не добавляется автоматически в список доступных пространств имен, поэтому в программе нужно вручную набрать оператор

```
using System.Text.RegularExpressions;
```

Регулярное выражение состоит из символов двух видов: *обычных символов*, представляющих в выражении самих себя, и *метасимволов*.

Метасимвол – это специальный символ, который служит обозначением класса символов (например, цифры или буквы), уточняет позицию поиска или задает количество повторений символов в выражении.

Классы символов:

. (**точка**) – любой символ, кроме \n .

Например, выражение "**К.т**" соответствует фрагментам строк **Кот, Кит, К&т** и т. д.

Чтобы точка воспринималась непосредственно (не как метасимвол), нужно перед ней ставить \.

[последовательность символов] или **[диапазон символов]** – любой одиночный символ из последовательности (или диапазона) внутри скобок.

Например, выражение "**К[иоэ-я]т**" соответствует фрагментам строк:

Кот, Кит, Кэт, Кят и Кют.

[^последовательность символов] или **[^диапазон символов]** – любой одиночный символ, не входящий в последовательность (или диапазон символов) внутри скобок.

Например, выражение "**К[^иоэ-я]т**" соответствует фрагментам строк **Кft, К&т** и не соответствует **Кот, Кит, Кэт, Кят и Кют.**

\w – любая буква или цифра.

Например, выражение "К\wt" или @"K\wt" соответствует фрагментам строк **Kft, K7т, Кот, К_т** и не соответствует **К:т, К%т** и т. д.

\W – любой символ, не являющийся буквой или цифрой.

Например, выражение "К\Wт" или @"K\Wт" соответствует фрагментам строк **К:т, К%т, К т, К\пт** и не соответствует **Kft, K7т, Кот, К_т** и т. д.

\s – любой пробельный символ (пробел, \n, \t, \v, \f, \r).

Например, выражение "К\sт" или @"K\sт" соответствует фрагментам строк **К\tr, К\vt, К т, К\пт**.

\S – любой не пробельный символ.

Например, выражение "К\Sт" или @"K\Sт" соответствует фрагментам строк **KRт, Кот, K7т, К\bt**.

\d – любая десятичная цифра.

Например, выражение "К\dт" или @"K\dт" соответствует фрагментам строк **К0т, К3т**.

\D – любой символ, не являющийся цифрой.

Уточняющие метасимволы (мнимые – им никакой символ в тексте не соответствует):

^ – фрагмент, соответствующий регулярному выражению, должен располагаться только в начале строки.

Например, в строках " **Кот Кит** " и " **Кыт Кит** " нет фрагмента, соответствующего регулярному выражению "**^К[ои]т**".

\$ – фрагмент, соответствующий регулярному выражению, должен располагаться только в конце строки. Этот метасимвол нужно располагать в конце регулярного выражения: "**@К[ои]т\$**".

\b – фрагмент, соответствующий регулярному выражению, должен располагаться на границе слова. Например, в строке " **fКит Котd** " фрагментом, соответствующим регулярному выражению "**@\bК[ои]т**", будет **Кот**. Выражению "**@К[ои]т\b**" будет соответствовать **Кит**.

\B – фрагмент, соответствующий регулярному выражению, не должен располагаться на границе слова. Какая именно граница имеется в виду, зависит от расположения этого метасимвола в регулярном выражении.

Повторители:

* – ноль или больше повторений предыдущего элемента.

Например, фрагментом, соответствующим регулярному выражению "**Ки*т**" будет **Кт, Кит, Киит** и т. д.

Выражению "**К[ои]*т**" будут соответствовать **Кт**, **Кит**, **Кот**, **Кииоит**, **Коиоит** и т. д.

+ – одно или больше повторений предыдущего элемента.

Выражению "**К[ои]+т**" будут соответствовать **Кит**, **Кот**, **Кииоит**, **Коиоит** и т. д., но не будет соответствовать **Кт**.

? – ни одного или одно повторение предыдущего элемента. Выражению "**К[ои]?т**" будут соответствовать **Кит**, **Кот**, **Кт**.

{**n**} – **n** повторений предыдущего элемента. Выражению "**Ки{3}т**" будет соответствовать **Кииит**.

{**n**, } – **n** или больше повторений предыдущего элемента.

{**n**, **m**} – от **n** до **m** повторений предыдущего элемента.

Выражению "**Ки{2,3}т**" будут соответствовать **Киит** и **Кииит**.

В качестве повторяющегося элемента может использоваться группа символов, заключенная в скобки.

Например, "**Тра(-ля){2}**", "**Тра(-л[яю]){3}**".

В регулярных выражениях можно использовать конструкцию выбора из нескольких элементов. Варианты выбора перечисляются через вертикальную черту.

Например, "**Василий|Петя|Ваня**".

В строке "Ваня Петя Василий" этому выражению будет соответствовать **Ваня**, в строке "Василий Ваня Петя" – **Василий**.

Фрагмент текста, соответствующий фрагменту регулярного выражения, можно запомнить в некоторой переменной:

(?<имя переменной>фрагмент выражения)

и использовать эту переменную внутри выражения.

Например, **@"(?<s>\w\w){2,}\k<s>"**.

Чтобы использовать в программе регулярное выражение, нужно создать объект класса **Regex**:

Regex <имя> = new Regex(<строка, задающая вид РВ>)

Например,

```
string rv = @"(?<s>\w\w){2,}\k<s>",
```

```
Regex R = new Regex(rv);
```

*Методы класса **Regex**:*

Нестатический **IsMatch(s)** возвращает true, если фрагмент, соответствующий вызывающему регулярному выражению в строке s, найден, и false в противном случае:

```
string rv = "(Ba\\wя){3}", s1 = "ВасяВаняВаля";  
Regex R = new Regex(rv);
```

R.IsMatch(s1) будет **true**.

Нестатический **Match(s)** возвращает первый фрагмент, соответствующий вызывающему регулярному выражению в строке **s**.

Для примера выше **R.Match(s1)** вернет **Вася**.

Нестатический **Matches(s)** возвращает коллекцию фрагментов, соответствующих вызывающему регулярному выражению в строке **s** (объект класса **MatchCollection**).

Обратиться к каждому элементу коллекции можно по его номеру (нумерация с нуля), указав его в квадратных скобках.

Например, **R.Matches(s1)[0]**.

Узнать количество найденных фрагментов можно с помощью свойства **Count** класса **MatchCollection**.

Например, **R.Matches(s1).Count**.

Преобразовать элемент коллекции в строку можно с помощью метода **ToString**.

Например, **string s = R.Matches(s1)[0].ToString()**.

Нестатический метод **Split(s)** разбивает строку **s** в соответствии с разделителями, заданными с помощью регулярного выражения и возвращает эти фрагменты в виде массива строк:

```
string rv = "[- ,.:;|]+",  
s1 = "Вася, Ваня::,Валя; Петя,- Муся.;; Буся."  
Regex R = new Regex(rv);  
string[] s = R.Split(s1);  
foreach (string ss in s) Console.WriteLine(ss);
```

Статический метод **Replace(s, srv, s1)** возвращает строку, которая является копией строки **s**, в которой все фрагменты, соответствующие регулярному выражению **srv** заменены на строку **s1**.

```
string stip = «Янчев – 200 тыс. руб. Кухаренко – 500000руб.»;  
string stip1 = Regex.Replace(stip, "( тыс. руб.)(000руб.)", "$");
```

Результат: **Янчев – 200\$ Кухаренко – 500\$**.

Тема 9. НАСЛЕДОВАНИЕ

9.1. Основы наследования. Создание производных классов

Наследование – это форма многократного использования программных средств, при которой новые классы создаются поглощением элементов существующего класса с приданием им новых возможностей.

Наследование позволяет:

- исключить повторяющиеся фрагменты кода;
- упростить модификацию программы и разработку новых программ на основе существующих;
- экономить время разработки программы;
- использовать объекты, исходный код которых недоступен, но которые нужно изменить.

Класс, который наследуется, называется *базовым (предком)*.

Класс, который наследует базовый класс, называется *производным (потомком)*.

В других языках программирования могут встречаться термины «надкласс» (суперкласс) и «подкласс».

После создания каждый производный класс может стать базовым для будущих производных классов.

Прямой базовый класс – это базовый класс, из которого совершает наследование производный класс.

Косвенный базовый класс – это класс, который наследуется из двух или более уровней выше производного по классовой иерархии.

Например, пусть имеется иерархия классов, показанная на рис. 9.1.

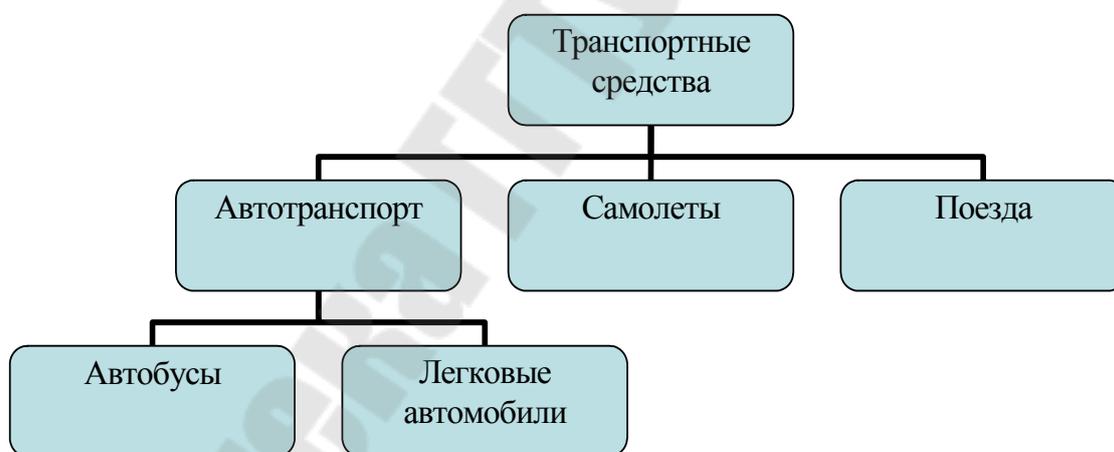


Рис. 9.1. Дерево классов

Для классов «Автотранспорт», «Самолеты» и «Поезда» класс «Транспортные средства» является *прямым* базовым классом, а для классов «Автобусы» и «Легковые автомобили» – *косвенным*. С# не поддерживает *множественного* наследования, когда один класс производится более чем из одного прямого базового класса.

Но все преимущества множественного наследования предоставляет возможность создания и использования интерфейсов, которые будут рассмотрены в последующих лекциях.

Класс не может быть базовым (ни прямо, ни косвенно) для самого себя.

Каждый объект производного класса является объектом его базового класса, но не наоборот.

(Все самолеты являются транспортными средствами, но не все транспортные средства являются самолетами).

Общая форма объявления класса, который наследует базовый класс:

```
class <имя_произв._кл.> : <имя_базового_кл.>
    { <тело класса> }
```

Пример

```
class Chelovek
{ public string fam, dat_rog;
  public double rost, ves;
  public void info()
  { Console.WriteLine(fam + " " + dat_rog + " " + rost + " " + ves);
  }
}
```

Будем использовать описанный класс в качестве базового для класса «Студент»:

```
class Student : Chelovek
{ public string vuz, gruppа;
  public int[] oцenki ;
  public void info_uch()
  { Console.Write(vuz + " " + gruppа + " оцeнки:");
    foreach (int x in oцenki) Console.Write(" "+x);
    Console.WriteLine();
  }
}
```

Тогда в методе Main можно так работать с объектом класса:

```
Student St1 = new Student();
//обращение к унаследованным полям от класса Chelovek:
St1.fam = "Левкович"; St1.dat_rog = "12.07.89";
St1.rost = 185; St1.ves = 78;
//обращение к собственным полям класса
St1.oцenki = new int[] { 7, 2, 6 };
```

```

St1.vuz="ГГТУ им. П.О.Сухого"; St1.gruppa="ИТ-21";
St1.info(); // от имени объекта st1 вызван унаследованный метод
St1.info_uch(); // от имени объекта st1 вызван собственный метод

```

Схематично класс Student можно представить так, как показано на рис. 9.2.

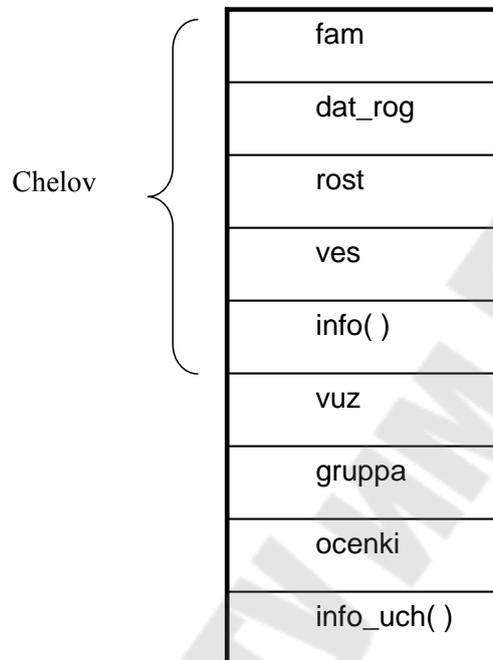


Рис. 9.2. Структура класса Student

Если класс используется в качестве базового производными классами, это не означает невозможность использования его самого.

Пример

```

Chelovek Man = new Chelovek();
Man.fam = "Сидоров"; Man.rost = 178; Man.info();

```

При этом нельзя использовать оператор **Man.info_uch()**

Класс **Chelovek** можно использовать в качестве базового и для других классов.

Пример

```

class Prepodavatel:Chelovek
{
    public int nagruzka, zarplata;
    public string kafedra;
    public void info_p()
{
        Console.WriteLine(kafedra + " нагрузка " + nagruzka +

```

```

        " зарплата:" + zarplata);
    }
}

```

В свою очередь, класс Student может быть базовым для другого класса:

```

class Superman : Student
{ public int stip = 500;}

```

Тогда допустимы следующие операторы:

```

SuperMan Sm = new SuperMan( );    Sm.fam = «Потер»;
Sm.ocenki = new int[] { 9, 9, 9 };    Sm.gruppa = "ИТ-21";
Console.WriteLine(Sm.fam + " " + Sm.stip);    Sm.info_uch();

```

9.2. Доступ к элементам базового класса

Производный класс наследует от базового класса все, что он имеет. Но воспользоваться в производном классе можно не всем наследством.

Производный класс не может получить доступ к тем из элементов, которые объявлены закрытыми (**private**).

Косвенное влияние на такие элементы – лишь через public-функции базового класса.

Например, пусть в классе **Chelovek** поле **fam** объявлено со спецификатором **private**:

```

private string fam;

```

Тогда следующий фрагмент будет ошибочным:

```

class Prepodavatel:Chelovek
{ public int nagruzka, zarplata;
  public string kafedra;
  public void info_p( )
  { Console.WriteLine(fam); //Ошибка! Попытка обращения к
                           // закрытому полю базового класса
    Console.WriteLine(kafedra + " нагрузка " + nagruzka +
                      " зарплата:" + zarplata);
  }
}

```

Ошибки можно избежать, определив в базовом классе открытое свойство для доступа к закрытому полю fam:

```

public string Fam
{ get { return fam; }
}

```

```

        set { fam = value; }
    }

```

Тогда обращение к закрытому полю можно заменить именем свойства.

C# позволяет создавать *защищенные элементы класса*.

Защищенный член создается с помощью спецификатора доступа **protected**.

Этот спецификатор обеспечивает открытый доступ к элементам базового класса, но только для производного класса. Для других классов доступ к защищенным элементам закрыт.

Например, дополним класс **Chelovek** защищенным полем

```

protected string status;

```

Тогда можно получить доступ к этому полю в методах классов **Student**, **Prepod** и **SuperMan**, но нельзя обратиться к полю **статус**, например, в классе **Program**.

При использовании производного класса в качестве базового для создания другого производного класса любой защищенный член исходного базового класса, который наследуется первым производным классом, также наследуется в статусе защищенного и вторым производным классом:

```

class Student : Chelovek
{
    public string vuz, gruppа;
    public int[ ] oцenki ;
    public void info_uch()
    {
        status = "студент";
        Console.WriteLine(status+" "+vuz + " " + gruppа+" оцeнки:" );
        foreach (int x in oцenki) Console.WriteLine(" "+x);
        Console.WriteLine();
    }
}

```

Можно так:

```

class SuperMan : Student
{
    public int stip = 500;
    public void info_s()
    {
        status = "суперстудент";
        Console.WriteLine(Fam+" – "+status); }
}

```

Но нельзя в классе **Program**, например, использовать код
Chelovek Man = new Chelovek();
Man.status = "человек";

9.3. Использование конструкторов базового класса

Если в базовом и производном классах не определены конструкторы, то при создании объекта производного класса используются конструкторы по умолчанию, предоставляемые системой программирования.

Базовые и производные классы могут иметь собственные конструкторы.

Конструктор базового класса инициализирует поля объекта, соответствующие базовому классу, а конструктор производного класса – поля объекта, соответствующие производному классу.

Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса без параметров (при его отсутствии – конструктор по умолчанию).

Например, определим в классе **Student** конструктор:

```
public Student(string vz, string grp, int n)
{
    vuz = vz; gruppа = grp; int[] oцenki = new int[n];
}
```

Теперь при создании объекта класса **Student**

```
Student St1 = new Student("ГГТУ им. П.О.Сухого","ИТ-21",3);,
```

так как конструктор в классе **Chelovek** отсутствует, будет вызван конструктор по умолчанию для полей, относящихся к базовому классу, а потом конструктор класса **Student**.

Но возникает следующая проблема – в классе **Student** нет конструктора без параметров, а в производном от него классе **SuperMan** нет явного вызова конструктора базового класса.

Есть два пути разрешения ситуации:

1) создать в классе **Student** конструктор без параметров, хотя бы пустой;

2) явно указать вызов нужного конструктора класса **Student**.

Конструктор производного класса с явным вызовом конструктора базового класса определяется следующим образом:

<имя конструктора>(<список_параметров>):
base (<список_аргументов>)

{ тело конструктора }

Список аргументов содержит аргументы для конструктора базового класса.

Если в базовом классе несколько конструкторов, то будет выбран конструктор, соответствующий количеству и типу аргументов в списке после слова **base**.

Например, определим в классе **SuperMan** конструктор с вызовом конструктора, определенного в классе **Student**:

```
public SuperMan(string vz, string grp, int n, int st) : base(vz, grp, n)  
{ stip = st; }
```

Тогда создание объекта класса с помощью этого конструктора может выглядеть следующим образом:

```
SuperMan Sm = new SuperMan("ГГТУ им. П.О.Сухого", "ИТ-22", 3, 1000);
```

Можно так:

```
public SuperMan( int st):  
base("ГГТУ им. П.О.Сухого", "ИТ-22", 3)  
{ stip = st; }
```

Создание объекта с помощью этого конструктора:

```
SuperMan Sm = new SuperMan(1000);
```

В иерархии классов конструкторы базовых классов вызываются, начиная с самого верхнего уровня

Для создания объектов в программе можно применять конструкторы трех степеней защиты:

public – при создании объектов в рамках данного пространства имен, в методах любого класса – члена данного пространства имен;

protected – при создании объектов в рамках производного класса, в том числе при построении объектов производного класса, а также для внутреннего использования классом – владельцем данного конструктора;

private – применяется исключительно для внутреннего использования классом-владельцем данного конструктора.

9.4. Переопределение элементов базового класса. Соккрытие имен

При объявлении элементов производного класса в C# разрешено использовать те же самые имена, которые применялись при объявлении элементов базового класса.

В этом случае элемент базового класса становится скрытым в производном классе.

При переопределении наследуемого элемента его объявление в классе-наследнике должно предваряться спецификатором **new**.

Если этот спецификатор в объявлении производного класса опустить, компилятор всего лишь выдаст предупреждающее сообщение, чтобы напомнить о факте сокращения имени.

Роль ключевого слова **new** в этом случае совершенно отличается от его использования при создании объектов.

По мнению создателей языка C#, тот факт, что ранее (до момента его переопределения) видимый элемент базового класса стал недоступен и невидим извне требует явного дополнительного подтверждения со стороны программиста.

Например, переопределим в классе **Student** метод **info()**:

```
public void info()  
    { Console.WriteLine(Fam + " " + dat_rog); }
```

Программа будет компилироваться и работать, но сначала будет выдано предупреждение. Чтобы оно не появлялось, лучше использовать спецификатор **new**:

```
new public void info()  
    { Console.WriteLine(Fam + " " + dat_rog); }
```

Константа, поле, свойство, объявленный в классе класс или структура скрывают все одноименные элементы базового класса.

Пример

```
class X  
{  
    int d;  
    public X(int x) { d = x; }  
    public void f( )  
        { Console.WriteLine("d="+d); }  
    public void f(int x)  
        { Console.WriteLine("x=" + x); }  
}  
class Y : X
```

```

    { int s;
      new double f;
      public Y(int x) : base(x) { s = 10; f = 20.5; }
      public void show()
      {
        Console.WriteLine(" f=" + f);
        base.f(); base.f(s);
      }
    }

```

Тогда можно так обращаться к элементам описанных классов:

```

X  x1 = new X(5); x1.f();
Y  y1 = new Y(3);
   y1.f(); y1.show();

```

Если поле **f** в производном классе описано со спецификатором **public**, то методы базового класса будут скрыты для других классов и этот оператор будет ошибочным. Можно только обращаться к полю производного класса **y1.f**;

Итак, если элемент в производном классе скрывает *нестатический* элемент с таким же именем в базовом классе, то формат обращения к нестатическому элементу базового класса из производного следующий:

base.<элемент базового класса>

Если элемент в производном классе скрывает *статический* элемент с таким же именем в базовом классе, то формат обращения к элементу

<имя класса>.< элемент базового класса >

Например, если в предыдущем примере метод **f** с параметром класса **X** сделать статическим:

```
public static void f(int x),
```

то оператор **base.f(s)**; следует заменить на **X.f(s)**;

Пусть в программе определен класс, наследующий от класса **Y**

```
class Z : Y:
```

```

    { new int f; // переопределено поле f
      public Z( ) : base(5) { }
      public void sh()
      { f = 5; base.f(); show();
        Console.WriteLine(" В Z f=" + f); }
    }

```

Так как поле **f** класса **Y** закрыто, то ссылка **base** означает класс **X** и происходит обращение к методу класса **X**. Если бы поле **f** класса **Y** было бы защищенным или открытым, то ключевое слово **base** обозначало бы класс **Y**, и оператор **base.f()**; был бы ошибочным, а можно было бы использовать, например, оператор **base.f = 2;**.

Определенный в производном классе метод скрывает в базовом классе свойства, поля, типы, обозначенные тем же самым идентификатором, а также одноименные методы с той же сигнатурой.

Например, пусть в классе **Y** элемент **f** не поле, а метод:

```
public void f(int x)  
    { Console.WriteLine(" В Y x=" + x); }
```

Тогда внутри класса **Y** можно использовать оператор **f()**, т. е. метод **f** без параметров класса **X** остается видимым в классе **Y**, и к нему можно обращаться без ключевого слова **base**.

Оператор **f(50);** будет выполнять обращение к методу **f** с параметром класса **Y**.

Чтобы вызвать метод с такой сигнатурой класса **X**, нужно использовать оператор **base.f(50)** (для нестатического метода) или **X.f(50)** (для статического).

Объявляемые в производном классе индексы скрывают индексы в базовом классе с аналогичной сигнатурой.

Пример

```
class X  
    { protected int[ ] x=new int[5];  
      public int this[int i]  
        { get { return x[i]; }  
          set { x[i] = value; }  
        }  
    }
```

В классе **Y** переопределим индексатор:

```
class Y:X  
    {  
        new public int this[int i]  
        {  
            get { return x[i]; }  
        }  
    }
```

Тогда, если создан объект класса **X**

```
X x1 = new X();
```

то оператор `x1[3] = 5;` обращается к индексатору, определенному в классе **X** и ошибки нет.

Для объекта класса **Y** `y1` оператор `y1[3] = 5;` вызывает индексатор, переопределенный в классе **Y**, что приведет к ошибке, поскольку индексатор только для чтения.

Если в производном классе переопределяется вложенный класс, то из производного класса виден только переопределенный класс, а для доступа к классу вложенному в базовый нужно использовать *полное квалифицированное имя*:

<имя базового класса>.<имя вложенного класса>

Пример

```
class A
{
    public class X
    { public int x = 100;}
}
class B:A
{
    new class X // Вложенный класс базового класса скрывается
    { public int x = 10;
      public double y = 2.5;
    }
    static void Main(string[ ] args)
    {
        X x1 = new X( ); // объект класса X из класса B
        Console.WriteLine(x1.x+" "+x1.y);
        A.X x2 = new A.X( ); // объект класса X из класса A
        Console.WriteLine(x2.x);
        Console.ReadKey( );
    }
}
```

9.5. Предотвращение наследования

Для того чтобы запретить наследовать от класса, необходимо при определении использовать спецификатор **sealed**.

Классы со спецификатором **sealed** называют бесплодными:

```
sealed class A { ... }
```

```
// Следующий класс создать невозможно.
```

```
class B : A { // ОШИБКА! Класс A не может иметь наследников.
... }
```

С помощью спецификатора **sealed** можно также запрещать переопределение элементов класса в производных классах:

```
class X
{ sealed public void f0()
  {}
}
class Y:X
{ public void f0(){}
  // ОШИБКА! Переопределение f0 запрещено!
}
```

9.6. Абстрактные классы

Если базовый класс используется исключительно как основа для создания классов – наследников, если он никогда не будет использован для создания объектов, если часть методов (возможно, что и все) базового класса обязательно переопределяется в производных классах, то класс объявляют как абстрактный со спецификатором **abstract**.

Методы, которые будут обязательно переопределяться в потомках, также определяются со спецификатором **abstract** и не содержат тела.

Объявление абстрактной функции завершается точкой с запятой.

В производном классе соответствующая переопределяемая абстрактная функция обязательно должна включать в заголовок функции спецификатор **override**.

На основе абстрактного класса невозможно создать объекты. Попытка создания соответствующего объекта – представителя абстрактного – класса приводит к ошибке.

Пример

```
abstract class X
{ protected double xn,xk,dx;
  abstract protected double f(double x);
  public void tab( )
  {Console.WriteLine
```

```

(" ||-----|| ");
    Console.WriteLine
(" ||    АРГУМЕНТ    ||    ФУНКЦИЯ    || ");
    Console.WriteLine

(" ||-----|| ");
    double x = xn;
    while (x < xk + dx / 2)
        { Console.WriteLine(" || {0,12:f2} || {1,15:f3} || ", x, f(x));
          x = x + dx; }
    Console.WriteLine

(" ||-----|| ");
    }
    }
class Y : X
    {
        public Y(double xxn, double xxk, double dxx)
            { xn = xxn; xk = xxk; dx = dxx; }
        protected override double f(double x) { return Math.Sin(x); }
    }
class Z : X
    { public Z()
      { xn = Double.Parse(Console.ReadLine());
        xk = Double.Parse(Console.ReadLine());
        dx = Double.Parse(Console.ReadLine()); }
      protected override double f(double x) { return Math.Cos(x); }
    }
class Program
    {
        static void Main(string[ ] args)
            {
                Y f1 = new Y(2, 5, 0.5); f1.tab();
                Z f2 = new Z( ); f2.tab( );
                Console.ReadKey( );
            }
    }

```

В C# можно описать массив объектов базового класса и занести в него объекты производного класса. Но для объектов производного

класса вызываются только методы и свойства, унаследованные от предка, т. е. определенные в базовом классе.

9.7. Виртуальные методы

Методы, которые в производных классах могут быть реализованы по другому, можно определять в качестве виртуальных.

Для этого используется ключевое слово **virtual**.

Например, **virtual public void vivod()**.

Объявление метода виртуальным означает, что решение о том, какой из одноименных методов иерархии будет вызван, принимается не на стадии компиляции, а на стадии выполнения программы, в зависимости от вызывающего объекта.

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

Переопределенный виртуальный метод должен иметь такой же набор параметров, как и соответствующий метод базового класса:

```
class F1
{
    protected double x;
    public F1(double x1)
    { x = x1; }
    virtual public double f()
    { return x + 2; }
    public void vivod()
    { Console.WriteLine("x=" + x + " значение функции = " +
        f()); }
}
class F2 : F1
{
    public F2(double x1):base(x1)
    { }
    public override double f()
    { return x + 5; }
}
class Program
{
    static void Main(string[] args)
    {
        F1 y = new F1(3); y.vivod();
        F2 z = new F2(5); z.vivod();
        Console.ReadKey();
    }
}
```

```
}  
}
```

Результаты работы:

x=3 значение функции = 5

x=5 значение функции = 10

Если без `override`, то результаты работы:

x=3 значение функции = 5

x=5 значение функции = 7

Тема 10. ИНТЕРФЕЙСЫ

10.1. Описание интерфейса

Интерфейс – это тип данных, предназначенный для определения характеристик и поведения, присущих классам, реализующим этот интерфейс.

В интерфейсе задается набор методов, свойств и индексаторов, которые *должны* быть реализованы в производных классах:

```
[<спецификаторы>] interface <имя > [: предки]  
    {<тело интерфейса>}
```

Спецификаторы – это **public** или **internal** (по умолчанию).

Для вложенных в класс интерфейсов можно использовать спецификаторы **new**, **protected**, **private**.

Существует традиция имя интерфейса начинать с префикса **I** (но это не является обязательным).

Предки – это интерфейсы, элементы которых наследует данный интерфейс. Имена предков перечисляются через запятую.

Тело интерфейса составляют заголовки методов, шаблоны свойств и индексаторов, события.

Заголовки методов объявляются следующим образом:

```
<тип_результата> <имя_метода> (<список_параметров>);
```

В интерфейсе методы неявно являются открытыми (**public**), при этом не разрешается *явным образом* указывать спецификатор доступа.

Шаблон свойства представляется следующим образом:

```
<тип свойства> <имя свойства>  
    { get ; set ;}
```

Свойство может быть только для чтения { **get ;** } или только для записи { **set ;** }.

Шаблон индекатора (одномерного) имеет вид:

```
<тип результата> this[<тип индекса> <имя индекса>]
    { get ; set ; }
```

Интерфейсы не могут иметь константы и поля. Они не могут определять конструкторы, деструкторы, операции.

Ни один член интерфейса не может быть объявлен статическим.

Например, рассмотрим интерфейс, определяющий некоторые характеристики объектов живой природы:

```
interface IOpisanie
{
    double Ves { get;} //шаблоны свойств

    double Rost {get;} // только для чтения

    int Vozrast( ); // метод

    void Vyvod(string name); // метод
}
```

10.2. Реализация интерфейсов

Интерфейс может быть реализован любым количеством классов. При этом один класс может реализовать любое число интерфейсов.

Чтобы реализовать интерфейс, нужно указать его имя после имени класса:

```
class <имя_класса> : <имя_интерфейса>
    {<тело класса>}
```

Класс, реализующий интерфейс, должен определять *все* элементы этого интерфейса.

В классах, реализующих интерфейс, можно также определять дополнительные элементы, не входящие в интерфейс.

Методы, которые реализуют интерфейс, должны быть объявлены *открытыми*.

Сигнатура в реализации метода должна в точности совпадать с сигнатурой метода, заданной в определении интерфейса.

Например, рассмотрим класс **Chelovek**, реализующий приведенный ранее интерфейс **IOpisanie**:

```
class Chelovek : IOpisanie
```

```

    {
        protected string fam;
        double ves, rost;
        public int god_rogd;
        public Chelovek(string f, double v, double r, int g_r)
            { fam = f; ves = v; rost = r; god_rogd = g_r; }
        public double Ves {get {return ves;}}
        public double Rost { get { return rost; } }
        public int Vozrast()
            { return DateTime.Now.Year – god_rogd; }
        public void Vyvod(string name)
            { Console.WriteLine(name + " " + fam + " "+Vozrast( )); }
    }

    public int Vozrast(int g) { return g – god_rogd; }
}

```

Тогда, например, в методе Main, можно следующим образом работать с объектом класса:

```

Chelovek men = new Chelovek("Васечкин", 89, 180, 1975);
men.Vyvod("бизнесмен");
Console.WriteLine("его год рождения:"+men.god_rogd+
    " в 2000 г. ему было "+men.Vozrast(2000));

```

Можно создавать объекты типа интерфейс следующим образом:

```

<имя интерфейса> <имя объекта> =
    new <конструктор класса, реализующего интерфейс>;

```

Например, `IOpisanie men1 = new Chelovek("Петров",65,165,1995);`

В этом случае созданный объект имеет доступ только к элементам интерфейса.

Например, можно `men1.Vyvod("школьник");`

Но нельзя использовать обращение

```

men1.god_rogd или men1.Vozrast(2000)

```

Этот же интерфейс `IOpisanie` может быть реализован другим классом, например, классом `Sobaka`:

```

class Sobaka : IOpisanie
    { string klichka;
      double ves, holka;
      string dat_rogd; // дата в формате дд.мм.гггг
      int[ ] schenki;
      public Sobaka(string imja, double v,double h,
                    int n,int[ ] k_sch,string d_r)
    }

```

```

{
    klichka = imja; ves = v; holka = h; dat_rogd = d_r;
    schenki = new int[n]; schenki = k_sch;
}
public double Ves { get { return ves; } }
public double Rost { get { return holka; } }
public int Voзраст( )
    { return
      DateTime.Now.Year –
      Convert.ToInt32(dat_rogd.Substring(dat_rogd.Length-4, 4));
    }
public void Vyvod(string poroda)
    { Console.WriteLine(poroda + " " + klichka + " " + Voзраст( ));
    }
public void Vyvod( )
    { Console.WriteLine("Количество щенков:");
      for(int i=0; i<schenki.Length; i++)
        Console.WriteLine((i+1) + "-й раз:" + schenki[i]);
    }
}
}

```

Классы могут реализовать несколько интерфейсов. В этом случае имена интерфейсов отделяются запятыми.

Например, создадим еще один интерфейс (рис. 10.1).

interface IObrabotka

```

{
    double Rezult( );
    bool Bolshe (IObrabotka S);
    int this[int i] { get;}
}

```

В данном случае в качестве аргумента можно использовать объект любого класса, реализующего данный интерфейс

Рис. 10.1

И пусть класс **Sobaka** реализует его наряду с интерфейсом **IOpisanie**.

Тогда заголовок класса должен выглядеть следующим образом:

```
class Sobaka : IOpisanie, IObrabotka
```

А класс нужно дополнить реализацией элементов интерфейса **IObrabotka**.

```

public double Rezult( ) // общее количество щенков
    { int s = 0; foreach (int sch in schenki) s = s + sch;
      return s; }
public int this[int i] { get { return schenki[i]; } }
public bool Bolshe(IObrabotka S)
    {
      if (Rezult( ) > S.Rezult( )) return true;
      else return false;
    }

```

Тогда, например, если **Mops** – объект класса **Sobaka**, допустим такой оператор:

```

if (Taksa.Bolshe(Mops))
    Console.WriteLine("У таксы больше щенков чем у мопса");
else
    Console.WriteLine("У таксы не больше щенков чем у мопса");

```

Если понадобится, чтобы студенты поддерживали интерфейс **IObrabotka**, нужно добавить этот интерфейс в заголовок класса:

```

class Student:Chelovek, IObrabotka

```

и дополнить класс Student реализацией элементов этого интерфейса.

Класс может наследовать базовый класс и реализовать один или несколько интерфейсов. В этом случае список интерфейсов должно возглавлять имя базового класса.

```

class <имя_класса> :
    <имя базового класса>, <имя_интерфейса_1>,
    <имя_интерфейса_2>, и т. д.

```

Если базовый класс реализовывал какой-либо интерфейс, то производный класс также является реализующим интерфейс.

При реализации в классе элемента интерфейса можно в заголовке явно указывать имя интерфейса перед именем метода, свойства или перед словом **this** индекса *без указания спецификатора доступа*:

```

<тип результата> <имя интерфейса >.<имя метода>
    (<список параметров>)
<тип результата> <имя интерфейса >.this[<тип> <индекс>]

```

К этим элементам можно обращаться только от имени объекта типа этого интерфейса.

Например, пусть в классе `Sobaka` индекатор описан следующим образом:

```
int IObrabotka.this[int i] { get { return schenki[i]; } }
```

Тогда следующий фрагмент приведет к ошибке:

```
int[ ] sch1 = {3,2,5};
```

```
Sobaka Mops = new Sobaka("Тузик", 20, 32, 3, sch1, "13.12.2000");
```

```
Console.WriteLine("У Тузика 1-й раз было щенков "+Mops[0]);
```

Правильно будет:

```
Console.WriteLine("У Тузика 1-й раз было щенков "+  
((IObrabotka)Mops)[0]);
```

или

```
IObrabotka Mops = new Sobaka("Тузик", 20, 32, 3, sch1, "13.12.2000");
```

```
Console.WriteLine("У Тузика 1-й раз было щенков "+Mops[0]);
```

Явное указание имени интерфейса может быть полезным при множественном наследовании интерфейсов, когда в разных интерфейсах встречаются методы с одинаковыми именами.

10.3. Операции `is` и `as`

Бинарная операция `is` определяет, совместим ли текущий тип объекта, расположенного слева от `is`, с типом, указанным справа.

<объект> is <тип>

Например, `x is Array`.

Результат операции равен `true`, если объект можно преобразовать к заданному типу, и `false` в противном случае.

Результат операции `Taksa is Student` – `false`.

Результат операции `Taksa is IOpisanie` – `true`.

Эту операцию можно использовать, чтобы проверить, поддерживает ли аргумент, используемый вместо параметра типа `object`, нужный интерфейс.

Например, метод `Bolshe` может быть следующим:

```
public bool Bolshe(Object S)  
{ if (S is IObrabotka)  
{  
    if (Rezult() > ( (IObrabotka) S).Rezult() ) return true;  
    else return false;  
}  
else
```

```

        { if (S is IOpisanie)
          { if (Rost > ((IOpisanie)S).Rost) return true; else return
false;
          }
          else throw new Exception("Несравнимые величины");
        } }

```

Тогда будем иметь право на использование такого оператора:

```

if (TheBad.Bolshe(men))
    Console.WriteLine(" Тимоти круче Васечкина");
else
    Console.WriteLine(" Тимоти не круче Васечкина");

```

Операция **as** выполняет преобразование к указанному типу, если это невозможно, формирует результат **null**.

<объект> as <тип>

Например, в метод **Bolshe** можно внести следующие изменения:

```

IOpisanie SS= S as IOpisanie;
if (SS!=null)
    { if (Rost > SS.Rost) return true;
      else return false;
    }
else throw new Exception("Несравнимые величины");

```

Литература

1. Павловская, Т. А. С#. Программирование на языке высокого уровня : учеб. для вузов / Т. А. Павловская. – Санкт-Петербург : Питер, 2007. – 432 с.
2. Биллиг, В. А. Основы программирования на С# / В. А. Биллиг. – Москва : Интернет-университет информац. технологий – ИНТУ-ИТ.ру, 2006. – 488 с.
3. Шилдт, Г. С. С#: Учебный курс / Г. С. Шилдт. – Санкт-Петербург : Питер, 2002. – 512 с.
4. Шилдт, Г. С. Полный справочник по С# / Г. С. Шилдт. – Москва : Издат. дом «Вильямс», 2004. – 752 с.
5. Фролов, А. В. Язык С# : самоучитель / А. В. Фролов, Г. В. Фролов. – Москва : Диалог-МИФИ, 2003. – 560 с.

Содержание

Введение	3
Тема 1. Основы объектно-ориентированного программирования	4
1.1. Основные направления в программировании.....	4
1.2. Понятие и основные принципы объектно-ориентированного программирования	5
1.3. Объекты и классы.....	8
Тема 2. Базовые конструкции языка C#	9
2.1. Характеристика языка.....	9
2.2. Идентификаторы	10
2.3. Классификация типов данных.....	11
2.4. Встроенные типы данных	12
2.5. Приведение типов	13
2.6. Переменные и константы	15
2.7. Структура консольной программы	16
2.8. Ввод-вывод в консольном приложении	17
2.9. Методы класса Math.....	20
Тема 3. Классы, основные элементы класса	20
3.1. Основные понятия и описание классов	20
3.2. Поля и константы класса	22
3.3. Методы	23
3.4. Конструкторы	28
3.5. Сбор мусора, деструкторы.....	31
3.6. Свойства класса	31
Тема 4. Обработка исключительных ситуаций	33
4.1. Понятие исключения, основные стандартные исключения ..	33
4.2. Оператор try	33
4.3. Генерирование исключений вручную	35
Тема 5. Использование массивов в C#	36
5.1. Основные понятия.....	36
5.2. Одномерный массив.....	37
5.3. Свойства и методы класса Array для обработки одномерных массивов.....	38
5.4. Многомерные массивы	40
5.4.1. Виды многомерных массивов	40
5.4.2. Двумерные прямоугольные массивы	40
5.4.3. Ступенчатые массивы.....	42

Тема 6. Индексаторы и операции класса	46
6.1. Одномерные индексаторы	46
6.2. Многомерные индексаторы	48
6.3. Операции класса	48
6.3.1. Унарные операции	49
6.3.2. Бинарные операции	51
6.3.3. Операции преобразования типа	54
Тема 7. Дополнительные возможности методов	56
7.1. Методы с переменным количеством аргументов	56
7.2. Перегрузка методов	57
Тема 8. Строки, символы и регулярные выражения	60
8.1. Символы	60
8.2. Строки типа string	62
8.2.1. Создание строки	62
8.2.2. Конкатенация строк	63
8.2.3. Сравнение строк	63
8.2.4. Поиск в строке	65
8.2.5. Обработка строк	65
8.2.6. Форматирование строк	67
8.3. Строки типа StringBuilder	67
8.3.1. Создание строк	67
8.3.2. Свойства и методы класса StringBuilder	68
8.4. Регулярные выражения	70
Тема 9. Наследование	73
9.1. Основы наследования. Создание производных классов	73
9.2. Доступ к элементам базового класса	77
9.3. Использование конструкторов базового класса	79
9.4. Переопределение элементов базового класса.	
Соккрытие имен	81
9.5. Предотвращение наследования	84
9.6. Абстрактные классы	85
9.7. Виртуальные методы	87
Тема 10. Интерфейсы	88
10.1. Описание интерфейса	88
10.2. Реализация интерфейсов	89
10.3. Операции is и as	93
Литература	95

Учебное электронное издание комбинированного распространения

Учебное издание

Романькова Татьяна Леонидовна

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

**Курс лекций
по одноименной дисциплине для студентов
специальности 1-40 01 02 «Информационные
системы и технологии (по направлениям)»
дневной формы обучения**

Электронный аналог печатного издания

Редактор *М. В. Аникеенко*
Компьютерная верстка *Н. Б. Козловская*

Подписано в печать 08.12.10.

Формат 60x84/16. Бумага офсетная. Гарнитура «Таймс».

Ризография. Усл. печ. л. 5,81. Уч.-изд. л. 5,95.

Изд. № 3.

E-mail: ic@gstu.by

<http://www.gstu.by>

Издатель и полиграфическое исполнение:
Издательский центр учреждения образования
«Гомельский государственный технический университет
имени П. О. Сухого».

ЛИ № 02330/0549424 от 08.04.2009 г.

246746, г. Гомель, пр. Октября, 48.