МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)» (СГАУ)

Основы программирования

Электронный учебно-методический комплекс
по дисциплине в LMS Moodle

САМАРА
2012

УДК 004

Автор-составитель: **Дегтярева Ольга Александровна**

Основы программирования [Электронный ресурс] : электрон. учеб.-метод. комплекс по дисциплине в LMS Moodle / Минобрнауки России, Самар. гос. аэрокосм. ун-т им. С. П. Королева (нац. исслед. ун-т); авт.-сост. О. А. Дегтярева. - Электрон. текстовые и граф. дан. - Самара, 2012. – 1 эл. опт. диск (CD-ROM).

В состав учебно-методического комплекса входят:

1. Курс лекций.
2. Задания к лабораторным работам.
3. Задания к практическим работам.
4. Темы для подготовки к зачету.
5. Темы для подготовки к экзамену.
6. Электронные тесты.
7. Рабочая программа.

УМКД «Основы программирования» предназначен для студентов факультета информатики, обучающихся по направлению подготовки бакалавров 010300.62 «Фундаментальная информатика и информационные технологии» в 1 и 2 семестрах.

УМКД разработан на кафедре программных систем.

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Самарский государственный аэрокосмический университет
имени академика С.П. Королёва (национальный исследовательский
университет)» (СГАУ)

Факультет информатики
Кафедра программных систем

Дегтярева О.А.

Курс лекций
«ОСНОВЫ ПРОГРАММИРОВАНИЯ»

Учебное пособие

для студентов, обучающихся по направлению 010300.62
«Фундаментальные информатика и информационные
технологии»

Самара 2012

Содержание

Тема 1. Платформа .NET. 4
Понятие платформы .NET 4
Структура Microsoft.NET Framework 4
Среда Common Language Runtime 5
Преимущества платформы .NET 8
Тема 2. Синтаксис языка C#. 10
Алфавит. 10
Типы данных. 13
Переменные. 16
Именованные константы. 18
Операции. 18
Приведение типов 20
Простейший ввод/вывод 21
Класс математических функций Math 24
Операторы языка 24
Тема 3. Объектно-ориентированное программирование. 33
Понятие объекта и класса. 33
Основные принципы ООП. 33
Состав класса. 34
Поля. 35
Константы. 36
Методы. 37
Ключевое слово this. 39
Конструкторы. 39
Тема 4. Состав класса в языке C#. 43
Свойства. 43
Индексаторы. 46
Параметры методов. 48
Операции класса. 50
Деструктор. 54
Вложенные типы данных. 55
Метод Main. 55
Тема 5. Стандартные классы. 56
Массивы. 56
Символы 62
Строки 63
Класс Random/ 67
Класс Object. 68
Тема 6. Исключительные ситуации. 69
Оператор try. 70
Генерирование исключений. 72
Класс Exception. 74
Пользовательские исключения. 74

Тема 7. Реализация принципов ООП в C#. 76
Наследование. 76
Полиморфизм в C#. 80
Бесплодные классы. 82
Интерфейсы. 84
Стандартные интерфейсы. 89
Перегрузка операций отношения 93
Тема 8. Особые классы. 98
Структуры. 98
Перечисления. 99
Тема 9. Коллекции. 102
Коллекция ArrayList. 103
Тема 10. Объекты файловой системы. 105
Тема 11. Схема потоков данных. 108
Байтовые потоки данных. 108
Текстовые потоки данных. 111
Двоичные потоки-обертки 113
Сериализация объектов. 115
Тема 12. Делегаты. События. 117
Делегаты. 117
Делегаты в параметрах методов 119
Операции с делегатами 119
События 120
Тема 13. Программирование под Windows. 123
Класс Control. 124
Обработка событий. 126
Класс Form. 128
Диалоговые окна. 129
Класс Application. 130
Введение в графику. 131

Тема 1. Платформа .NET.

Понятие платформы .NET

Под платформой Microsoft.NET следует понимать интегрированную систему (инфраструктуру) средств разработки, развертывания и выполнения сложных распределенных программных систем.

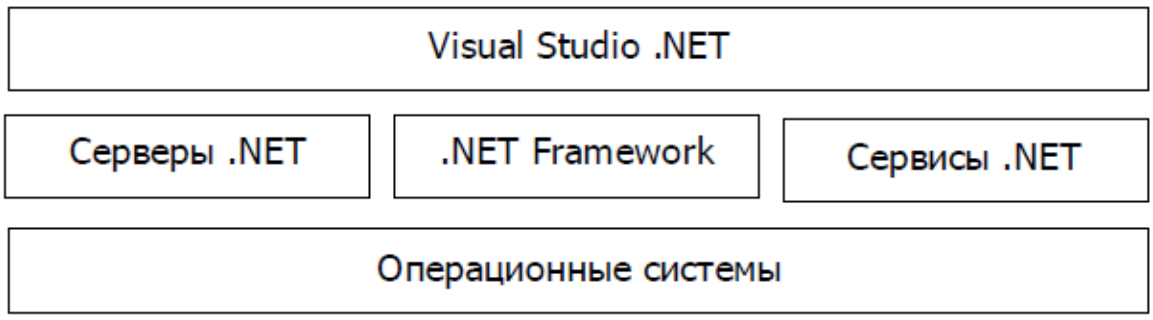


Рисунок 1 Платформа Microsoft.NET

Платформа .NET состоит из нескольких основных компонентов (см. рис. 1):

− операционные системы корпорации Microsoft (Windows 2000/XP/ME/CE), представляющие собой базовый уровень платформы .NET,

− серверы MS.NET (.Net Enterprise Servers) являются программными продуктами корпорации Microsoft, использование которых позволяет снизить сложность разработки сложных программных систем. В числе готовых для применения серверы Application Center 2000, Exchange Server 2000, SQL Server и др.,

− сервисы MS.NET (.Net Building Block Services) представляют собой готовые "строительные блоки" сложных программных систем, которые могут быть использованы через Интернет как сервисные услуги. Набор таких сервисов MS.Net планируется последовательно расширять. Примером имеющегося сервиса платформы MS.Net является Microsoft Passport, позволяющий установить единое имя пользователя и пароль на всех сайтах, поддерживающих аутенфикацию через Passport,

− интегрированная среда разработки приложений Visual Studio.NET – верхний уровень MS.NET - обеспечивает возможность создания сложного ПО на основе платформы и продолжает в этом плане ряд разрабатываемых корпорацией Microsoft средств разработки профессионального программного обеспечения.

Центральной частью платформы .NET является подсистема Microsoft.NET Framework.

Структура Microsoft.NET Framework

Подсистема MS.NET Framework является ядром платформы .NET, обеспечивая возможность построения и исполнения .NET приложений.

На верхнем уровне рассмотрения в составе .NET Framework могут быть выделены (см. рис. 2) общеязыковая среда выполнения (Common Language

Runtime или CLR) и библиотеки классов подсистемы MS.NET Framework. По своему функциональному назначению в составе библиотек классов могут быть выделены:

− набор базовых классов, обеспечивающих, например, работу со строками, ввод-вывод данных, многопоточность и т.п.,

− набор классов для работы с данными, предоставляющих возможность использования SQL-запросов, ADO.NET и обработки XML данных,

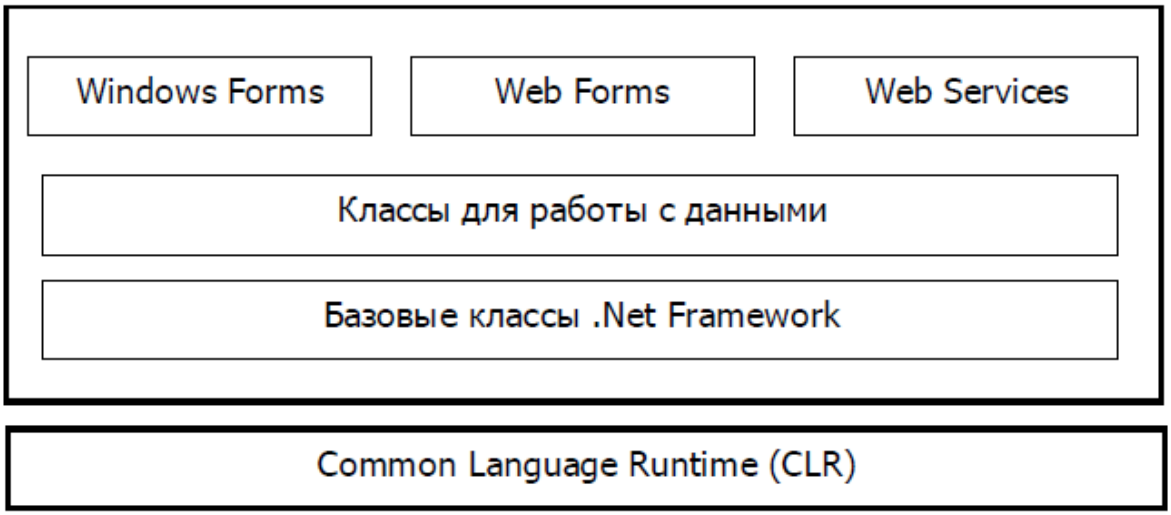


Рисунок 2 Архитектура MS.NET Framework

− набор классов Windows Forms, позволяющих создавать обычные Windows-приложения, в которых используются стандартные элементы управления Windows,

− набор классов Web Forms, обеспечивающих возможность быстрой разработки Web-приложений, в которых используется стандартный графический интерфейс пользователя,

− набор классов Web Services, поддерживающих создание распределенных компонентов-сервисов, доступ к которым может быть организован через Интернет.

Базовый уровень подсистемы MS.NET Framework (см. рис. 2) составляет общеязыковая среда выполнения (Common Language Runtime или CLR).

Среда Common Language Runtime

Общеязыковая среда выполнения CLR обеспечивает исполнение программного кода и является в этом плане основой платформы .NET. Среда CLR активизирует исполняемый код, выполняет для него проверку безопасности, располагает этот код в памяти и исполняет его. Важной частью работы среды CLR является управление свободной памятью, автоматически обеспечивая использование освобождающейся при работе программ памяти (сборку мусора).

Для представления общей схемы функционирования среды CLR дадим ряд важных для платформы .NET понятий и определений:

Для обеспечения возможности многоязыковой разработки ПО программный код, получаемого после компиляции программы на одном из алгоритмических языков платформы .NET, представляется на специально разработанным в корпорации Microsoft общем промежуточном языке (Common Intermediate Language или CIL). Этот язык, с одной стороны, достаточно близок к машинно-зависимым языкам – ассемблерам, с другой стороны, CIL обеспечивает некоторый более высокий уровень представления различных компьютерных платформ. Как результат, программа на языке CIL остается платформенно-независимой, однако требует некоторой дополнительной настройки (компиляции) перед началом своего выполнения.

Программные файлы на языке CIL, получаемые после компиляции программ на алгоритмических языках платформы .NET, называются сборками (assembly), другое более технические наименование сборок - переносимые исполняемые файлы (Portable Executable или PE). Сборки являются основными структурными элементами разрабатываемого программного обеспечения. В конструктивном плане, сборки являются файлами с расширениями exe или dll и состоят из непосредственно программного кода на языке CIL и дополнительных служебных данных, именуемых в .NET метаданными (в составе метаданных необходимая информация о сборке – сведения о типах, данные о версии, ссылки на внешние сборки и т.п.),

Как уже отмечалось выше, сборки перед своим исполнением должны пройти определенную настройку для работы в условиях конкретной выбранной платформы – для выполнения таких настроек в составе среды CLR имеется ряд JIT-компиляторов (Just-In-Time compilers), вызываемых для перевода программного кода на промежуточном языке (CIL-кода) в машинный (native) код платформы исполнения.

С учетом введенных понятий представим общую схему исполнения сборок в среде CLR (см. рисунок 3):

− При запуске сборки загрузчик распознает .NET приложение и передает управление среде CLR, далее загрузчик классов находит и загружает класс, в котором содержится точка начала работы сборки (обычно функция Main), затем CLR передает управление сборке.

− Загрузчик классов выполняется каждый раз при первом обращении к необходимым для выполнении сборки классам. Загрузчик находит нужный класс, размещает информацию о типах класса в кэше и загружает класс для выполнения (при этом в методах загружаемого класса делается отметка, что они не являются еще готовыми для выполнения, поскольку не прошли через JIT-компиляцию).

− Верификатор является частью JIT-компилятора и отвечает за проверку корректности CIL-кода и метаданных. Подобный контроль повышает надежность работы программ, с другой стороны, верификация может и не осуществляться (например, для доверенного кода).

Рис. 3. Среда CLR: общая схема исполнения сборок

− JIT-компиляторы вызываются при обращении к CIL-коду, который ранее при работе программы еще не компилировался. В результате компиляции получается машинный код, оптимизированный для выбранной платформы, для откомпилированного метода устанавливается адрес полученного машинного кода и управление передается исполняемой сборке. Как результат, компиляция методов классов осуществляется только в момент первого к ним обращения (и, тем самым, не используемые методы классов останутся неоткомпилированными). Подобная схема компиляции по мере необходимости может существенно снизить размеры порождаемого программного кода и сократить время подготовки сборки для выполнения, вместе с этим, в среде CLR может быть выполнена и полная компиляция сборки перед началом исполнения (pre-JITing).

− Непосредственное исполнение машинного кода также происходит при активном взаимодействии со средой CLR. Функции среды выполнения (см. рисунок 3) состоят в управлению свободной памятью, обработке исключений, поддержке безопасности и др.

Новый язык программирования C# разработан компанией Microsoft с учетом основных положений технологий Microsoft.NET и вбирает в себя

положительный опыт практического использования языков C++ и Java. После своего создания язык C# прошел серьезную проверку, поскольку Microsoft использовал C# для разработки большей части базовых классов и утилит платформы .NET.

Преимущества платформы .NET

В ходе последовательного развития новых методов, средств и подходов разработки сложного обеспечения регулярно возникали моменты обобщения и интеграции, когда появлялись решения, органично вбирающие в себя последние достижения в области науки и практики программирования. Эти знаковые решения (заслуженно иногда называемые революционными открытиями) приводили к подъему мира программирования на качественно иной уровень состояния дел. Так происходило при разработке языков программирования Pascal и C, создании операционных систем Unix и Windows, отработке принципов объектно-ориентированного программирования и их реализации в языке программирования C++, применении интегрированных и визуальных средств разработки, появлении Интернет и Java, и т.д. Так произошло и при создании новой платформы .NET для разработки, развертывания и выполнения сложного программного обеспечения.

Таким образом, первое, что характеризует новое предложение корпорации Microsoft, - это современность используемых в рамках платформы решений. Платформа .NET, наряду с наличием многих новаторских решений, вбирает в себя самые передовые технологии разработки масштабного ПО. Можно сказать, что для получения .NET была выполнена переплавка всей лучшей "руды" информационной индустрии, в результате чего удалось получить надежную современную основу производства и использования сложных программных систем.

Следует отметить и ряд других ключевых моментов, характеризующих значимость появления платформы .NET (для понимания приводимых далее характеристик в целом достаточно изложенного ранее материала, для обоснования же отдельных утверждений необходимо более глубокое проникновение в технологию .NET):

- Современные средства разработки - платформа .NET включает в себя как готовые компоненты для построения ПО, так и интегрированную среду разработки, обеспечивающая возможность многоязыковой разработки программных систем с использованием разных языков программирования (C#, C++, VBasic.Net, Java# и др.). Как результат, разработчик программ уже не ограничивается выбором одного какого-либо языка программирования, а может варьировать средства разработки с учетом собственного опыта и свойств разрабатываемых программ даже в пределах одной программной системы. Следует отметить также, многие общие части (типы данных, обработка исключений, библиотеки) разных языков программирования являются одинаковыми,

- Компонентное представление ПО – MS.Net развивает существующие подходы к основному способу снижения сложности ПО - компонентному представлению программных систем - предлагая более простой, удобный и надежный метод формирования программных компонент.

- Распределенные вычисления – использование платформы MS.Net в значительной степени снижает сложность современной формы разработки ПО в виде распределенных программных систем или клиент-серверных приложений.

- Интернет технологии - платформа MS.Net содержит большинство существующих Интернет технологий, обеспечивающую возможность быстрой разработки как обычных Web-приложений, так и Web-сервисов, выступающих как доступные через Интернет "строительные блоки" современного сервис-ориентированного программного обеспечения и др.

Можно сказать, что в мире программирования после появления технологии .NET появилась новая знаковая отметка – индустрия разработки ПО до и после появления .NET.

Тема 2. Синтаксис языка C#.

Изучение любого языка начинается с рассмотрения тех символов, из которых составляются языковые конструкции и тексты. В том числе, и языка программирования. Из символов языка программирования затем составляются исполняемые программы. В языке C# для кодирования символов используется таблица Unicode.

Алфавит.

Символы языка C# делятся на следующие группы:

- буквы (латинские и национальных алфавитов) а также символ подчеркивания «_», который используется в качестве буквы;
- цифры;
- специальные символы, например «-», «+», «’» и т.д.;
- пробельные символы (пробел и табуляция);
- символы перевода строки.

Из символов составляются следующие конструкции: лексемы, директивы препроцессора и комментарии.

Директивы препроцессора достались по наследству языку C# от языка C++ и не являются предметом изучения данного курса

Комментарии.

Комментарий – это способ пояснения кода программы. При компиляции комментарии игнорируются и исключаются из исполняемого кода.

Комментарии в языке делятся на два типа – однострочные и многострочные. Как следует из названия, однострочный комментарий представляет собой единственную строку до конца:

```
// Это текст однострочного комментария.
```

Многострочный комментарий представляет собой несколько строк:

```
/*Это текст
многострочного
комментария*/
```

Многострочный комментарий может также содержать только часть строки.

Комментарии не могут быть вложенными и могут содержать любой текст.

Существует еще один вид многострочного комментария – так называемый комментарий документирования, предназначенный для формирования документации к программе в формате XML. Он начинается с комбинации символов «///».

Лексемы.

Лексема (или слово) – это минимальная единица языка, имеющая собственный смысл. В C# выделяют следующие типы лексем:

- идентификаторы;
- ключевые слова;
- символы операций;
- разделители;
- литералы (константы).

Идентификаторы.

Идентификатор – это имя какого-либо объекта программы. Он служит для обращения к этому объекту в программном коде. Правила составления идентификаторов:

идентификатор может состоять только из букв (плюс символ «_») и цифр. В языке различается регистр символов, поэтому идентификаторы «a» и «A» интерпретируются как два разных имени;

- первым символом в идентификаторе может быть только буква или «_»;
- количество символов в идентификаторе не регламентируется;
- пробелы в идентификаторе не допускаются;
- допустимо использование букв национальных алфавитов, хотя это и не приветствуется;
- совпадение идентификатора с ключевыми словами не допускается.

Существует также несколько правил по именованию, относящихся к хорошему стилю программирования:

- имена должны быть понятными и соответствовать контексту использования;
- имена составляются из слов, описывающих контекст использования именуемого объекта;
- имя должно соответствовать одной из принятых нотаций – правилу создания имён.

В соответствии с нотацией Паскаля каждое слово в идентификаторе начинается с большой буквы, например, CountSqareOfRectangle, PrintArray.

В соответствии с нотацией Camel каждое слово в идентификаторе начинается с большой буквы, кроме первого, например, countSqureOfRectangle, printArray.

Данные две нотации наиболее часто используются в языке C#.

В венгерской нотации в начало имени приписывается префикс, представляющий собой первый символ названия типа, которому принадлежит именуемый объект, например, dCountSqareOfRectangle, vdPrintArray.

Также существует несколько соглашений, в соответствии с которыми идентификатор составляется из слов, записанных большими буквами, и

разделенных символом подчеркивания, например, PRINT_ARRAY; из слов, записанных маленькими буквами, и разделенных символом подчёркивания, например, print_array; идентификатор начинается с символа подчеркивания, например, _printArray.

Ключевые слова.

Ключевые слова – это идентификаторы, которые являются зарезервированными и имеют специальное значение. Их около 100 штук, с ними мы будем знакомиться по ходу изучения данного курса.

Символы операций и разделители.

Операция – это действие над операндами. Операции обозначаются одним или несколькими символами. Операции могут записываться как с помощью специальных символов, так буквенным обозначением (такие операции являются ключевыми словами). Операции делятся на: унарные – над одним операндом, бинарные – над двумя операндами, тернарные – над несколькими операндами. Операция представляет собой отдельную лексему, то есть внутри операции пробелы не допускаются, исключениями являются операции ( ), [ ] и ? : .

Разделители предназначены для отделения элементов друг от друга. Или, наоборот, группирования элементов в единую конструкцию. Разделители также записываются специальными символами, например, «.», «,», «(», «)».

Литералы.

Литералами называют величины-значения, например, число 10, или строку «Мама мыла раму». Поскольку такие значения нельзя изменить, то литералы часто называют константами. Литералы бывают:

- логические – true и false;
- целые – в десятичном или шестнадцатеричном коде, например, 10 или 0xA;
- вещественные – записываются как в десятичной, так и в экспоненциальной форме, например 31.5, 0.315e2;
- символьные – символ, заключенный в «’»;
- строковые – строка, заключенная в «”»;
- null – константа, означающая отсутствие объекта (пустая ссылка).

Литералы вычленяются из текста программы и относятся к какому-либо типу данных в соответствии со своим внешним представлением прямо на этапе компиляции программы.

Символьные литералы могут быть представлены в следующих формах:

- символ, имеющий графическое отображение – например, ’a’ или ’0’;
- символ, записанный своим 16-ричным кодом – например, ’\xA’ ’\x0F’. Шестнадцатеричный код символа начинается с префикса

«\x», далее сразу записывается код символа, числовое значение которого должно находиться в диапазоне от 0 до $2^{16} – 1$;

- управляющая последовательность (escape-последовательность) – символ, или набор символов, предваренный «\». Например, ’\a’ – звуковой сигнал, ’\b’ – возврат на символ, ’\n’ – перевод строки, ’\r’ – возврат в начало строки, ’\t’ – горизонтальная табуляция, ’\v’ – вертикальная табуляция, ’\\’ – символ «\», ’\’’ – символ «’», ’\”’ – символ «”», ’\0’ – ноль-символ, используемый в строковых значениях;
- escape-последовательность в кодировке Unicode – например, ’\uA0F’, ’\U10’. Такая escape-последовательность представляет символ в кодировке Unicode с помощью его 16-ричного кода с префиксом «\u» или «\U».

Управляющая последовательность интерпретируется как один символ, поэтому она может быть включена в состав строкового литерала. Например, строковый литерал ”Мама\nмыла\nраму” при выводе на экран будет выглядеть следующим образом:

```
Мама
мыла
раму
```

Существуют также так называемые «дословные строковые литералы», снабженные префиксом «@». Для такого литерала отключена обработка управляющих последовательностей. То есть строка @”Мама\nмыла\nраму” будет представлена на экране как

```
Мама\nмыла\nраму
```

Дословные литералы удобно использовать тогда, когда они содержат большое количество символов, представимых только с помощью управляющей последовательности – символов «\», «”» или «’», например, пути к файлам, или тексты с названиями. Тогда содержимое дословного строкового литерала освобождается от лишних префиксов «\», обозначающих начало управляющей последовательности, что улучшает читабельность текста:

```
”C:\\MyDirectory\\MyFile.cs”
@”C:\MyDirectory\MyFile.cs”
```

Содержимое строкового литерала может быть пустым (обозначается «””»), а вот пустой символьный литерал не допускается.

Типы данных.

Программа – это набор действий по обработке некоторых данных. Все данные хранятся в оперативной памяти, связанной с программой. При описании данных необходимо указывать, каким образом они представляются

в памяти, сколько памяти занимают, каков допустимый диапазон значений, какие действия над ними можно выполнять. Эту информацию содержит описание типа, которому принадлежат данные. Поэтому в программе не может быть величин, не принадлежащих какому-либо типу данных.

Существует несколько классификаций типов данных – встроенные и пользовательские, статические и динамические, простые и структурированные, значащие и ссылочные. Классификации естественно являются перекрестными. Объем динамических данных определяется в процессе работы программы, а статических – на этапе их описания. Структурированные типы агрегируют группу данных простых и структурированных типов. Встроенные (или стандартные) типы данных относятся к ядру языка программирования, а пользовательские – определяются программистом, как правило, на основе стандартных. Данные значащих типов хранятся в статической памяти (stack – стэк), а ссылочных – в динамической памяти (heap - куча). В языке C# основной памятью для хранения данных является хип.

Начнем с рассмотрения типов данных, относящихся к ядру языка C# - встроенных или стандартных типов данных.

Встроенные типы данных

Для каждого стандартного типа существует свое ключевое слово, которое и используется для типизирования данных – то есть отнесения данных к определенному типу. Встроенные типы данных для языка C# представлены в таблице 1.

Таблица 1. Встроенные типы данных

Тип данных (ключевое слово)	Описание	Стандартный класс библиотеки .NET	Занимаемый объем памяти, байт	Диапазон возможных значений
bool	Логический тип	Boolean	1	true, false
Целые типы данных				
sbyte	Байт со знаком	SByte	1	-128..127
byte	Байт без знака	Byte	1	0..255
short	Короткое целое со знаком	Int16	2	-32768..32767
ushort	Короткое целое без знака	UInt16	2	0..65535
int	Целое со знаком (или просто целое)	Int32	4	$-2*10^9..2*10^9$

uint	Целое без знака	UInt32	4	$0..4*10^9$
long	Длинное целое со знаком	Int64	8	$-9*10^{18}..9*10^{18}$
ulong	Длинное целое без знака	UInt64	8	$0..18*10^{18}$
char	Символьный (код Unicode-символа)	Char	2	$0000..FFFF_{16}$
Вещественные типы данных				
float	Вещественное число	Single	4	-45..38 (указана степень числа 10)
double	Вещественное число с двойной точностью	Double	8	-324..308 (указана степень числа 10)
decimal	Финансовый тип (предназначен ля денежных вычислений)	Decimal	16	$10^{-28}..7{,}9*10^{28}$
string	Строка	String	Длина не ограничена	Состоит из символов
object	Базовый объектный тип	Object		
enum	Перечисление			
struct	Структура			

Пользовательский тип после полного описания может использоваться наравне со встроенными.

Следует отметить, что каждому встроенному типу языка C# в обязательном порядке соответствует стандартный класс библиотеки .NET. Это значит, что имя типа можно без видимых последствий заменить на имя соответствующего класса. А это в свою очередь означает, что у встроенных типов данных есть функциональность, присущая понятиям класса и объекта.

Литералы также обязательно типизируются. Целочисленный литерал по умолчанию относится к целочисленному типу данных, причем из типов int, uint, long, ulong выбирается тот, у которого наименьший диапазон возможных значений, при этом значение целочисленного литерала конечно же должно попадать в этот наименьший диапазон. Вещественные литералы

по умолчанию всегда типизируются как double. Для принудительной типизации используются специальные суффиксы (регистр суффиксов не важен): u – uint, l – long, ul (lu) – ulong, f – float, d – double, m – decimal. Например: 10d (литерал 10 теперь принадлежит типу double, а не int); 2.5f (литерал 2.5 принадлежит типу float, а не double). Принудительная типизация используется для уменьшения количества преобразований типов.

Значащие и ссылочные типы данных.

Перечисленные встроенные типы данных (кроме object и string) относятся к значимым типам. То есть память для их хранения выделяется на этапе компиляции в стэке. Типы object и string относятся к ссылочным типам данных, то есть величина хранит только ссылку на данные, а сами данные хранятся в хипе. Все объектные типы данных являются ссылочными.

При присваивании величины значащего типа копируются сами данные, а при присваивании величины ссылочного типа копируется ссылка на данные. При сравнении величин значащего типа сравниваются хранимые данные, то есть величины равны, если они хранят одинаковые данные. При сравнении величин ссылочного типа сравниваются ссылки на данные, то есть величины равны, если они ссылаются на одни и те же данные. На рисунке 1 величины `a` и `b` являются значимыми и равны между собой, поскольку хранят одинаковые значения. Величины `c`, `d` и `e` относятся к ссылочному типу. Величина `d` равна величине `e`, поскольку они ссылаются на одни и те же данные. А вот величины `c` и `e` не равны, так как ссылаются на разные данные, даже несмотря на то, что содержимое этих данных одинаково.

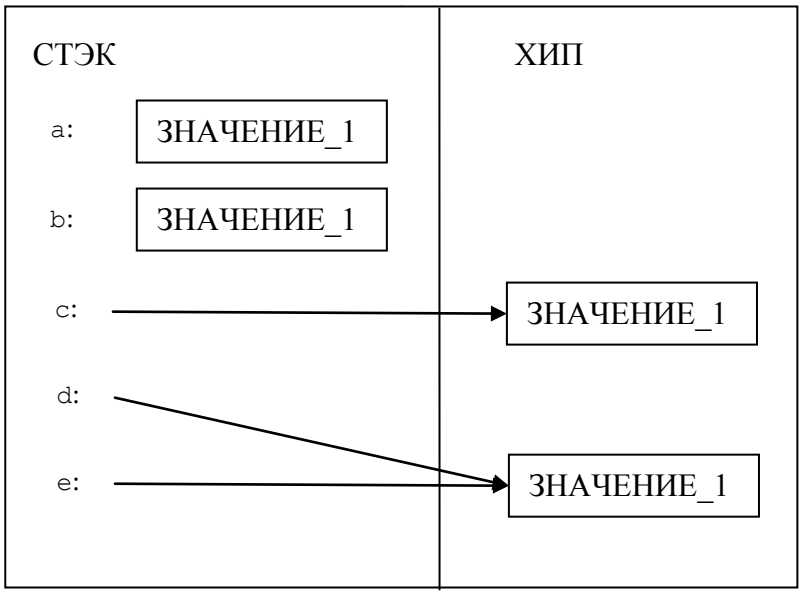


Рисунок 1. Значимые и ссылочные типы.

Переменные.

Переменная – это область памяти, имеющая имя, предназначенная для хранения данных определенного типа. Хранимые в этой области памяти

данные доступны для чтения и изменения по имени переменной. Все переменные в программе должны быть описаны, при описании указывается тип переменной, ее имя и, при необходимости, её начальное значение (инициализация). Например:

```
double a; //объявление переменной вещественного типа
int a = 2;//объявление и инициализация переменной целого типа
```

Имя для переменной задается разработчиком и должно соответствовать всем правилам составления идентификаторов, описанным выше. Для инициализации переменной можно использовать не только литерал, но и выражение. Только операнды в выражении должны быть вычислимы к этому моменту, чтобы и само выражение было вычислимым.

Переменные можно описывать в том месте программы, где появилась необходимость в объявлении переменной. Область действия (видимости, существования) переменной начинается с момента ее объявления и заканчивается с окончанием соответствующего блока кода. Блок кода представляет собой группу операторов, обрамленных «{» и «}». Блок кода интерпретируется как один оператор, и переменная действует внутри того блока, в котором объявлена. Блоки могут быть вложенными. Имя переменной должно быть уникальным во всей области ее действия. Приведем пример, иллюстрирующий области действия переменных.

```
…
{                   //начало блока 1
  int a;            //объявление переменной a
  int b;            //объявление переменной b
  {                 //начало вложенного блока 2
    int c           //объявление переменной c
    int a;          //объявление переменной a НЕДОПУСТИМО
  }                 //конец вложенного блока 2
  {                 //начало вложенного блока 3
    int c;          //объявление переменной c
  }                 //конец вложенного блока 3
}                   //конец блока 1
…
```

В блоке 1 объявлены две переменных целого типа – `a` и `b`. Блок 2 является вложенным в блок 1, поэтому объявление переменной `c` корректно, но объявление новой переменной `a` внутри блока 2 недопустимо, в нем уже есть переменная `a`, объявленная в блоке 1 и действующая до его окончания. Блок 3 также является вложенным в блок 1, но не зависит от блока 2, поэтому переменная `c`, объявленная в блоке 3, не имеет никакого отношения к переменной `c`, объявленной в блоке 2. Конфликта имен в этом случае не возникает, и обе переменные `c` объявлены корректно.

Именованные константы.

В некоторых случаях в программе необходимо запретить изменение значений какой-либо переменной. Для этого используется ключевое слово const.

```
const int a = 10;
const float b = 22.5, c = 3.6;
```

Во второй строке кода тип `float` и ключевое слово `const` относится и к переменной `b` и к переменной `c`.

Такие неизменяемые переменные называются именованными константами. Они должны инициализироваться сразу же при описании. Для инициализации можно использовать как литерал, так и вычислимое выражение.

Операции.

Операции чаще всего являются составной частью выражений. Выражение – это правило, после выполнения которого получается результат, принадлежащий определенному типу. Участвующие в выражении операнды связываются операциями, в выражении могут также участвовать вызовы функций и другие выражения.

Операции делятся на унарные (над одним операндом), бинарные (над двумя операндами) и тернарные (над несколькими операндами). Операции записываются как с помощью специальных символов, так и в буквенном представлении (обозначение такой операции является ключевым словом). Знак операции может состоять из одного или нескольких символов. Внутри знака операции пробелы не допускаются, это правило касается всех операций, кроме ( ), [ ] и ? : . Каждая операция имеет свой приоритет выполнения, как в математике. Порядок выполнения операций в выражении зависит от их приоритета. Операции одинакового приоритета выполняются слева направо, кроме условной операции и операции присваивания, которые выполняются справа налево. Порядком выполнения операций можно управлять с помощью скобок «(», «)». Операнды в операции всегда вычисляются слева направо. Операции языка C# перечислены в таблице 2.

Таблица 2. Операции языка C#.

Категория	Знак	Описание	Пример
Первичные операции	.	Доступ к элементу	myObj.Member this.Count
	[ ]	Индексирование	s[i]
	( )	Вызов метода	this(a,b)
	a++, a--	Инкремент/декремент постфиксная форма	
	++a, --a	Инкремент/декремент префиксная форма	
	new	Выделение опера-	

		тивной памяти	
	typeof	Определение типа	
	checked (unchecked)	Проверяемый/ непроверямый код	
Унарные операции	+	Унарный плюс	+2
	-	Арифметическое отрицание	-3
	!	Логическое отрицание	!a
	~	Побитовое отрицание	
	(type)a	Приведение типа	
Бинарные операции			
Мультипликатив-ные операции	*	Умножение	
	/	Деление	
	%	Деление по модулю	
Аддитивные операции	+	Сложение	
	-	Вычитание	
Операции сдвига	>>	Побитовый сдвиг вправо	
	<<	Побитовый сдвиг влево	
	>>>	Арифметический сдвиг вправо	
Операции сравнения	<	Меньше	
	>	Больше	
	<=	Меньше или равно	
	>=	Больше или равно	
Операции проверки типа	is	Принадлежность типу	
	as	Приведение типа	
Сравнение на равенство	==	Равно	
	!=	Не равно	
Поразрядные логические операции	&	Побитовое И	
	^	Побитовое исключающее ИЛИ	
	|	Побитовое ИЛИ	
Условные логические операции	&&	Логическое И	
	||	Логическое ИЛИ	
Условная операция (тернарная операция)	? :	Проверка условия	
Присваивания	=		a = 2;

	*=, -=, +=, ...	Присваивание с действием	a*=2 ⇔ a=a*2

В таблице 2 операции сгруппированы по приоритетам. Только в условных поразрядных и условных логических операциях приоритет уменьшается в следующем порядке – И, ИЛИ с исключением, ИЛИ.

Приведение типов

В одном выражении должны участвовать операнды одного типа. Именно к этому типу и относится результат выражения. Однако чаще всего в одном выражении соседствуют операнды различных типов. При этом для некоторых из них операция может быть вообще не определена. То есть возникает необходимость в так называемом приведении (или преобразовании) типа. Существует два типа преобразований типа – явное и неявное. Неявное преобразование выполняется автоматически, без участия разработчика, конечно, если это возможно. Как правило, неявно преобразуется более короткий тип к более длинному, поскольку это не приводит к потере данных и точности. Схема допустимых неявных преобразований для арифметических значащих типов данных представлена на рисунке 2.

Когда в выражении встречаются операнды разных типов, то они неявно приводятся к самому длинному типу, встречающемуся в выражении. Конечно, если такое преобразование возможно. Если пути преобразования нет (см. рисунок 2), то возникает ошибка компиляции.

Следует особо отметить, что для типов, короче `int` арифметические операции не определены. Если в арифметическом выражении встречаются операнды таких типов, то они неявно преобразуются к `int`.

Если неявное преобразование невозможно, то можно применить операцию явного преобразования, упомянутую в таблице 2.

```
double b = 5.2;
int a = (int)b;  //Неявное преобразование
                 //из double в int невозможно,
                 //применим явное
```

При применении явного преобразования может возникнуть потеря данных, поскольку преобразуется более длинный тип к более короткому. Ответственность за это несет разработчик, применяющий явное преобразование на свой страх и риск. Особенно часто неявные и явные преобразования используются при работе со ссылочными типами данных.

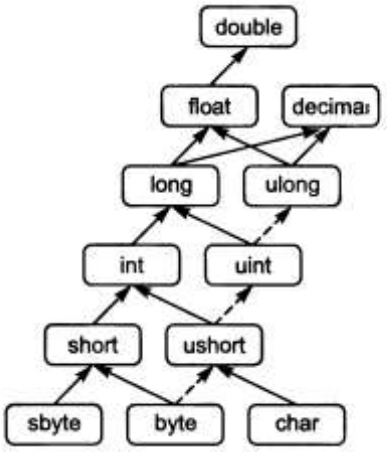


Рисунок 2. Схема неявных преобразований встроенных типов данных

Простейший ввод/вывод

К простейшему вводу/выводу отнесем взаимодействие пользователя во время работы программы с устройствами ввода/вывода, которые в совокупности называются консолью (клавиатура и экран). В языке нет операторов ввода или вывода. Для этого используется стандартный системный объект, описанный классом Console, находящийся в пространстве имен System. В классе Console описаны методы для ввода и вывода информации. Рассмотрим пример:

```
class Program
{
    static void Main()
    {
       int a = 10;
       float x = 3.14, y = 6.28;
       string s = ”Мама мыла раму”;
       Console.WriteLine(s); // вывод 1
       Console.WriteLine(“a = “ + a); // вывод 2
       Console.WriteLine(“x = “ + x + “; y = ”+ y); // вывод 3
    }
}
```

Результатом работы данной программы является следующий вывод на экран:

```
Мама мыла раму
a = 10
x = 3.14; y = 6.28
```

Метод `WriteLine` выводит строковое значение, переданное в скобках, на экран и переводит курсор на следующую строку (вывод 1). В примере (вывод 2) на экран выводится строка, составленная из конкатенации строки `“a = “` и строкового представления значения переменной `a`. Для всех встроенных типов данных определено преобразование значения в строковое представление – строку. Это преобразование вызывается автоматически, когда необходимо (при конкатенации с другой строкой,

например). Такое преобразование может быть инициировано и явным образом:

```
string s = a.ToString();
```

Однако для встроенных типов данных в этом нет необходимости. Оператор, помеченный комментарием (вывод 3) выводит на экран строку, являющуюся результатом склеивания четырех строк – строки "x = ", строкового представления значения переменной x, строки "; y = " и строкового представления значения переменной y.

В классе Console описано несколько методов с именем WriteLine, предназначенных для вывода параметров различных типов. Такая группа методов с одинаковыми именами и различными параметрами называется перегруженным методом.

Одной из перегрузок является так называемый форматный вывод:

```
int x = 2, y = 4;
Console.WriteLine("x = {0}; y = {1}",x,y);
```

Набор параметров в скобках делится на две части – строка формата и список вывода (набор элементов, разделенных «,»). Строка формата кроме обычных символов содержит параметры формата в фигурных скобках. Параметры формата нумеруются с 0. Номера могут следовать в произвольном порядке. При выводе в строку формата подставляется строковое представление соответствующего по номеру элемента из списка вывода. Результат работы приведенного примера будет выглядеть следующим образом:

```
x = 2; y = 4
```

Если в списке вывода перечислено больше элементов, чем указано в параметрах формата, то неиспользуемые элементы просто не включаются в строку формата. А вот если в строке формата в качестве параметра указан номер элемента из списка вывода, которого в этом списке вывода нет – возникнет ошибка компиляции.

Для параметров формата в фигурных скобках можно задать ширину поля и формат вывода (для чисел):

{номер элемента из списка вывода,ширина поля:формат вывода}

```
{0,5:0.##0}
```

Если значению требуется больше позиций, чем указано в ширине поля, то этот параметр игнорируется. После задания ширины поля можно задать формат вывода. Его задание не является обязательным и имеет смысл только для числовых данных. На месте цифры 0 всегда отображается соответствующая цифра числа, на месте символа # отображаются все цифры числа, кроме 0.

Метод Write также предназначен для вывода строковой информации на экран. Он аналогичен методу WriteLine и отличается лишь тем, что не переводит курсор на следующую строку.

В классе Console нет методов, считывающих непосредственно число с клавиатуры, а только строку или символ. Поэтому ввод значения другого типа данных выполняется методом ReadLine и состоит из двух этапов:

- символы с клавиатуры считываются в строковую переменную;
- полученная строковая переменная преобразуется к значению искомого типа.

Преобразование строки в другой тип данных можно выполнить методами стандартного класса Convert из пространства имен System. Второй способ – использовать метод Parse, описанный в каждом стандартном классе простых встроенных типов данных.

Рассмотрим пример:

```
class Program
{
    static void Main()
    {
       string s = null;
       Console.WriteLine("Введите строку");
       s = Console.ReadLine();
       Console.WriteLine("Вы ввели строку: " + s);
       Console.Write("Введите целое число ");
       s = Console.ReadLine();
       int a = Int32.Parse(s);
       Console.WriteLine("Вы ввели число: " + a);
       Console.Write("Введите вещественное число ");
       s = Console.ReadLine();
       double x = Convert.ToDouble(s);
       Console.WriteLine("Вы ввели число: " + x);
    }
}
```

Метод ReadLine считывает все символы из буфера клавиатуры до символа Enter в строковую переменную. При этом символ перевода строки удаляется из буфера, а сам буфер клавиатуры очищается. Если преобразование считанных символов к нужному типу возможно, то оно будет выполнено. Если преобразование невозможно, то возникнет ошибка исполнения программы.

Метод Read предназначен для считывания одного символа из буфера клавиатуры. Результат выполнения метода имеет тип int (метод возвращает значение -1, если символов в буфере нет), поэтому требуется явное преобразование к типу char. К тому же, метод Read не очищает буфер клавиатуры. А ввод данных с клавиатуры в буфер производится по нажатию клавиши Enter вместе с ее кодом. То есть после выполнения метода буфер может остаться не пустым, и следующий метод Read считает следующий

символ из буфера, не ожидая его ввода. Для принудительной очистки буфера клавиатуры можно применить метод `ReadLine` и не сохранять результат его выполнения. Рассмотрим иллюстрирующий пример.

```
class Program
{
    static void Main()
    {
      char c = null;
      Console.WriteLine("Введите символ");
      c = (char)Console.Read(); //считывание одного символа
      Console.ReadLine(); //очистка буфера клавиатуры
      Console.WriteLine("Вы ввели символ: " + c);
    }
}
```

Класс математических функций Math

Математические функции описаны в специальном классе Math из пространства имен System. В этом классе описаны:

- тригонометрические функции: Sin, Cos, Tan;
- обратные тригонометрические функции: ASin, ACos, ATan, ATan2;
- гиперболические функции: Tanh, Sinh, Cosh;
- экспонента и логарифмические функции: Exp, Log, LoglO;
- модуль (абсолютная величина), квадратный корень, знак: Abs, Sqrt, Sign;
- округление: Ceiling, Floor, Round;
- минимум, максимум: Min, Max;
- степень, остаток: Pow, IEEEReminder;
- полное произведение двух целых величин: BigMul;
- деление и остаток от деления: Div Rem.
- константы e и π.

Пример вычисления корня квадратного:

```
double a = 4.5;
double b = Math.Sqrt(a);
```

Операторы языка

Каждый оператор языка заканчивается символом «;». Оператор может быть пустым. То есть не выполнять никаких действий. Оператор может быть составным, в этом случае он называется блоком. О нем шла речь в разделе о переменных. Блок может также быть пустым.

Рассмотрим операторы языка (конструкции программирования), которые управляют ходом выполнения программы.

Условный оператор if

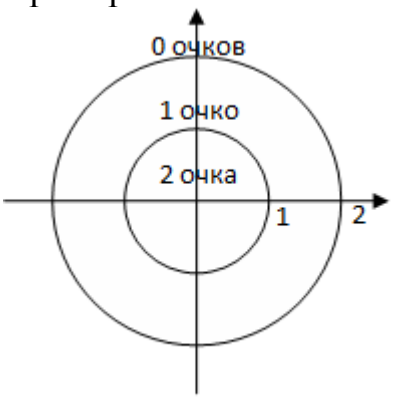
Оператор `if` предназначен для разветвления хода выполнения программы на две ветви в зависимости от выполнения условия.

Синтаксис оператора:

```
if (<условие>) <оператор1>;
else <оператор2>;
```

Сначала проверяется условие. Условие описывается логическим выражением. Если оно истинно, то выполняется оператор1, если ложно – выполняется оператор2. Ветвь `else` может отсутствовать, это значит, что в случае ложности условия не надо выполнять никаких действий. Оператор1 и оператор2 могут быть составными, то есть блоками. После выполнения какой-либо из ветвей будет выполняться оператор, следующий за условным.

Пример: Посчитать количество очков при попадании в мишень.




```
Console.Write(”Введите координату X выстрела: ”);
double x = Convert.ToDouble(Console.ReadLine());
Console.Write(”Введите координату Y выстрела: ”);
double y = Convert.ToDouble(Console.ReadLine());
int num = 0;
if (x * x + y * y < 1) num = 2;
else if (x * x + y * y < 4) num = 1;
Console.WriteLine(”Вы получили {0} очков”, num):
```

Оператор выбора switch

Оператор `switch` (переключатель) предназначен для ветвления алгоритма на несколько направлений.

Синтаксис оператора:

```
switch (<выражение>)
{
  case <константа1>: <операторы1>; break;
  case <константа2>: <операторы2>; break;
  case <константа3>: <операторы3>; break;
  …
  default : <операторы>; break;
}
```

Сначала вычисляется выражение. Результат выражения должен принадлежать одному из перечислимых типов (целый, строковый). Затем в списке `case` ищется то значение константы, которое совпало с результатом выражения, и выполняются операторы соответствующей ветви. Если совпадений в списке `case` не найдено, то выполняются операторы ветви `default`. Ветвь `default` не является обязательной. В том случае, когда совпадений в списке `case` не найдено, и ветвь `default` отсутствует, оператор ничего не выполняет. После выполнения оператора `switch` выполняется следующий за ним оператор. Все константы в списке `case` должны быть неявно приводимы к типу результата выражения. Каждая ветвь в списке `case` и ветвь `default` должны заканчиваться оператором передачи управления (перехода) – `break`, `goto` или `return` (в приведенном примере используется оператор `break`). О них речь пойдет чуть позднее.

Пример: Вывести на экран остаток от деления введенного числа на 5.

```
Console.WriteLine(”Введите целое число”);
int num = Int32.Parse(Console.ReadLine());
switch (num % 5)
{
  case 1: Console.WriteLine(”Остаток = 1”); break;
  case 2: Console.WriteLine(”Остаток = 2”); break;
  case 3: Console.WriteLine(”Остаток = 3”); break;
  case 4: Console.WriteLine(”Остаток = 4”); break;
  default :Console.WriteLine(”Остаток = 0”); break;
}
```

Оператор `switch` позволяет разветвить алгоритм на несколько ветвей, но условие разветвления не должно быть вещественным.

Циклы

Циклические операторы используются для многократного повторения набора операторов. Этот набор операторов называется телом цикла. Тело цикла должно состоять из одного оператора, но этот оператор может быть составным, то есть блоком. Одно выполнение тела цикла называется итерацией. Параметром цикла называется счетчик цикла или переменная, влияющая на условие продолжения выполнения цикла. Проверка условия продолжения выполнения цикла проводится на каждой итерации цикла. Для принудительного выхода из цикла служат уже упомянутые операторы передачи управления.

Цикл с предусловием while

Синтаксис оператора:

```
while (<логическое выражение>)
{
  <тело цикла>;
```

```
    }
```

Тело цикла будет выполняться до тех пор, пока логическое выражение истинно. Если логическое выражение сразу ложно, то цикл не выполнится ни разу. После выхода из цикла выполняется следующий за циклом оператор.

Для корректной работы циклической конструкции необходимо проверить следующее:

- объявление и инициализация параметров цикла (до описания тела цикла);
- корректность условия продолжения выполнения цикла;
- модификация параметров цикла при выполнении тела цикла.

Пример: Организация пользовательского меню.

```
class Program
{
  static void Main()
  {
    Console.WriteLine(”1 – задача 1, 2 – задача 2, 0 – выход”);
    string buf = Console.ReadLine();//инициализация параметра цикла
    while (buf != ”0”)
    {
      switch (buf)
      {
        case ”1”: <Решение задачи 1>; break;
        case ”2”: <Решение задачи 2>; break;
        case ”0”: ;
        default: Console.WriteLine(”Неверный пункт меню”); break;
      }
      Console.WriteLine(”1 – задача 1, 2 – задача 2, 0 – 
выход”);
      buf = Console. ReadLine(); //изменение параметра цикла
    }
  }
}
```

Внесем коррективы в представленный пример, используем для выхода из цикла оператор перехода.

```
class Program
{
  static void Main()
  {
    string buf = null;//объявление и инициализация параметра цикла
    while (true)
    {
      Console.WriteLine(”1 – задача 1, 2 – задача 2, 0 – выход”);
      buf = Console.ReadLine();//модификация параметра цикла
      switch (buf)
      {
        case ”1”: <Решение задачи 1>; break;
        case ”2”: <Решение задачи 2>; break;
        case ”0”: return; //выход из программы
        default: Console.WriteLine(”Неверный пункт меню”); break;
```

```
      }
    }
  }
}
```

Как видим, во втором примере параметр цикла не влияет на условие продолжения выполнения цикла. Для принудительного выхода из цикла, а также и для принудительного завершения программы используется оператор `return`.

Цикл с постусловием do/while

Синтаксис оператора:

```
do
{
  <тело цикла>;
}
while (<логическое выражение>);
```

Цикл завершается, когда логическое выражение становится равным `false`. Цикл с постусловием отличается от цикла с предусловием тем, что сначала выполняется тело цикла, и только потом проверяется условие продолжения выполнения цикла. То есть даже в том случае, когда это условие ложно, цикл точно выполнится один раз. Цикл можно завершить принудительно, используя операторы передачи управления. После завершения цикла выполняется следующий за ним оператор.

Цикл с постусловием удобно использовать в тех случаях, когда тело цикла должно выполниться хотя бы один раз. Например, при вводе данных и последующей их проверке. Или при создании пользовательского меню, когда пользователь должен осуществить выбор пункта меню хотя бы один раз.

Пример: Организация пользовательского меню.

```
…
string buf = null;//объявление и инициализация параметра цикла
do
{
  Console.WriteLine(”1 – задача 1, 2 – задача 2, 0 – выход”);
  buf = Console.ReadLine();//ввод пункта меню пользователем
  switch (buf)
  {
    case ”1”: <Решение задачи 1>; break;
    case ”2”: <Решение задачи 2>; break;
    case ”0”: ;
    default: Console.WriteLine(”Неверный пункт меню”); break;
  }
}
while (buf != ”0”);
…
```

Модифицируем приведенный пример – используем оператор явного перехода:

…

```
string buf = null;//объявление и инициализация параметра цикла
do
{
  Console.WriteLine(”1 – задача 1, 2 – задача 2, 0 – выход”);
  buf = Console.ReadLine();//ввод пункта меню пользователем
  switch (buf)
  {
    case ”1”: <Решение задачи 1>; break;
    case ”2”: <Решение задачи 2>; break;
    case ”0”: return;
    default: Console.WriteLine(”Неверный пункт меню”); break;
  }
}
while (true);
```

Цикл с предусловием for

Синтаксис оператора:

```
for (<секция инициализации>; <логическое выражение>;<секция изменения>)
{
     <тело цикла>;
}
```

Секция инициализации в заголовке цикла предназначена для задания начальных значений переменных, используемых в цикле. Также, в этой секции можно объявить и проинициализировать новые переменные, которые будут существовать только внутри цикла. Объявлять можно только переменные, принадлежащие одному типу. Если в секции инициализации нужно задать значения нескольким разнотипным переменным, то объявить их следует до цикла, а задать для них начальные значения – в заголовке цикла. В общем случае в секции инициализации можно задать значения любым переменным, как использующимся внутри цикла, так и не имеющим к нему отношения.

Логическое выражение аналогично логическому выражению в синтаксисе цикла `while`. Поэтому цикл `for` и относится к циклам с предусловием. Если значение логического выражения истинно, то тело цикла выполняется, если ложно, то цикл завершается. Цикл `for` может не выполниться ни разу, если логическое выражение сразу имеет результат `false`.

Секция изменения предназначена для изменения параметров цикла. Эта секция содержит операторы, которые исполняются при каждой итерации цикла. В общем случае секция изменения может содержать любые исполняемые операторы.

Порядок выполнения цикла `for`:

1) задаются значения переменным, объявленным и проинициализированным в секции инициализации;
2) вычисляется результат логического выражения. Если выражение истинно – выполняется тело цикла, если ложно – цикл завершается;
3) после выполнения тела цикла исполняется секция изменения. Затем переход к пункту 2.

Все три секции в заголовке цикла `for` являются необязательными.

Пример: Табуляция функции `sin(x)` в заданном диапазоне с заданным шагом.

```
…
Console.Write(”Введите xmin:”);
double xmin = Double.Parse(Console.ReadLine());
Console.Write(”Введите xmax:”);
double xmax = Double.Parse(Console.ReadLine());
Console.Write(”Введите шаг табуляции dx:”);
double dx = Double.Parse(Console.ReadLine());
Console.WriteLine(”|---x---|---y---|”);
for (double x = xmin; x < xmax; x +=dx) // табуляция функции
{
  Console.WriteLine(”|{0,7}|{1,7}|”, x, Math.Sin(x));
}
…
```

В приведенном примере цикл `for` используется в классической традиционной форме – со всеми стремя секциями заголовка.

Приведем несколько вариаций использования цикла `for` для реализации табуляции функции:

– в секции инициализации нет объявления переменных цикла

```
double x;
for (x = xmin; x < xmax; x +=dx) // табуляция функции
{
  Console.WriteLine(”|{0,7}|{1,7}|”, x, Math.Sin(x));
}
```

– отсутствует секция инициализации

```
double x = xmin;
for (; x < xmax; x +=dx) // табуляция функции
{
  Console.WriteLine(”|{0,7}|{1,7}|”, x, Math.Sin(x));
}
```

– отсутствуют секции инициализации и изменения

```
double x = xmin;
for (; x < xmax;) // табуляция функции
{
  Console.WriteLine(”|{0,7}|{1,7}|”, x, Math.Sin(x));
  x += dx;
}
```

– тело цикла пусто, табуляция выполняется в секции изменения

```
for (double x = xmin; x < xmax; x +=dx, Console.WriteLine(
”|{0,7}|{1,7}|”, x, Math.Sin(x)))
{
}
```

Приведенные примеры решают задачу одинаково. Вариаций, приводящих к искомому результату, может быть довольно много.

Также следует отметить, что

```
for(;;)
{
}
```

является синтаксически корректной конструкцией.

Цикл foreach

Цикл применяется для перебора всех элементов из составного (агрегативного) объекта.

Синтаксис оператора:

```
foreach (<тип> <имя> in <агрегат>)
{
    <тело цикла>
}
```

Агрегат – это перебираемый составной объект, в заголовок цикла передается его имя. Тип – тип элементов агрегата. Имя – локальная переменная – параметр цикла, соответствующая по типу элементам агрегата (то есть тип элементов агрегата должен быть неявно приводим к типу параметра). На каждой итерации один из элементов агрегата копируется в локальную переменную. В теле цикла производится обработка переменной цикла. Таким образом, сам агрегат может быть использован только для чтения. Цикл гарантирует, что элементы агрегата будут перебраны полностью, порядок обхода в общем случае не определен. К недостаткам этого цикла можно отнести невозможность обратиться к соседним по отношению к обрабатываемому элементам. Цикл используется в тех случаях, когда не требуется изменение самого агрегативного объекта. Часто цикл foreach используется при работе с массивами, о которых речь пойдет в более поздних темах.

Операторы передачи управления (операторы перехода)

Применяются для принудительного изменения порядка выполнения последовательной программы.

Оператор безусловного перехода goto

Оператор имеет три формы:

```
goto <метка>;
goto case константа;
goto default;
```

Первая форма передает управление оператору, помеченному меткой:

```
<метка>: <оператор>;
```

Внутри блока метка должна быть уникальной. Такая форма оператора `goto` может передать управление только оператору внутри текущего блока. Передать управление другому блоку нельзя. Использование оператора `goto` нежелательно, поскольку нарушает принципы структурного программирования. Однако в некоторых случаях он может быть оправдан и даже полезен.

Две оставшихся формы оператора `goto` используются в переключателе `switch`. Одна – для передачи управления соответствующей ветви `case`, другая – ветви `default`.

Оператор выхода из блока break

Оператор используется для принудительного прекращения выполнения текущего блока (тела цикла, переключателя) и передачи управления оператору, следующему непосредственно за прерванным.

Оператор перехода к следующей операции continue

Оператор используется только в циклических конструкциях для принудительного прекращения выполнения текущей итерации цикла и перехода к следующей итерации.

Оператор возврата из метода return

Оператор прекращает выполнение метода и возвращает управление в точку вызова прерванного метода. Оператор имеет две формы:

```
return;
return <значение>;
```

Первая форма используется для выхода из тех методов, которые не имеют возвращаемого значения – методов с возвращаемым значением `void`.

Вторая форма используется для выхода из метода и возвращения значения, являющегося результатом выполнения метода. Следует отметить, что значение, возвращаемое с помощью оператора `return`, должно соответствовать типу, заявленному в заголовке метода.

Тема 3. Объектно-ориентированное программирование.

Объектно-ориентированного программирование – объединение в единую сущность данных и методов их обработки.

Достоинства ООП:

- разделение функциональности, разделение кода, следовательно, упрощение разработки;
- расширяемость программируемых решений;
- легкость модернизации, сохранение совместимости.

Недостатки ООП:

- избыточность кода;
- сложность проектирования;
- неэффективность исполнения;
- неэффективность распределения памяти.

Понятие объекта и класса.

Написание программы на языке C# ведется с точки зрения описания классов. На основе описания классов создаются сущности – объекты (экземпляры класса). Объекты обладают следующими свойствами:

- состояние объекта – его свойства, параметры;
- поведение объекта – возможные действия с объектом;
- уникальность объекта – его отличие от других объектов того же класса.

Все объекты, являющиеся экземплярами одного класса, ведут себя одинаково и имеют одинаковый набор свойств. Каждый объект хранит информацию о своем состоянии. Со временем состояние объекта может измениться, но только в результате его поведения. Состояние объекта является, как правило, уникальным.

Итак, класс – это описание множества объектов, объект – это один экземпляр класса. Чем меньше подробностей имеется в описании класса, тем большее количество объектов может подходить под это описание.

Основные принципы ООП.

Инкапсуляция – сокрытие внутренней реализации класса и отделение его внешнего представления от внутренней реализации.

Наследование – отношение между классами, при котором один класс использует структуру и поведение другого класса (одиночное наследование) или других классов (множественное наследование).

Полиморфизм – положение теории типов, согласно которому имена могут обозначать объекты разных, но имеющих общего предка, классов.

Состав класса.

Свойства объекта описываются *полями*, поведение – *методами*. В классе также могут быть описаны *события* и *вложенные классы*.

Синтаксис описания класса:

```
<атрибуты><спецификаторы> class <имя класса>:<предки>
{
    <тело класса>;
}
```

Атрибуты задают дополнительную информацию о классе и будут рассматриваться по мере необходимости.

Спецификаторы делятся на две категории – спецификаторы и спецификаторы доступа. Спецификаторы доступа указывают доступность класса из других классов программы. Спецификаторы задают для класса некоторые свойства.

Спецификаторы доступа:

`public` – публичный класс с неограниченным доступом;

`protected` – доступ только из самого класса и из производных классов (потомков);

`internal` – доступ только в пределах текущей сборки;

`protected internal` – доступ из производных классов и из текущей сборки;

`private` – доступ только в пределах текущего класса.

Спецификаторы:

`new` – новый вложенный класс, переопределяющий класс, унаследованный от предка;

`static` – статический класс (для вложенных классов);

`abstract` – абстрактный класс, предназначен для наследования;

`sealed` – класс, от которого запрещено наследовать.

При описании класса могут быть указаны несколько спецификаторов – простых и доступа.

Имя класса составляется по нотации Паскаля.

Атрибуты, спецификаторы и предков указывать не обязательно (для класса будут установлены значения по умолчанию), тело класса может быть пустым.

Объекты системных классов создаются неявным образом при первом обращении к ним, объект класса, описываемого программистом, создается явным образом с помощью операции `new` – операции выделения динамической памяти для хранения объекта. Результатом выполнения операции является ссылка на созданный объект:

```
MyClass c = new MyClass();
```

Операции присваивания и сравнения для объектов выполняются по правилам ссылочных переменных. При сравнении объектов сравниваются ссылки на них. Ссылки равны, если они указывают на один и тот же объект. При присваивании копируется ссылка на объект, а не сам объект:

```
MyClass c = new MyClass();
MyClass d = new MyClass(); // c ≠ d
MyClass e = c; // c = e
```

В языке C# в состав класса могут входить следующие элементы:

- поля – состояние объекта;
- константы – поля с неизменяемым значением, как правило, связаны с классом вообще, а не с отдельными объектами;
- методы – описание поведения класса;
- конструкторы – инициализаторы объектов класса;
- свойства – специальные методы доступа к полям класса;
- индексаторы – специальные методы индексированного доступа к полям класса;
- операции – действия над объектами класса в символах операций;
- деструктор – описание действий, которые необходимо выполнить перед удалением объекта;
- события – уведомления, генерируемые объектами класса, применяются в событийном программировании;
- вложенные классы – классы, описанные внутри класса.

Поля.

Поле описывает один из элементов состояния объекта. В совокупности поля представляют собой полный набор характеристик экземпляров класса и предназначены для хранения значений этих характеристик. Описание поля очень похоже на описание переменной. Синтаксис описания поля;

```
<атрибуты> <спецификаторы> <тип> <имя> = <начальное значение>;
```

Атрибуты будут рассмотрены позднее, спецификаторы – один из спецификаторов доступа и спецификаторов, описывающих дополнительные свойства поля. Спецификаторы доступа уже были перечислены, перечислим теперь спецификаторы для полей:

`new` – новое поле, скрывающее поле, унаследованное от предка;

`static` – статическое поле, является характеристикой класса, а не объекта;

`readonly` – ссылочное поле только для чтения (ссылочная константа);

`volatile` – поле, предназначенное для совместного использования несколькими потоками инструкций.

Тип поля – это тип данных, которому принадлежит значение, хранимое полем. Имя поля составляется, как правило, по нотации Camel. Начальное значение – это то значение, которое будет принимать поле в момент создания любого экземпляра класса. Задавать начальное значение

при описании поля не обязательно. Это лишь означает, что инициализирующее значение будет задано позже специальным образом, для каждого экземпляра класса уникальное.

Если для поля не указать спецификатор доступа, то по умолчанию такое поле считается приватным, то есть ему добавляется спецификатор `private`. Поля необходимо делать приватными, за исключением особых случаев, поскольку поля относятся ко внутренней реализации класса, которую в соответствии с принципом инкапсуляции нужно скрывать от других элементов программы.

Доступ к полю осуществляется с помощью операции разыменования «.» : `<имя объекта>.<имя поля>`. Если поле приватно, то такое обращение невозможно. Как реализуется доступ к приватным полям, мы рассмотрим при описании методов.

Константы.

Константы – это поля с неизменяемым значением. Их описание отличается от обычных полей лишь наличием ключевого слова `const`.

```
<атрибуты><спецификаторы>const<тип><имя> = <начальное значение>;
```

Однако назначение констант сильно отличается от назначения обычных полей. Поскольку значение константы изменить нельзя, то для нее обязательно задается начальное значение при описании. Кроме этого, константа существует в единственном экземпляре на все объекты класса, поэтому она должна обладать доступностью шире, чем private, и обращение к ней проводится по имени класса, а не по имени объекта:

`<имя класса>.<имя константы>.`

Говорят, что константы используются в контексте класса, а не в контексте объекта.

Аналогичны константам так называемые статические (`static`) поля. Они существуют в единственном экземпляре на класс, используются в контексте класса, вызываются по имени класса, а не объекта, должны быть сразу проинициализированы, однако их значения можно изменять.

Пример:

```
class MyClass
{
   public const int a = 1;//публичная константа
   public double b = 0.5; //публичное поле (не рекомендуется!)
   public static string s = ”Мама мыла раму”; //статическое поле
   float x = 1.2; //непубличное (private) поле
}
class Program
{
   public static void Main()
   {
      MyClass c = new MyClass(); //создание объекта
      double x = c.b + MyClass.a; /*обращение к публичному полю и
      публичной константе*/
```

```
      Console.WriteLine(MyClass.s);  /*обращение к статическому
полю*/
      Console.WriteLine(c.x); //НЕДОПУСТИМО!
   }
}
```

Методы.

Методы описывают поведение экземпляров класса, то есть те действия, которые может совершать объект или действия, совершаемые над объектом. Существуют методы, описываемый поведение класса, а не экземпляра. Метод немного похож на привычную для структурного программирования подпрограмму, только неразрывно связан с тем классом, в котором описан.

Синтаксис метода:

```
<атрибуты><спецификаторы><тип><имя метода> (<список параметров>
{
      <тело метода>;
}
```

Первая строка является заголовком метода. Рассмотрим его подробнее.

Атрибуты метода будут рассмотрены по мере необходимости. Спецификаторы аналогичны спецификаторам класса и поля, то есть представляют собой комбинацию спецификатора доступа и спецификатора, наделяющего метод особыми характеристиками. Спецификаторы доступа уже были рассмотрены. Спецификаторы метода:

`new` – новый метод, переопределяющий метод, унаследованный от предка;

`static` – статический метод, выполняется в контексте класса, а не объекта;

`virtual` – виртуальный метод, возможно переопределение в потомках;

`abstract` – абстрактный метод без реализации, предназначен для задания реализации в потомках;

`override` – метод, переопределяющий виртуальный или абстрактный метод в потомке;

`sealed` – метод, переопределение которого в потомках запрещено.

Чаще всего метод имеет спецификатор доступа, шире, чем `private`, поскольку метод относится к внешнему представлению и должен быть доступен извне.

Тип – это тип результата, который возвращается из метода. Для возвращения результата используется оператор `return` с параметром (тип параметр должен соответствовать типу, заявленному в заголовке метода). Если метод не возвращает значения, то в качестве типа указывается ключевое слово `void`. Тогда для выхода из метода используется простая форма оператора `return`. Если же оператор `return` является последним оператором в теле метода, то записывать его не обязательно.

Имя метода должно отражать суть действий, выполняемых методом, и составляется по нотации Паскаля.

Список параметров – набор параметров метода с указанием их типов. Параметр метода является локальной для метода переменной и действует во всем теле метода наравне с локальными переменными, объявленными в самом теле метода. Параметры в метод всегда передаются по значению.

Тело метода – блок операторов, выполняющих действия над объектами.

Имя метода и список параметров представляют *сигнатуру* метода. В классе не должно быть методов с одинаковыми сигнатурами. Но в классе может быть группа методов с одинаковыми именами. Разницу в сигнатуры при этом вносит список параметров. Такой метод называется перегруженным. Компилятор выбирает для исполнения ту версию метода, которая подходит под передаваемые значения параметров. При этом стоит учитывать возможность неявных преобразований:

```
public int Method1 (int x)
{
     …
}
public int Method1 (double x)
{
     …
}
```

Формально методы имеют разные сигнатуры – их параметры различаются по типу. Однако следующий вызов:

```
<…>.Method1(10);
```

приведет к ошибке компиляции, поскольку переданное значение подходит и под первую, и под вторую сигнатуры.

Существует несколько соглашений по включению в состав класса методов с традиционными именами и традиционным назначением. Например, методы, предоставляющие доступ к непубличным полям – те методы, которые реализуют принцип инкапсуляции. То есть значение поля на чтение и на запись предоставляется опосредованно, через методы доступа. Метод, предназначенный для считывания значения поля, начинается с префикса `Get`. Такие методы называют геттерами. Метод для установки значения поля начинается с префикса `Set` – методы сеттеры: После префикса в имени метода следует название поля, доступ к которому обеспечивает метод. Приведем пример классического геттера и классического сеттера:

```
class MyClass
{
   int x = 5; //непубличное поле
   public int GetX() //метод-геттер
   {
       return x; //возврат значения непубличного поля
   }
```

```
   public void SetX(int val) //метод-сеттер
   {
       x = val; //задание значения непубличного поля
   }
}
```

Следует также отметить, что в теле методов могут производиться любые действия, но, как правило, эти действия связаны с проверками значений параметров на валидность или прав доступа.

Еще одним устоявшимся именем называются методы преобразования. Они начинаются с префикса `To`, далее следует название того, к чему метод преобразует. Например, метод `ToString()`, уже упоминавшийся в предыдущей теме, предназначен для преобразования содержимого переменной к строке.

Ключевое слово this.

Каждый объект содержит ссылку на самого себя. Это неявная переменна внутри объекта носит имя `this`. Эта ссылка явным образом используется в нескольких случаях. Например, когда параметр метода совпадает по имени и типу с полем объекта, логично предположить, что возникнет конфликт имен, или локальный параметр закроет собой поле. Однако поле – это важная часть объекта, она всегда должна быть доступна для обращения к ней. Эта доступность достигается с помощью ссылки `this`.

```
class MyClass
{
   int a;//поле
   public void SetA(int a)//параметр совпадает по имени с полем
   {
      this.a = a; //поле осталось доступно по ссылке this
   }
}
```

Также ссылка `this` используется для идентификации объекта, который вызывает на исполнение метод. Связано это с тем, что объекты в памяти хранят только содержимое полей, поскольку они описывают уникальность объекта. Методы, поскольку они общи для всех объектов класса, хранятся не в каждом объекте, а в единственном экземпляре в памяти. Все объекты используют метод совместно. Для того, чтобы метод работал именно с полями вызвавшего его объекта, из этого объекта в метод и передается ссылка `this`.

Ключевое слово this используется еще в нескольких случаях, о которых речь пойдет чуть позже. Однако оно всегда означает ссылку объекта самого на себя.

Конструкторы.

Конструктор предназначен для инициализации экземпляра класса. Это «создатель» объекта. Он должен разместить объект в динамической памяти, а

также правильно и корректно задать начальные значения полей объекта. Конструктор вызывается тогда, когда экземпляр класса создается с помощью операции new. Конструктор может задавать разные значения полей для разных экземпляров. Особенности конструктора:

- имя конструктора совпадает с именем класса;
- у конструктора нет возвращаемого значения (даже `void`);
- в классе может быть несколько конструкторов с разными параметрами для разных инициализаций (перегруженный конструктор), все конструкторы должны иметь разные сигнатуры;
- если в конструкторе каким-либо полям не задаются значения, то они инициализируются значениями по умолчанию – `false` для булевских полей, 0 – для числовых полей, `null` – для ссылочных полей;
- если в классе не описан ни один конструктор, то автоматически в состав класса добавляется конструктор по умолчанию, который задает полям значения по умолчанию;
- пользователь может описать конструктор, также называемый конструктором по умолчанию – обычный конструктор без параметров – и задать в нем собственную инициализацию полей;
- если в классе нет ни конструктора по умолчанию, ни конструктора без параметров, значит в классе пользователем описан как минимум один конструктор с параметрами;
- конструктор должен иметь спецификатор доступа шире, чем `private`.

Конструктор может вызвать на исполнение другой конструктор текущего класса. Такой вызов удобен, если один конструктор инициализирует те поля, которые еще не были заданы в другом.

Рассмотрим пример:

```
class MyClass
{
   int a, b; // поля объекта
   double c; //
   public MyClass() //конструктор без параметров –
   {                //конструктор по умолчанию
      a = 1;
      b = 1;
      c = 1.1;
   }
   public MyClass(int a) //конструктор с целым параметром
   {
      this.a = a;
   }
   public MyClass(int b) //конструктор с целым параметром
   {                     //недопустимо!!! сигнатура совпадает
      this.b = b;         //с предыдущим конструктором!
   }
   public MyClass(double a) //конструктор с вещественным
```

```
   {                            //параметром
      c = a;  // обращение к полю возможно и без слова this
   }
   public MyClass(int a, double b) //конструктор с двумя
   {                               //параметрами
      this.a = a;//или можно задать значение поля b: this.b = a;
      c = b;
   }
   public MyClass(int a, int b)//конструктор с двумя целыми
   {                           // параметрами
      this.a = a;
      this.b = b;
   }
   public MyClass(int a, int b, double c) //конструктор с тремя
   {                                      //параметрами
      this.a = a;
      this.b = b;
      this.c = c;
   }
}
```

В данном примере перечислены не все возможные конструкторы, тело конструктора тоже может быть задано иначе. Главное – все приведенные конструкторы имеют разные сигнатуры. Для сигнатуры важны не имена параметров, а их тип и порядок следования. Проиллюстрируем возможность вызова конструктора из другого конструктора, перепишем предыдущий пример:

```
class MyClass
{
   int a, b; // поля объекта
   double c; //
   public MyClass(int a) //конструктор с целым параметром
   {
      this.a = a;
   }
   public MyClass(int a, int b): this(a) //конструктор с двумя
   {                                     //параметрами
      this.b = b;
   }
   public MyClass(int a, int b, double c): this(a, b)
   {                          //конструктор с тремя параметрами
      this.c = c;
   }
}
```

Второй конструктор вызывает первый конструктор явным образом с помощью ключевого слова `this` и передает ему в качестве параметра значение своего параметра `a`. Третий конструктор явно вызывает второй конструктор и передает ему два параметра. Конструкция после «:» вызывается на исполнение до начала исполнения тела конструктора и называется *инициализатором*. Таким образом, поля, значения которых

задаются в инициализаторе, нет необходимости инициализировать еще раз в самом конструкторе. Циклические взаимные вызовы конструкторов отслеживаются на этапе компиляции.

Количество конструкторов в классе в общем случае не ограничивается, однако все же стоит руководствоваться здравым смыслом и необходимостью описания того или иного конструктора.

Тема 4. Состав класса в языке C#.

Базовая структура класса была рассмотрена в теме 3. Но кроме перечисленных элементов (полей, методов и конструкторов) в классе могут быть описаны дополнительные элементы, упрощающие работу с объектами класса и предоставляющие широкий спектр возможностей.

Свойства.

Свойства – это специальные методы доступа к полям объекта (методы-аксессоры).

```
<атрибуты> <спецификаторы> <тип> <Имя свойства>
{
   <атрибуты> <спецификаторы> get
   {
      <код доступа для чтения поля>;
   }
   <атрибуты> <спецификаторы> set
   {
      <код доступа для записи поля>;
   }
}
```

Атрибуты и спецификаторы соответствуют атрибутам и спецификаторам методов. Чаще всего используется спецификатор `public`, поскольку свойства открывают доступ к закрытым полям и представляют собой внешнее представление объекта. Блок `get` или блок `set` могут отсутствовать, но не оба сразу. Если блок `get` отсутствует, то свойство является свойством только для записи, если блок `set` отсутствует, то свойство является свойством только для чтения. Когда присутствуют оба блока, то свойство предоставляет доступ и на чтение, и на запись.

Блоки `get` и `set` могут иметь собственные атрибуты и спецификаторы доступа, отличные от спецификатора доступа всего свойства, но не шире его. То есть свойство предоставляет возможность гибко управлять видимостью чтения и записи. Если спецификаторы и атрибуты у блоков отсутствуют, то на них распространяется спецификатор свойства.

Тип – это тот тип данных, к которому принадлежит то поле, к которому открывает доступ свойство.

Имя свойства, как правило, совпадает с именем поля, но составляется по нотации Паскаля (поле именуется в соответствии с нотацией Camel).

В блоке set свойства используется системная переменная value, тип которой совпадает с типом, заявленным в заголовке свойства. Она используется для задания значения контролируемого свойства.

Пример.

```
class Rectangle
{
   double height;//приватное поле – высота прямоугольника
```

```
   double width://приватное поле – ширина прямоугольника
   double square;//приватное поле – площадь прямоугольника
   public Rectangle() /*конструктор по умолчанию, задающий
   единичные значения ширине и высоте*/
   {
      height = 1;
      width = 1;
      square = 1;
   }
   public double Height//свойство – аксессор к полю height
   {
      get
      {
         return height;//чтение поля height
      }
      set
      {
         if (value > 0)
         {
            height = value;//задание значения поля height
            square = height * width;//пересчет площади
         }
      }
   }
   public double Width//свойство – аксессор к полю width
   {
      get
      {
         return width;//чтение поля width
      }
      set
      {
         if (value > 0)
         {
            width = value;//задание значения поля width
            square = height * width; //пересчет площади
         }
      }
   }
   public double Square//свойство для чтения поля square
   {
      get
      {
         return square;
      }
   }
}
…
public static void Main()
{
   Rectangle r1=new Rectangle();//создание единичного прямоугольника
   r1.Height=2;//установка значения высоты (блок set свойства Height)
```

```
   Console.WriteLine(”Height = {0}, width = {1}, square = {2}”,
r1.Height, r1.Width, r1.Square);/*вызовы блоков get свойств Height,
Width, Square*/
   r1.Width = 3;//установка значения ширины (блок set свойства Width)
   Console.WriteLine(”Height = {0}, width = {1}, square = {2}”,
r1.Height, r1.Width, r1.Square);/*вызовы блоков get свойств Height,
Width, Square*/
}
…
```

В данном примере свойство `Square` предназначено только для чтения поля square, поскольку значение площади не может устанавливаться независимо от значений высоты и ширины прямоугольника, то есть поле является зависимым, его необходимо пересчитывать при каждом изменении ширины или высоты, и, следовательно, явное хранение его в объекте нецелесообразно.

Изменим предыдущий пример:

```
class Rectangle
{
   double height;//приватное поле – высота прямоугольника
   double width://приватное поле – ширина прямоугольника
   public Rectangle() /*конструктор по умолчанию, задающий
   единичные значения ширине и высоте*/
   {
      height = 1;
      width = 1;
   }
   public Rectangle(double h,double w)//конструктор с параметрами
   {
      height = h;
      width = w;
   }
   public double Height//свойство – аксессор к полю height
   {
      get
      {
         return height;//чтение поля height
      }
      set
      {
         if (value > 0) height=value;//задание значения поля height
      }
   }
   public double Width//свойство – аксессор к полю width
   {
      get
      {
         return width;//чтение поля width
      }
      set
      {
         if (value > 0) width = value;//задание значения поля width
```

```
      }
   }
   public double Square //свойство, имитирующее наличие поля square
   {
      get
      {
         return height * width; //возвращается вычисляемое значение
      }
   }
}
…
public static void Main()
{
   Rectangle r1=new Rectangle(5, 6);//создание прямоугольника
   Console.WriteLine(”Height = {0}, width = {1}, square = {2}”,
r1.Height, r1.Width, r1.Square);/*вызовы блоков get свойств Height,
Width, Square*/
}
…
```

В приведенном примере поля `square` в классе не существует, но внешнее представление объекта включает в себя свойство только для чтения `Square`, имитируя наличие поля.

Блоки свойства предназначены для различных проверок присваиваемых в поле значений, а также для проверок возможности чтения или изменения поля каким-либо внешним объектом. В общем случае, в свойстве могут быть описаны любые операторы. По своему назначению свойство является методом доступа для поля (внешним представлением внутренней реализации), но может и не соответствовать полю, а имитировать его наличие в объекте (внешнее представлении открыто, а внутренняя реализация скрыта).

Индексаторы.

Индексатор – это разновидность свойства. Применяется, как правило, в том случае, если в классе описано скрытое поле – массив. Индексатор позволяет обращаться к элементу массива по индексу, записываемому у самого объекта, без явного обращения к массиву внутри объекта. То есть индексатор позволяет интерпретировать объект, содержащий массив, как просто массив.

```
<атрибуты> <спецификаторы> <тип> this[<список параметров>]
{
   <атрибуты> <спецификаторы> get
   {
      <код доступа для чтения поля>;
   }
   <атрибуты> <спецификаторы> set
   {
      <код доступа для записи поля>;
   }
}
```

Атрибуты и спецификаторы аналогичны атрибутам и спецификаторам свойства, блоки get и set могут иметь разные спецификаторы доступа, не расширяющие область видимости всего индексатора. Если для блоков спецификаторы не указываются, то на блоки распространяется спецификатор доступа индексатора. Также, как и в свойстве, в индексаторе может отсутствовать либо блок set (индексатор только для чтения), либо блок get (индексатор только для записи), но не оба сразу.

Тип – это тип элементов массива, список параметров – индексы элементов в массиве и(или) необходимые дополнительные параметры. Индексатор может быть перегруженным, то есть, если в классе описывается несколько индексаторов, то все они должны иметь разные списки параметров.

В индексаторах, как и в свойствах, содержатся проверки допустимости переданных значений индексов, дополнительная валидация данных, проверки прав доступа. В общем случае, в индексаторе могут содержаться любые операторы.

Пример.

```
class MyArray
{
   int[] ar; //приватное поле – массив
   public MyArray(int len)
   {
      ar = new int[len]; //создание массива
   }
   public int this[int i]
   {
      get
      {
         if (i >= 0 && i < ar.Length) return ar[i]; /*получение
элемента массива по индексу*/
      }
      set
      {
         if (i >= 0 && i < ar.Length) ar[i] = value; /*запись
элемента массива по индексу*/
      }
   }
   public int Length//свойство для чтения, возвращающее длину массива
   {
      get
      {
         return ar.Length
      }
   }
}
…
public static void Main()
{
   MyArray my = new MyArray(10);
```

```
   for (int i = 0; i < my.Length; i++)/*чтение длины массива через
свойство объекта*/
   {
      my[i] = i * 5;/*обращение к объекту как к массиву (блок set
индексатора)*/
   }
   for (int i = 0; i < my.Length; i++)
   {
      Console.Write(my[i]+" ");/*обращение к объекту как к массиву
(блок get индексатора)*/
   }
}
…
```

Как и свойство, индексатор может имитировать наличие приватного массива в описании класса.

```
…
class MyArray
{
…
   public int this[int i]
   {
      get
      {
         return i;/*индексатор возвращает в качестве элемента массива
переданный параметр*/
      }
   }
…
}
…
```

Параметры методов.

В методы параметры могут передаваться двумя способами – по значению (для значащих типов данных – передается значение параметра) и по ссылке (для ссылочных типов данных – передается ссылка на объект).

Однако в языке существует еще несколько способов передачи параметров в метод.

1. Передача значащего параметра по ссылке (параметр-ссылка). Такая передача производится с помощью ключевого слова `ref` (сокращение от reference - ссылка). Такая передача используется в тех случаях, когда нужно изменить значащий параметр в теле метода так, чтобы все изменения влияли на исходное значение параметра. Метод, получив ссылку на параметр, работает непосредственно с ним, а не с его локальной значащей копией. Такой параметр-ссылку можно считать еще одним результатом выполнения метода, в дополнение к тому, который заявлен в заголовке метода и возвращается из тела метода с помощью оператора `return`. Такой подход применяется при передаче значащих параметров. В передаче ссылочных параметров с помощью ключевого слова `ref` нет особого смысла, поскольку

ссылочные параметры и так передаются по ссылке. Однако передача ссылочного параметра как параметра-ссылки (через `ref`) позволит изменить в теле метода не только содержимое объекта, но и саму ссылку на него, то есть после выполнения метода параметр может ссылаться на другой объект, а не на исходный.

Следует особо отметить, что передача параметра в метод с помощью слова `ref` требует ОБЯЗАТЕЛЬНОЙ предварительной инициализации параметра.

Пример. Метод принимает 2 целых числа и максимально возможную верхнюю границу, возвращает сумму переданных чисел, и меняет значение верхней границы на максимальное из двух переданных значений, если значения равны, то максимум не меняется.

```
…
public static int Sum(int x, int y, ref int max)
{
   if (x > y) max = x;
   else if (y > x) max = y;
   return x + y;
}
…
int x =5, y = 10, max = 8;
int sum = Sum(x, y, ref max); //sum = 15, max = 10
…
```

Как и в описании метода, так и при вызове метода параметр-ссылка передается по ключевому слову `ref`.

2. Выходной параметр. Передается с помощью ключевого слова `out`. Используется тогда, когда в теле метода формируется новое значение параметра, или создается новый объект, который нужно вернуть из метода в дополнение к основному возвращаемому результату. Параметру, переданному по `out`, в теле метода ОБЯЗАТЕЛЬНО должно быть присвоено значение, а вот предварительная его инициализация перед передачей в метод (как в случае параметра-ссылки) не нужна. В остальном работа с параметром, переданным по `out` аналогична работе с параметром, переданным по `ref`.

Пример. Метод деления двух целых чисел нацело с остатком.

```
…
public static int Divide(int x, int y, out int rem)
{
   rem = x % y;
   return x / y;
}
…
int x =12, y = 5, rem;
int z = Divide(x, y, out rem); //z = 2, rem = 2
…
```

3. Методы с переменным числом параметров. Иногда заранее неизвестно, какое количество параметров будет передано в метод, известен

только их тип. Тогда для передачи параметров в метод используется ключевое слово `params`. Параметр, передаваемый с помощью ключевого слова `params` должен быть самым последним в списке параметров метода, и обозначает массив неопределенной длины элементов заданного типа. Массивы подробно будут рассмотрены в теме 5.

Пример. Метод выводит на экран переданные целые значения, количество выводимых значений заранее неизвестно.

```
…
public static void Print(string s, params int[] a)
{
   Console.WriteLine(s);
   if (a != null && a.Length !=0)
       foreach (int element in a)
       Console.WriteLine(”Элемент = ” + element);
}
…
int[] ar = new int{1, 2, 3, 4, 5};
Print(”Массив целых чисел”, ar);
Print(”Переменное число параметров”, 2, 7, 3, 9, 12, 1);
…
```

Внутри метода с параметром, переданным с помощью ключевого слова params работа ведется, как с обычным массивом. Поэтому первый вызов метода Print, в который явным образом передается массив, тоже будет корректным. Во втором вызове массив явным образом не передается, но считываемые значения неявно копируются во внутренний массив, поэтому передаваемые значения должны допускать неявное преобразование к типу элементов массива.

Операции класса.

В языке существует возможность переопределить символ стандартной операции таким образом, чтобы для объектов конкретного класса операция выполнялась по-своему, например, для чисел и для строк операция сложения выполняется по-разному. Такое переопределение позволяет использовать объекты пользовательских типов в выражениях наравне со стандартными типами. Такой прием называется *перегрузкой* операции. Перегрузка операции имеет смысл только тогда, когда очевидно, что означает операция для объектов класса, когда операция делает текст программы более понятным и читабельным. Если неясно, что делает объект, являясь операндом операции, то в перегрузке операции нет смысла.

Общие свойства переопределяемой операции:

1. Операция объявляется как `public static`.
2. Параметры операции передаются по значению (без `ref` и `out`).
3. Операции не должны совпадать по сигнатуре.
4. Спецификатор доступа класса, для которого переопределяется операция, не должен сужать спецификатор доступа операции.
5. Нельзя вводить собственные обозначения операций.

6. Приоритеты операций при перегрузке сохраняются.

7. Операция не должна изменять значение передаваемого параметра, а должна создавать и новый объект, возвращаемый в качестве результата.

Для перегрузки операции используется ключевое слово `operator`, тело операции представляет собой обычное тело метода.

Унарные операции

К унарным операциям относят «+», «-», «~», «!», «++», «--».

Операции «+», «-», «~», «!» должны в качестве параметра принимать объект текущего класса, могут возвращать величину любого типа. Операции «++» и «--» должны и принимать и возвращать тот тип, для которого переопределяются. Префиксный и постфиксный инкремент и декремент при переопределении не различаются, выполняются стандартным образом.

Сигнатура унарной операции:

```
public static <тип> operator <символ операции> (<тип параметра>
<имя параметра>)
{
<тело операции>
}
```

Пример. Описать класс рациональной дроби и перегрузить в нем унарные операции.

```
class Drob
{
   int ch,zn; //числитель и знаменатель
   public Drob()
   {
      ch = 1;
      zn = 1;
   }
   public int Chislitel // доступ к числителю
   {
      get
      {
         return ch;
      }
      set
      {
         ch = value;
      }
   public int Znamenatel //доступ к знаменателю
   {
      get
      {
         return zn;
      }
      set
      {
         if (value != 0)zn = value;
         else zn = 1; //знаменатель не может быть равен 0
      }
```

```
   }
   public static Drob operator – (Drob d)/*перегрузка операции
арифметического отрицания, возвращается новая дробь*/
   {
      Drob newd = new Drob();//создание нового объекта
      newd.Chislitel = -d.Chislitel;
      newd.Znamenatel = d.Znamenatel;
      return newd;/*возвращение созданного объекта в качестве
результата*/
   }
   //операция «+» перегружается аналогично
   //операции «!», «~» для дробей не имеют смысла
   public static Drob operator ++ (Drob d)/*перегрузка операции
инкремента*/
   {
      Drob newd = new Drob();
      newd.Chislitel = d.Chislitel+d.Znamenatel;//увеличение на 1
      newd.Znamenatel = d.Znamenatel;
      return newd
   }
//операция «--» перегружается аналогичным образом
}
…
public static void Main()
{
   Drob d = new Drob();
   d.Chislitel = 2;
   d.Znamenatel = 3;
   d++; //инкремент дроби
   Console.WriteLine(”Дробь={0}/{1}”, d.Chislitel, d.Znamenatel);
   Drob d2 = -d;
   Console.WriteLine(”Дробь={0}/{1}”,d2.Chislitel,d2.Znamenatel);
}
```

Бинарные операции.

К бинарным операциям относят «+», «-», «*», «/», «%», «&», «|», «^», «>>», «>>>» ,«<<» и т.д.. а также операции отношения и сравнения «==», «!=», «>=», «<=», «<», «>». Операции отношения перегружаются парами, возвращают результат логического типа, подробнее перегрузку операций отношения мы рассмотрим в теме 7.

Бинарные операции принимают два параметра, хотя бы один из них должен быть того типа, для которого перегружается операция, могут возвращать величину любого типа.

Добавим в описанный ранее класс Drob несколько бинарных операций:

```
public static Drob operator + (Drob d1, Drob d2)/*перегрузка
операции сложения двух дробей, при этом конфликта с операцией
арифметического плюса конфликтов не возникает, они отличаются
сигнатурами*/
{
   Drob newd = new Drob();//создание нового объекта
```

```
   newd.Chislitel = d1.Chislitel * d2.Znamenatel + d1.Znamenatel
* d2.Chislitel; //числитель суммы двух дробей
   newd.Znamenatel=d1.Znamenatel*d2.Znamenatel;//знаменатель суммы
   return newd
}
```

Добавим еще несколько операций сложения с различными сигнатурами:

```
public static Drob operator + (int d1, Drob d2)/*перегрузка
операции сложения целого числа и дроби, при этом конфликта с
операцией сложения двух дробей не возникает, сигнатуры
различны*/
{
   Drob newd = new Drob();//создание нового объекта
   newd.Chislitel = d1*d2.Znamenatel+d2.Chislitel;//числитель суммы
   newd.Znamenatel=d2.Znamenatel;//знаменатель суммы
   return newd
}
public static Drob operator + (Drob d1, int d2)/*перегрузка
операции сложения дроби и целого числа, конфликта с ранее
описанными операциями сложения нет, сигнатуры различны*/
{
   Drob newd = new Drob();//создание нового объекта
   newd.Chislitel = d1.Chislitel+d1.Znamenatel*d2;//числитель суммы
   newd.Znamenatel=d1.Znamenatel;//знаменатель суммы
   return newd
}
```

Остальные бинарные операции перегружаются аналогичным образом, если в них есть необходимость.

Операции преобразования типа.

Описывают явное и неявное преобразование объекта текущего пользовательского типа к другим типам данных. По обыкновению, различают операции неявного и явного преобразования.

Операция неявного преобразования:

```
implicit operator <тип> (<параметр>) {}
```

Операция явного преобразования:

```
explicit operator <тип> (<параметр>) {}
```

Параметр обеих операций преобразуется к указанному в заголовке операции типу. Тип параметра и тип результата не должны быть связаны наследованием.

Пример. Операции преобразования для класса рациональных дробей.

```
public static implicit operator double (Drob d) /*перегрузка
операции неявного преобразования дроби к вещественному числу*/
{
   return (double)d.Chislitel / d.Znamenatel;
}
```

```
public static explicit operator Drob (int i) /*перегрузка операции явного преобразования целого числа к дроби*/
{
   Drob d = new Drob();
   d.Chislitel = i;
   d.Znamenatel = 1;
   return d;
}
…
Drob a = new Drob();
a.Chislitel = 2;
a.Znamenatel = 3;
double b = a; /*вызывается описанная операция неявного преобразования дроби к вещественному числу*/
int c = 12;
Drib d = (Drob)c; /*вызывается описанная операция явного преобразования целого числа к дроби*/
…
```

Неявное преобразование выполняется автоматически при присваивании объекта переменной, при передаче объекта в метод в качестве параметра, а также может быть вызвано явным образом. Явное преобразование выполняется при вызове операции явного преобразования. По смыслу неявное преобразование описывается тогда, когда текущий тип преобразуется к более широкому типу, то есть уменьшения функциональности объекта не происходит. Явное преобразование описывается для преобразования текущего типа к более узкому типу. Операции преобразования типа описываются только в случае необходимости.

Ключевые слова `implicit` и `explicit` НЕ ВКЛЮЧАЮТСЯ в сигнатуру операции, то есть нельзя одно и то же преобразование описать и операцией явного преобразования и операцией неявного преобразования. Кроме того, для операций преобразования тип возвращаемого значения ВКЛЮЧЕН в сигнатуру для того, чтобы можно было определять преобразования к разным типам одного и того же объекта.

Деструктор.

Деструктор – это специальный метод, который вызывается сборщиком мусора непосредственно перед удалением объекта из динамической памяти.

Синтаксис деструктора:

```
~<Имя класса>()
{
    <тело деструктора>
}
```

Поскольку процессом удаления объекта программист не управляет, то в языке предоставлена возможность задать некоторые операторы в деструкторе. Необходимость такого подхода обусловлена тем, что даже с недоступным объектом в динамической памяти могут быть связаны

некоторые ресурсы, которые нужно корректно освобождать. Если во время работы программы сборщик мусора не запустился, то и деструктор не будет выполнен. Это стоит учитывать, поэтому в деструкторе прописываются аварийные действия при удалении объекта. Функциональные действия по освобождению связанных с объектом ресурсов лучше выполнять в методах объекта, а не в деструкторе.

Вложенные типы данных.

В составе класса можно объявлять внутренние классы. Эти типы данных являются вспомогательными и имеют смысл только в контексте содержащего их класса или объекта. Механизм описания вложенных классов позволяет скрывать ненужные детали внутреннего устройства классов и объектов и является реализаций принципа инкапсуляции. Для вложенных типов используются спецификаторы полей. В остальном они похожи на обычные не вложенные классы.

Метод Main.

Метод Main является точкой входа в программу, с него начинается выполнение программы. К этому моменту еще ни один метод не запущен, ни один объект не создан. Поэтому метод имеет спецификаторы public static, чтобы его могла запустить среда исполнения. Метод может возвращать значение целого типа или не возвращать значений (void), может не принимать параметров, или принимать в качестве параметра массив строк.

Существуют 4 формы описания метода:

```
public static void Main() {…}
public static int Main() {…}
public static void Main(string[] args) {…}
public static int Main(string[] args) {…}
```

Если метод не возвращает значений, то по умолчанию возвращается 0, среда исполнения интерпретирует возвращенный из метода 0 как успешное выполнение. Ненулевое значение означает ошибку исполнения метода. Если метод возвращает целое значение, то в случае успешного завершения 0 возвращается явным образом, в случае ошибки возвращается код ошибки.

В качестве параметра метод может принять только массив строк. Строки считываются из командной строки, в которой разделяются пробелами. В заголовке метода массив традиционно имеет имя args, хотя может иметь и любое другое имя. Принятые строковые параметры преобразуются в методе к тем типам, которые необходимы для работы метода.

Тема 5. Стандартные классы.

Массивы.

Массивы в C# делятся на три типа – одномерные, прямоугольные и ступенчатые. Массив в языке C# является ссылочным (объектным) типом данных. То есть описание массива, а также операции создания объекта массива и элементов массива – это три различных этапа. Массив может состоять из элементов значащего типа, или из элементов ссылочного типа. При создании объекта массива его элементам задаются значения по умолчанию – 0 для числовых типов, `null` для ссылочных. Поскольку массив является ссылочным типом данных, то ему присущи элементы, характерные для объектов – поля, методы и т.д. Состав базового класса `Array` для массивов будет рассмотрен позднее.

Переменные типа массив можно присваивать и сравнивать между собой. Присваивание и сравнение проводится по правилам ссылочных переменных. То есть присваиваются и сравниваются ссылки на массивы, а не элементы массивов.

Одномерные массивы

Одномерный массив – это ограниченная последовательность однотипных данных, имеющих одно имя и предполагающих обращение к элементам последовательности по их номеру (индексу).

Описание одномерного массива:

```
<тип>[] <имя>; //описание ссылки на массив
int[] arr;//описание ссылки на массив из элементов целого типа
```

При описании ссылки на массив объект массива не создается, а ссылке присваивается значение `null`. Объект массива создается явно с помощью операции new. Тогда в динамической памяти выделяется место для хранения массива, и только на этом этапе задается количество элементов массива (длина массива).

```
int[] arr = new int[10];/*объявление и создание массива из 10 целых чисел*/
```

Количество элементов массива может быть задано целым положительным числом. Также количество элементов массива может быть задано вычислимым выражением, результат которого имеет тип, приводимый к `int`, `uint`, `long` или `ulong`.

Как правило, для задания максимального индекса элемента в массиве используется его поле `Length`, хранящее количество элементов в массиве.

Элементы массива всегда нумеруются с 0, то есть последний элемент массива имеет номер, на единицу меньший, чем длина массива. Элементы массива инициализируются явно, как правило, с помощью циклических операторов, например:

```
int[] arr = new int[10];
```

```
for (int i = 0; i < arr.Length; i++)
{
  arr[i] = i;
}
```

С элементами массива можно выполнять все операции, допустимые для соответствующего им типа данных.

Для полного задания массива должно быть известно следующее: тип элементов, имя массива, количество элементов, значения элементов. В связи с этим возможны несколько вариантов описания и инициализации массива (если заранее известны значения его элементов):

```
int[] arr = new int[4] {1,2,3,4};//самая полная форма
int[] arr = new int[] {1,2,3,4};//нет задания длины
int[] arr = {1, 2, 3, 4}; //сокращенная форма
```

Все три примера описывают задание одного и того же массива. Все три формы являются равнозначно применимыми.

Пример работы с массивом – ввод элементов массива с клавиатуры и вывод элементов на экран:

```
Console.WriteLine(”Введите длину массива”);
int length = Int32.Parse(Console.ReadLine());
int[] arr = new int[len]; /*объявление ссылки на массив и создание объекта массива*/
for (int i = 0; i < arr.Length; i++)//ввод массива
{
  Console.Write(”Введите {0}-й элемент массива:”, i);
  arr[i]= Int32.Parse(Console.ReadLine()); /*поэлементный ввод*/
}
for (int i = 0; i < arr.Length-1; i++)//вывод массива
{
  Console.Write(”{0}, ”, arr[i]); /*вывод через «,» всех элементов, кроме последнего*/
}
Console.WriteLine(arr[arr.Length-1]); /*вывод последнего элемента и перевод на новую строку*/
```

Прямоугольные массивы

Прямоугольные массивы имеют более одного измерения. В двумерном случае, наиболее часто встречающемся, массив представляет собой прямоугольник (матрицу). В трехмерном случае – параллелепипед, и т.д.

Будем рассматривать многомерные прямоугольные массивы на примере двумерных.

Объявление ссылки на двумерный массив элементов целого типа:

```
int[,] arr; // arr = null
```

Объявление и создание объекта двумерного массива:

```
int[,] arr = new int[2,3];
```

Первым указывается число строк, вторым – число столбцов матрицы. И строки и столбцы нумеруются с 0. В данном примере значения элементов по умолчанию задаются равными 0.

При известных значениях элементов описать массив можно одним из следующих способов:

```
int[,] arr = new int[2,3];
int[,] arr = new int[2,3] {{1,2,3}, {4,5,6}}; /*матрица из двух строк и трех столбцов*/
int[,] arr = new int[,] {{1,2,3}, {4,5,6}}; /*матрица из двух строк и трех столбцов без указания размерности*/
int[,] arr = {{1,2,3},{4,5,6}}; //сокращенная форма
```

Обработка двумерных массивов производится, как правило, с помощью двух вложенных циклов. Пример ввода двумерного массива с клавиатуры и вывода на экран:

```
Console.WriteLine(”Введите число строк массива”);
int rows = Int32.Parse(Console.ReadLine());
Console.WriteLine(”Введите число столбцов массива”);
int columns = Int32.Parse(Console.ReadLine());
int[,] arr = new int[rows,columns]; /*объявление ссылки на массив и создание объекта массива*/
for (int i = 0; i < rows; i++)//построчный ввод массива
{
   for (int j = 0; j < columns; j++)
   {
      Console.Write(”Введите {0}-й элемент {1}-й строки:”, j, i);
      arr[i,j]= Int32.Parse(Console.ReadLine());//ввод элемента
   }
}
for (int i = 0; i < rows; i++)//построчный вывод массива
{
   for (int j = 0; j < columns-1; j++)
   {
      Console.Write(”{0}, ”, arr[i,j]); /*вывод через «,» всех элементов, кроме последнего*/
   }
   Console.WriteLine(arr[i,columns-1]);     /*вывод     последнего элемента и перевод на новую строку*/
}
```

Работа с массивами размерностью больше двух ведется аналогичным образом, только увеличивается количество индексов и уровень вложенности циклов обработки.

Ступенчатые массивы

Ступенчатый массив – это одномерный массив ступенчатых массивов, размерность которых на единицу меньше. То есть двумерный ступенчатый массив – это одномерный массив одномерных массивов, трехмерный ступенчатый массив – это одномерный массив двумерных ступенчатых массивов, каждый из которых является одномерным массивом и т.д. На

рисунке 2 представлена в общем виде структура двумерного ступенчатого массива.

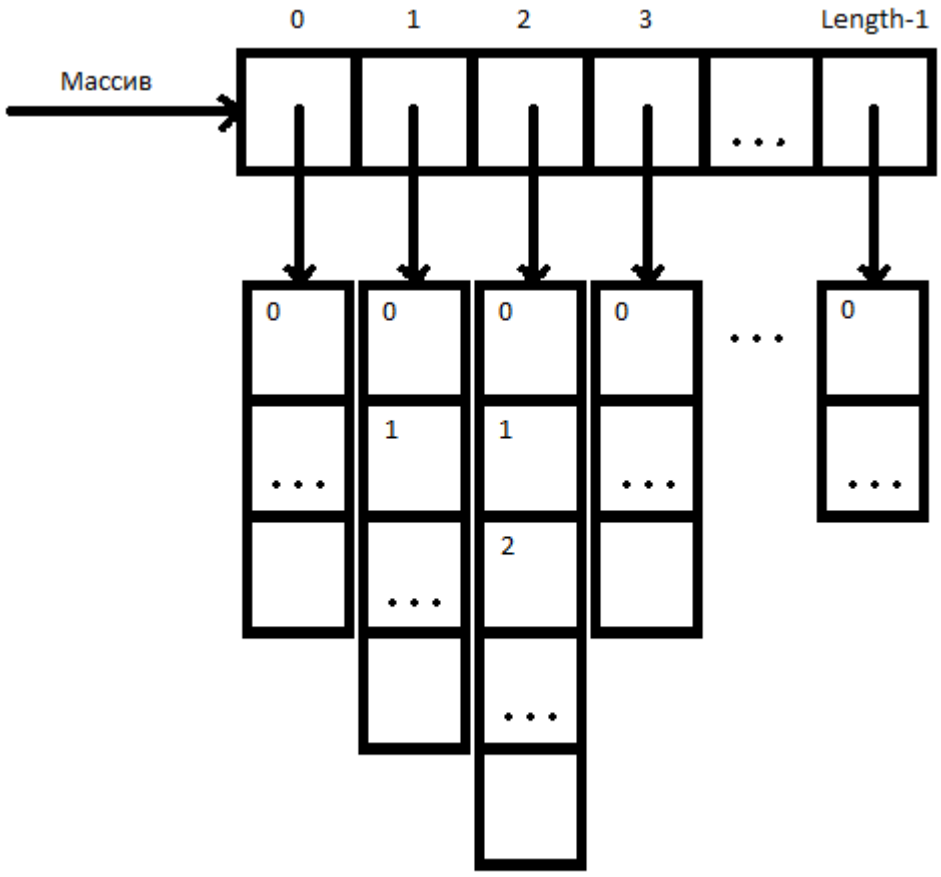


Рисунок 2. Структура двумерного ступенчатого массива

То есть ступенчатый массив – это одномерный массив массивов с различным количеством элементов.

Объявление ссылки на двумерный ступенчатый массив элементов целого типа:

```
int[][] arr; // arr = null
```

Объявление и создание объекта двумерного ступенчатого массива:

```
int[][] arr = new int[2][];
```

Таким образом, сначала создается объект одномерного массива заданной длины, а каждый из массивов ступенчатого массива нужно создавать отдельно:

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

В данном примере объявлен ступенчатый массив, состоящий из двух одномерных массивов целых элементов, первый из которых с номером 0

состоит из трех элементов, а второй с номером 1 состоит из четырех элементов.

Массив можно также задать в сокращенной форме:

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

или, если значения элементов известны:

```
int[][] arr = {{1,2,3}, {1,2,3,4}};
```

Обращение к элементу производится по двойному индексу

```
arr[i][j]
```

Многомерные ступенчатые массивы строятся аналогично двумерным ступенчатым. Вообще, ступенчатые массивы похожи на прямоугольные, и работа с ними производится теми же циклическими конструкциями различного уровня вложенности.

Класс Array.

Класс `Array` находится в пространстве имен `System` и является базовым для всех массивов. То есть все массивы обладают функциональностью, описанной в этом классе. Элементы класса приведены в таблице.

Элемент	**Описание**	**Вид**
Length	Количество элементов в массиве	Свойство (поле)
Rank	Размерность массива	Свойство (поле)
BinarySearch	Двоичный поиск, массив считается сортированным по возрастанию	Статический метод
Clear	Очистка массива (присвоение элементам значений по умолчанию)	Статический метод
Copy	Копирование заданного диапазона элементов (или всех элементов) в другой массив	Статический метод
CopyTo	Копирование всех элементов одномерного массива в другой одномерный массив	Метод
GetValue	Получение элемента массива по индексу	Метод
IndexOf	Поиск первого вхождения значения в одномерный массив	Статический метод
LastIndexOf	Поиск последнего вхождения значения в одномерный массив	Статический метод
Reverse	Реверс (изменение порядка элементов в массиве на	Статический метод

	обратный)	
SetValue	Установка значения элемента по индексу	Метод
Sort	Сортировка массива по возрастанию	Статический метод

Класс `Array` является базовым для всех массивов, значит, любой массив неявно преобразуем к типу `Array`. Но стоит заметить, что класс `Array` не поддерживает индексацию. То есть, если на массив есть только ссылка типа `Array`, то к элементам массива нет доступа с помощью операции «[ ]», а только с помощью методов GetValue и SetValue,

Пример:

```
int[] arr = {2,5,3,1,4};
for (int i = 0; i < arr.Length; i++) //вывод массива
{
   Console.Write(arr[i]+” ”);
}
Console.Write(”\n”);
Array.Reverse(arr); // реверс массива
for (int i = 0; i < arr.Length; i++) /*вывод реверсированного
массива*/
{
   Console.Write(arr[i]+” ”);
}
Console.Write(”\n”); //перевод на новую строку
Array.Sort(arr); //сортировка массива по возрастанию
for (int i = 0; i < arr.Length; i++) /*вывод отсортированного
массива*/
{
   Console.Write(arr[i]+” ”);
}
Console.Write(”\n”);
Array a = arr;//ссылка a имеет тип Array,ссылается на массив arr
for (int i = 0; i < a.Length; i++) /*вывод массива arr через
ссылку a, индексация не поддерживается*/
{
   Console.Write(a.GetValue(i)+” ”);
}
Console.Write(”\n”);
```

Цикл foreach для работы с массивами.

Цикл `foreach` был описан в теме 2. Для работы с массивами цикл применяется тогда, когда элементы массива не нужно изменять. То есть для вывода или поиска.

Пример: Вывод массива с помощью цикла `foreach`.

```
int[] arr = {2,5,3,1,4};
foreach (int element in arr)
{
```

```
   Console.Write(elem+ ” ”);
}
Console.Write(”\n”); //перевод на новую строку
```

Массивы объектов.

Как уже говорилось, массив может содержать элементы не только значащего типа. В массиве могут содержаться элементы класса, описанного пользователем. Такой массив называют массивом объектов. Его описание и создание похоже на создание ступенчатого массива – и сам массив и объекты нужно создавать явным образом с помощью операции `new`.

```
class MyClass
{
   …
}
…
MyClass[] arrayOfMyClass = new MyClass[3];
arrayOfMyClass[0] = new MyClass();
arrayOfMyClass[1] = new MyClass();
arrayOfMyClass[2] = new MyClass();
```

Объекты в массиве создаются вызовом конструктора. Конечно, для инициализации объектов можно использоваться любой конструктор, имеющийся в классе.

Символы

Тип `char` предназначен для хранения символа в кодировке `Unicode`. Класс `Char` из пространства имен `System` является классом-оберткой для типа `char` и предоставляет значимому типу объектные возможности. В классе определены следующие методы, все они являются статическими.

Метод	Описание
GetNumericValue	Возвращает числовое значение цифры или -1, если символ не цифра
GetUnicodeCategory	Возвращает категорию символа
IsControl	Возвращает true, если символ является управляющим
IsDigit	Возвращает true, если символ – цифра
IsLetter	Возвращает true, если символ - буква
IsLetterOrDigit	Возвращает true, если символ – буква или цифра
IsLower	Возвращает true, если символ принадлежит нижнему регистру (строчный)
IsNumber	Возвращает true, если символ – цифра (десятичная или шестнадцатеричная)
IsPunctuation	Возвращает true, если символ – знак препинания
IsSeparator	Возвращает true, если символ – разделитель

IsUpper	Возвращает true, если символ принадлежит верхнему регистру (прописной)
IsWhiteSpace	Возвращает true, если символ – пробельный
Parse	Преобразует односимвольную строки в символ
ToLower	Преобразует символ к нижнему регистру
ToUpper	Преобразует символ к верхнему регистру
MaxValue	Возвращает символ с максимальным кодом
MinValue	Возвращает символ с минимальным кодом

Пример:

```
…
char a = 'A', b = '\x63', c = '\u0032';
Console.    WriteLine("{0},    {1},    {2}",    Char.ToLower(a),
Char.ToUpper(b), Char.GetNumericValue(c));
char d;
for (;;)
{
   Console.WriteLine("Введите символ");
   d = Char.Parse(Console.ReadLine());
   if (Char.IsLetter(d)) Console.WriteLine("Буква");
   if (Char.IsDigit(d)) Console.WriteLine("Цифра");
   if (Char.IsUpper(d)) Console.WriteLine("Верхний регистр");
   if (Char.IsLower(d)) Console.WriteLine("Нижний регистр");
   if (Char.IsPuctuation(d))Console.WriteLine("Знак препинания");
   if (d == '0') return;
}
…
```

Символы могут объединяться в массивы. Массив символов, как и любой массив, является объектом класса `Array`. К нему применимы все операции, характерные для массивов. Массив символом НЕ ЯВЛЯЕТСЯ строкой.

Строки

Строковый тип String

Тип `string` относится ко встроенным типам языка C#, является ссылочным типом. Все переменные типа `string` являются объектами класса `String` из пространства имен `System` и хранят набор символов в кодировке `Unicode`. Строка обладает упрощенным синтаксисом создания, то есть операция `new` при создании объекта строки не пишется, если строка инициализируется строковым литералом:

```
string s = "Мама мыла раму";
```

Существует еще несколько явных способов создания и инициализации строки:

```
string s = new string('a', 20);//строка из 20 символов «a»
char[] a = new {'a', 'b', 'c'};
```

```
string s = new string(a);
```

Строки относятся к неизменяемым типам данных. То есть при выполнении методов и операций, изменяющихся строку, создается новый экземпляр строки (а старый остается недоступным в динамической памяти и потом убирается сборщиком мусора).

Операции со строками:

- присваивание - «=»;
- проверка на равенство – «==»;
- проверка на неравенство – «!=»;
- конкатенация – «+»;
- чтение по индексу – «[ ]»;

При сравнении строки сравниваются по значению, несмотря на то, что являются ссылочным типом данных. Строки считаются равными, если имеют одинаковую длину и равны посимвольно. Во всех остальных случаях строки считаются неравными.

В классе String описаны следующие элементы, предоставляющие строке объектную функциональность:

Элемент	**Описание**	**Вид**
Compare	Сравнение двух строк	Статический метод
CompareTo	Сравнение текущей строки с другой строкой	Метод
Concat	Склейка произвольного числа строк	Статический метод
Copy	Создание копии строки	Статический метод
Empty	Пустая строка	Статическое поле
IndexOf	Индекс первого вхождения подстроки (символа) в строку	Метод
LastIndexOf	Индекс последнего вхождения подстроки (символа) в строку	Метод
IndexOfAny	Индекс первого вхождения любого из набора символов в строку	Метод
LastIndexOfAny	Индекс последнего вхождения любого из набора символов в строку	Метод
Insert	Вставка подстроки в заданную позицию строки	Метод
Length	Длина строки – число символов в строке	Свойство
PadLeft	Добавление заданного числа пробелов в начало строки	Метод
PadRight	Добавление заданного числа	Метод

	пробелов в конец строки	
TrimStart	Удаление пробела из начала строки	Метод
TrimEnd	Удаление пробела из конца строки	Метод
Trim	Удаление пробелов из начала и из конца строки	Метод
StartsWith	Возвращает `true`, если строка начинается с заданной подстроки	Метод
EndsWith	Возвращает `true`, если строка заканчивается заданной подстрокой	Метод
Remove	Удаление подстроки заданной длины из заданной позиции строки	Метод
Replace	Замена всех вхождений подстроки (символа) другой подстрокой (символом)	Метод
Substring	Копирование подстроки с заданной позиции в строке	Метод
ToCharArray	Преобразование строки в массив символов	Метод
ToLower	Преобразование всех символов в строке к нижнему регистру	Метод
ToUpper	Преобразование всех символов в строке к верхнему регистру	Метод
Split	Разделение строки на массив строк (слов), разделители указываются в параметрах метода	Метод
Join	Преобразование массив строк (слов) в одну строку через указанный разделитель	Статический метод

Следует отметить, что все методы изменения строки возвращают в качестве результата новую строку. Старая строка остается неизменной.

Пример.

```
…
string s = ”Мама мыла раму”;
Console.WriteLine(s);
string s1 = s.Substring(5); //копируется подстрока с 5 символа
Console.WriteLine(s1);
string s2 = s.Remove(4,5);
Console.WriteLine(s2);
string[] words = s.Split(’ ’);
```

```
string res = String.Join(';', words);
Console.WriteLine(res);
…
```

Строковый тип StringBuilder

Поскольку строка `String` относится к неизменяемым типам данных, то при многочисленных преобразованиях каждый раз создается новый строковый объект. То есть память расходуется нерационально. Для работы со строками, требующими многочисленных изменений, используются объекты класса `StringBuilder`. Этот класс принадлежит пространству имен `System.Text`. Строка `StringBuilder` требует обязательного явного создания экземпляра. В классе описано несколько перегруженных конструкторов.

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder(”Мама мыла раму”);
StringBuilder sb3 = new StringBuilder(20);
StringBuilder sb4 = new StringBuilder(”Мама мыла раму”, 20);
StringBuilder sb5 = new StringBuilder(”Мама мыла раму”,0,4,20);
```

Второй пример – создание строки типа `StringBuilder` с инициализацией строковым литералом типа `String`. Третий пример – создание объекта типа `StringBuilder` размером 20 байт. Четвертый пример – создание объекта с явной инициализацией и указанием размера. Пятый пример – объект инициализируется подстрокой в 4 символа, начиная с нулевого, и размером 20 байт. В первом примере, когда не задана инициализация и не задан размер, создается пустая строка по умолчанию размером 16 байт.

Состав класса `StringBuilder`.

Элемент	**Описание**	**Вид**
Append	Добавление в конец строки	Метод
Capacity	Емкость буфера (размер строки)	Свойство
Insert	Вставка подстроки в заданную позицию строки	Метод
Length	Количество символов в строке (длина строки)	Свойство
MaxCapacity	Максимально возможная емкость буфера	Свойство
Remove	Удаление подстроки	Метод
Replace	Замена всех вхождений подстроки (символа) другой подстрокой (символом)	Метод
ToString	Преобразование к строке типа String	Метод

Следует отметить, что методы изменения строки редактируют текущую строку без порождения новых объектов.

Пример.

```
Console.WriteLine(”Введите строку”);
Console.ReadLine();
StringBuilder s = new StringBuilder();
s.Append(”Вы ввели строку: ”);
s.Append(Console.ReadLine());
Console.WriteLine(s);
s.Replace(’a’, ’o’);
Console.WriteLine(s);
```

Класс Random/

Этот класс описывает генератор случайных последовательностей (ГСП) и принадлежит пространству имен `System`. Создание ГСП возможно с помощью двух конструкторов:

```
Random gen1 = new Random(); // 1
Random gen2 = new Random(1); // 2
```

В первом примере инициализация ГСП производится от текущего момента времени, при каждом новом запуске программы генерируется уникальная последовательность. Во втором примере ГСП инициализируется целым числом, при каждом запуске программы последовательность будет повторяться.

Методы класса Random для получения случайных чисел:

- Next() – возвращает случайное число типа int из всего диапазона возможных значений;
- Next(<max>) – возвращает случайное целое число из диапазона [0, max);
- Next(<min>, <max>) – возвращает случайное целое число из диапазона [min, max);
- NextBytes(array) – заполняет массив случайными числами типа byte из диапазона [0, 255];
- NextDouble() – возвращает случайное вещественное число из диапазона [0, 1).

Пример.

```
…
Random gen = new Random(); //создание ГСП
for (int i = 0; i < 10; i++)
Console.Write(”{0} ”, gen.Next(1000)); //вывод 10 чисел < 1000
Console.WriteLine();
byte[] ar = new byte[10];//создание массива байт
a.NextBytes(ar);//заполнение массива байт
for (int i = 0; i < ar.Length; i++)
Console.Write(”{0} ”, ar[i]);
Console.WriteLine();
```

```
Console.WriteLine(gen.NextDouble());//вывод вещественного числа
…
```

Класс Object.

Класс `Object` является родительским классом для всех классов библиотеки .NET. Все дочерние классы наследуют элементы класса `Object`. Наследники могут пользоваться базовыми реализациями методов класса, либо переопределять некоторые из них.

Состав класса `Object`.

- `public virtual bool Equals(object o)` – проверка на эквивалентность текущего объекта и переданного параметра. Базовая реализация возвращает true, если текущий объект и переданный параметр являются ссылками на один и тот же объект (для ссылочных типов данных), или если текущее значение равно переданному значению (для значащих типов данных). Метод можно переопределять в потомках для задания собственного признака эквивалентности объектов;
- `public virtual int GetHashCode()` – метод, возвращающий хэш-код текущего объекта. Хэш-код – это целое число, идентифицирующее объект. У эквивалентных объектов хэш-коды должны быть равны, поэтому этот метод переопределяется вместе с методом `Equals`;
- `public static bool Equals(object o1, object o2)` – метод возвращает true, если ссылки `o1` и `o2` ссылаются на один и тот же объект (для ссылочных типов данных), или значение `o1` равно значению `o2` (для значащих типов данных). Метод является статическим и не предназначен для переопределения;
- `public Type GetType()` – метод возвращает представление класса, к которому принадлежит текущий объект (в виде объекта класса `Type`);
- `public virtual string ToString()` – метод преобразует содержимое объекта в строку. Базовая реализация метода возвращает в виде строки тип объекта и адрес его размещения в памяти, метод можно переопределить в потомках для формирования строкового представления объекта.

Тема 6. Исключительные ситуации.

Во время выполнения любой программы возможно возникновение ошибок – некорректность ввода, деление на 0, выход за границу группы, разрыв соединения и т.д. Эти ошибки нужно каким-то образом обрабатывать. Существуют различные подходы к обработке ошибок исполнения, например, прекращение работы программы с возвращением кода ошибки (как в языка Паскаль), или возвращение из функции кода ошибки для дальнейшей его программной обработки (как в языке СИ). В объектно-ориентированных языках оперируют понятием *исключительной ситуации* – ошибки исполнения программы.

Исключительные ситуации делятся на: предсказуемые и непредсказуемые, устранимые и неустранимые. Предсказуемые исключительные ситуации программист может предположить, исходя из написанного программного кода, непредсказуемые трудно предположить, глядя на код, последствия неустранимых исключительных ситуаций критичны, они завершают программу, а устранимые исключительные ситуации позволяют программисту произвести ряд восстановительных действий без прекращения работы программы.

Механизм исключительных ситуаций позволяет логически отделить момент обнаружения ошибки от процесса ее обработки, то есть появляется ошибка в одном блоке кода, а вот обрабатывается в другом.

Мы будем иметь дело с предсказуемыми устранимыми исключительными ситуациями.

При возникновении ошибки исполнения программы в динамической памяти создается особый объект – *исключение*. Его основное назначение – прервать работу того метода, который привел к ошибке. Говорят, что исключение «выбрасывается». Исключение может быть выброшено как средой исполнения, так и самим программистом.

Все исключения принадлежат одной иерархии наследования, корнем которой является класс `Exception`, о нем поговорим чуть позже. Классы исключений именуются следующим образом: <НазваниеОшибки>Exception. Приведем примеры наиболее часто используемых исключений.

- `ArithmeticException` - арифметические ошибки, класс является базовым для `DivideByZeroException` и `OverFlowException`;
- `ArrayTypeMismatchException` – запись в массив элемента несоответствующего типа;
- `DivideByZeroException` – деление на 0;
- `FormatException` – параметр неверного типа;
- `IndexOutOfRangeException` – выход за границу массива;
- `InvalidCastException` – ошибка преобразования типа;

– `OutOfMemoryException` – недостаточно памяти для создания объекта;
– `OverFlowException` – арифметическое переполнение;
– `StackOverFlowException` – переполнение стэка.

Конечно, это не все классы исключений, в языке их гораздо больше, мы будем с ними знакомиться по мере изучения курса.

Выброшенный объект исключения нужно отлавливать с помощью специального оператора.

Оператор try.

Оператор состоит из трех частей.

```
try
{
  <контролируемый блок>
}
catch()
{
  <обработка исключения>
}
finally
{
  <финальный блок>
}
```

Блок `try` содержит код, который *может* выбросить исключение. Блок `catch` предназначен для отлова и обработки выброшенного исключения. Блок `finally` будет выполняться в любом случае, независимо от того, было ли выброшено исключение в блоке `try`. Отсутствовать может или блок `catch` или `finally`, но не оба сразу.

Блоков `catch` может быть несколько. Блоки catch могут быть записаны в следующих трех формах:

1.

```
catch (<тип исключения> <имя>)
{
  <обработка объекта исключения>;
}
```

Объекту исключения присваивается имя, по этому имени объект и его содержимое доступно в теле блока обработки.

2.

```
catch (<тип исключения>)
{
  <обработка исключения>;
}
```

Обработка исключения не предполагает доступа к самому объекту исключения, в теле обработчика известна информация только о типе исключения.

3.

```
catch
{
  <обработка исключения>;
}
```

Данный вид предназначен для единообразной обработки любых исключений.

Все три вида блоков `catch` могут соседствовать друг с другом в операторе `try`.

Блоки `catch` по сути представляют собой обработчики соответствующих типов исключений. При возникновении исключения блоки `catch` просматриваются один раз по порядку до тех пор, пока объект исключения не *подойдет* по типу к указанному в параметре блока `catch`. Как только такое совпадение найдено – выполняется соответствующий обработчик исключения. То есть в начале списка `catch` типы исключений следует указывать с самого низа иерархии наследования исключений. Сначала «самые» дочерние, а уж потом родительские типы исключений. Третий вид, как правило, записывается самым последним, используется для отлова всех еще не отловленных исключений. Запись его в начале или середине списка нецелесообразна.

Блок `finally`, если он есть, выполняется всегда. В блоке `finally` обычно выполняются действия, связанные с освобождением ресурсов, поскольку эти действия должны быть выполнены независимо от того, возникла ошибка или нет.

Порядок обработки исключений.

1. Потенциально опасный код в блоке `try` выбрасывает исключение. Выполнение блока прекращается (так как он привел к ошибке), далее переход к пункту 2. Если ошибок не было, то код выполняется корректно, переход к пункту 3.

2. Ищется обработчик возникшего исключения в списке `catch`. Если таковой найден, то переход к пункту 3. Если не найден, то исключение передается на более высокий уровень по стэку вызовов (в крайнем случае работа программы завершается с выводом информации об ошибке).

3. Если блок `finally` есть, то он выполняется. Затем управление передается оператору, следующему за `try`.

Пример.

```
int a, b, res;
try
{
   Console.Write(”Введите делимое: ”);
   a = Int32.Parse(Console.ReadLine());
   Console.Write(”Введите делитель: ”);
```

```
   b = Int32.Parse(Console.ReadLine());
   res = a/b;
   Console.WriteLine(”Частное = ” + res);
}
catch (FormatException e)
{
   Console.WriteLine(”Введено не целое число.” + e.Message);
}
catch (DivideByZero)
{
   Console.WriteLine(”Деление на 0!”);
}
catch
{
   ConsoleWriteLine(”Неизвестная ошибка!”);
}
```

Операторы `try` могут быть вложенными. Исключение может возникнуть в одном из вложенных блоков. Если оно не перехватывается во вложенном `try/catch/finally`, то передается на один уровень выше по стэку вызовов, и обработчик ищется там. И т.д. до метода `Main`. Этот механизм называется *распространением*, или «пробросом» исключения. Распространение позволяет разделить этапы обработки ошибок, то есть на нижнем уровне произвести частичную обработку, а затем пробросить исключение выше для полной обработки.

Можно выделить три подхода к обработке исключительных ситуаций.

1. Полная обработка исключения там, где оно было выброшено.

2. Отсутствие обработки, и неявный проброс исключения выше.

3. Промежуточный подход. Перехват исключения, его частичная обработка и дальнейший проброс старого объекта или создание нового исключения взамен старого. Этот подход используется тогда, когда на текущий момент неизвестно, какие восстановительные действия можно произвести.

Генерирование исключений.

Принудительный выброс исключений выполняется с помощью ключевого слова `throw`. Оператор применяется в двух формах.

1. `throw`;

Такая форма применяется для принудительного проброса уже пойманного объекта исключения.

2. `throw <объект исключения>;`

Форма применяется для создания нового объекта исключения. Объект исключения создается с помощью оператора `new`, как и любой другой объект.

Пример.

```
class Program
{
```

```
   static void Method1()
   {
      try
      {
        Method2();
      }
      catch (Exceprion e)
      {
        Console.WriteLine(”Исключение, пойманное в методе1 ” +
e.Message);
        throw;// проброс пойманного исключения
      }
   }
   static void Method2()
   {
      throw new Exception(”Исключение, сгенерированное в методе2
”);
   }
   static void Main()
   {
      try
      {
          Method1();
      }
      catch (Exceprion e)
      {
          Console.WriteLine(”Исключение, пойманное в Main” +
e.Message);
      }
   }
}
```

При выполнении данного кода на экран будет выведено следующее:

```
Исключение, пойманное в методе1 Исключение, сгенерированное в
методе2
Исключение, пойманное в Main Исключение, сгенерированное в
методе2
```

Модифицируем пример, заменив проброс исключения в `Method1` выбросом нового исключения (остальной код остается без изменений):

```
static void Method1()
{
   try
   {
     Method2();
   }
   catch (Exceprion e)
   {
     Console.WriteLine(”Исключение, пойманное в методе1 ” +
e.Message);
     throw new Exception(”Исключение, сгенерированное в
методе1”);// выброс нового исключения
}
```

Результат работы будет следующим:

```
Исключение, пойманное в методе1 Исключение, сгенерированное в методе2
Исключение, пойманное в Main Исключение, сгенерированное в методе1
```

Как видим, теперь метод Main обрабатывает новое исключение, выброшенное в `Method1` и не знает о существовании исключения, выброшенного `Method2`.

Класс Exception.

Класс является базовым для всех исключений. В состав класса входят следующие нестатические свойства:

Свойство	Описание
Message	Текстовое описание ошибки, устанавливается при создании объекта. Свойство только для чтения.
Source	Имя объекта или приложения, сгенерировавшего исключение
HelpLink	Путь к файлу справки по исключению
StackTrace	Последовательность вызовов, приведших к исключению. Свойство только для чтения.
InnerException	Ссылка на исключение, явившееся причиной выброса текущего исключения.
TargetSite	Метод, выбросивший исключение.

В классах-потомках могут быть описаны дополнительные элементы.

Следует отметить, что большого смысла в расширенном составе класса исключения нет, поскольку главной является информация о классе сгенерированного исключения, а не о составе класса. Поскольку класс, которому принадлежит исключение, и есть главное описание ошибки.

Пользовательские исключения.

Несмотря на довольно обширную иерархию классов исключений, пользователь может описать свои классы. Как правило, пользовательские исключения описываются для уточнения ошибки, принадлежащей стандартному классу. Конечно, ничто не мешает добавить в класс дополнительные элементы, но достаточно просто объявить имя класса и его предка, и перегрузить два конструктора:

```
class MyException: Exception
{
     public MyException():base()
     {
     }
     public MyException(string message):base(message)
     {
     }
```

```
}
```

Следует отметить, что механизм исключений довольно громоздкий и медленно исполняемый.

Тема 7. Реализация принципов ООП в C#.

Реализация инкапсуляции в языке осуществляется с помощью спецификаторов доступа. Все приватные элементы класса относятся к его внутренней реализации, то есть напрямую доступны только внутри того класса, в котором описаны. Все неприватные элементы класса относятся к его внешнему представлению, то есть позволяют обращение из внешних объектов и классов.

Рассмотрим реализацию остальных принципов ООП в языке C#.

Наследование.

При решении сложных задач часто возникает необходимость в описании большого количества классов, многие из которых имеют некоторую общую функциональность. Такой набор классов нуждается в упорядочении и структурировании. В этом помогает реализация принципа наследования.

Пример. Опишем класс студентов и класс преподавателей.

```
class Student
{
   string name; //фамилия
   string address; //адрес
   DateTime birthday; //дата рождения
   int id; //номер зачетной книжки
   int group; //номер группы
}
class Teacher
{
   string name; //фамилия
   string address; //адрес
   DateTime birthday; //дата рождения
   string department; //кафедра
   int salary; //зарплата
}
```

Конечно, состав классов можно серьезно расширить, однако даже в таком простом примере видно, что в обоих классах есть повторяющиеся элементы структуры. Выделим эту общность в отдельный класс, а оба существующих класса сделаем дочерними от нового:

```
class Person
{
   string name; //фамилия
   string address; //адрес
   DateTime birthday; //дата рождения
}
class Student:Person
{
   int id; //номер зачетной книжки
   int group; //номер группы
```

```
}
class Teacher:Person
{
   string department; //кафедра
   int salary; //зарплата
}
```

Как видно, добавление в иерархию нового класса, являющегося родительским для обоих сущностных классов, позволяет исключить повторяющиеся блоки кода, упростить модификацию классов, упростить расширение иерархии классов. Всего этого позволяет достичь наследование.

Для классов в языке C# реализовано одиночное наследование, то есть класс может иметь только одного предка, а вот количество потомков не регламентируется и определяется необходимостью и здравым смыслом.

Родительский класс указывается в заголовке при описании класса (см. пример), если предок явно не указан, то класс считается наследующим от базового класса `Object`.

На этапе исполнения объект дочернего типа представляет собой единое целое с унаследованной родительской частью. А при описании дочернего класса в его теле нет непосредственного доступа к приватным элементам, унаследованным от родителя (реализация инкапсуляции).

При наследовании дочерний класс использует структуру родительского класса, то есть наследует все его элементы, и приватные, и публичные.

Конструкторы не наследуются!

Класс-потомок должен иметь свои собственные конструкторы для инициализации всех своих полей. В том числе и унаследованных от родителя, в том числе и приватных. Непосредственно из конструктора потомка, как уже говорилось, доступа к приватным полям, унаследованным от предка, нет, однако конструктор дочернего класса вызывает конструктор родительского класса – явно или неявно. Если явного вызова конструктора родительского класса нет, то неявно (по умолчанию) вызывается конструктор по умолчанию родительского класса. Если же конструктора по умолчанию в классе-предке не описано, то в конструкторе класса-потомка возникает необходимость явного вызова какого-то из конструкторов родительского класса. Явный вызов конструктора класса-предка осуществляется с помощью ключевого слова base (поэтому родительский класс иногда называют базовым классом). И явный и неявный вызов конструктора базового класса осуществляется ДО начала выполнения операторов в теле конструктора дочернего класса. При иерархии наследования нескольких уровней конструкторы вызываются последовательно, начиная с корневого предка и заканчивая потомком. Каждый из этих конструкторов инициализирует свой кусочек создаваемого объекта.

Пример.

```
class BaseClass // родительский класс
{
   int a; // поле объекта родительского класса
```

```
   public BaseClass() //конструктор по умолчанию родительского класса
   {
      a = 1;
   }
   public BaseClass(int a) //конструктор с целым параметром
   {
      this.a = a;
   }
}

class MyClass: BaseClass // дочерний класс
{
   double b;         //поле объекта дочернего класса
   public MyClass() //конструктор неявно вызывает конструктор по
   {     //умолчанию родительского класса, задающие значение поля a
      b = 1;         //инициализация поля дочернего класса
   }
   public MyClass(int a, double b):base(a)
   //конструктор явно вызывает конструктор с параметром родительского класса
      this.b = b;    //инициализация поля дочернего класса
   }
}
```

В приведенном примере представлен неявный вызов конструктора по умолчанию из родительского класса для инициализации поля, унаследованного от родителя, значением, заданным в самом конструкторе предка. А также явный вызов конструктора с параметром из родительского класса для инициализации поля, унаследованного от предка значением, полученным извне в конструктор потомка.

Все остальные элементы класса-предка наследуются классом-потомком. Для этих элементов существуют свои правила наследования.

Наследование полей.

Поля предка в полном составе наследуются потомком. При этом возможны некоторые особенности. Если и унаследованные, и описанные в потомке поля различны по именам, то никакого конфликта имен не возникает, поля доступны в потомке, если поля предка были приватны, то их можно явно вызвать по ссылке `this`. Если в классе-потомке описывается поле, совпадающее по имени с полем, унаследованным от предка, то ему необходимо задать спецификатор `new`, чтобы указать, что это именно новое поле. При этом старое поле не перестает существовать, поскольку поля являются важной частью внутренней реализации класса, и их нельзя терять. Старое поле остается доступным по ссылке `base`. Если спецификатор `new` для нового поля не указывать, то ошибки компиляции не будет, компилятор выдаст предупреждение о совпадении имен. Причем, приватные поля предка явно в потомке не видны, и доступны только через методы доступа. Но если к ним обращаться через ссылку родительского типа `base`, то они доступны напрямую. Таким образом, наследование – это возможное нарушение инкапсуляции.

Наследование методов.

Методы наследуются аналогично полям. Конечно, при наследовании методов есть некоторые особенности, которые мы и рассмотрим.

Если в классе-потомке описывается метод, не совпадающий по сигнатуре с методом, унаследованным от класса-предка, то в потомке существуют оба метода. Конфликта имен и сигнатур не происходит.

Если в классе-потомке описать метод, совпадающий по сигнатуре с методом, унаследованным от предка, то его необходимо маркировать спецификатором `new`. При отсутствии спецификатора возникнет предупреждение, а не ошибка компиляции. При этом старая версия метода в теле класса-потомка доступна по ссылке `base`. Следует заметить, что при работе с объектами таких классов на исполнение будет вызваться та версия метода, которая соответствует типу ссылки на объект, а не настоящему типу объекта. То есть, если на объект дочернего типа направлена ссылка дочернего типа, то на исполнение вызывается версия метода, определенная в дочернем классе; если на объект дочернего типа направлена ссылка родительского типа, то на исполнение будет вызван метод, определенный в родительском классе.

Пример.

```
class BaseClass // родительский класс
{
   public Print() //метод родительского класса
   {
      Console.WriteLine(”BaseClass string”);
   }
}

class MyClass: BaseClass // дочерний класс
{
   public Print() //метод дочернего класса с сигнатурой, совпадающей
   {     //с сигнатурой метода, унаследованного от родительского класса
      Console.WriteLine(”MyClass string”);
   }
}
…
BaseClass bc = new BaseClass();
bc.Print();                      //BaseClass string
MyClass mc = new MyClass();
mc.Print();                      //MyClass string
BaseClass basec = mc; //ссылка типа родителя указывает на дочерний объект
basec.Print();                   //BaseClass string
                        //объект дочерний, но работает родительский метод
…
```

Как избежать зависимости версии метода от типа ссылки на объект мы рассмотрим далее.

Полиморфизм в C#.

Итак, в простом переопределении методов, рассмотренном пунктом выше, исполняемая версия метода зависит от типа ссылки на объект. Однако под ссылкой родительского типа можно хранить объект любого из его дочерних типов. Хотелось бы, чтобы объект работал именно как дочерний, а не как родительский. Следовательно, нужен механизм, который позволяет выполнять ту версию метода, которая определена для типа объекта независимо от типа ссылки на него. Такой подход позволяют осуществить *виртуальные* методы.

Виртуальные методы.

Метод базового класса, который предполагается к переопределению в потомках, объявляется виртуальным (помечается спецификатором *virtual*). При этом виртуальный метод обладает своей собственной реализацией. В дочернем классе виртуальный метод родительского класса переопределяется со спецификатором override. Следует также отметить, что если в дочернем классе новая реализация виртуального метода не нужна, то метод можно использовать с унаследованной от родителя реализацией.

Пример.

```
class BaseClass // родительский класс
{
   public virtual Print() //виртуальный метод родительского класса
   {
      Console.WriteLine(”BaseClass string”);
   }
}

class MyClass: BaseClass // дочерний класс
{
   public override Print() //переопределяющий метод дочернего класса
   {
      Console.WriteLine(”MyClass string”);
   }
}
…
BaseClass bc = new BaseClass();
bc.Print();                          //BaseClass string
MyClass mc = new MyClass();
mc.Print();                          //MyClass string
BaseClass basec = mc; //ссылка типа родителя указывает на дочерний объект
basec.Print();                       //MyClass string
                        //объект дочерний, работает дочерний метод
…
```

Таким образом, вызывается та версия метода, которая описана в конкретном классе, независимо от типа ссылки на объект.

Абстрактные методы.

Бывают ситуации, когда в методе базового класса реализация не нужна вовсе. Например, родительский класс предполагает расширение дочерними, и некоторые действия на этапе описания родительского класса еще не определены. Однако предполагается, что все дочерние классы должны иметь общую часть внешнего представления (метод с одной и той же сигнатурой), а выполнять ее каждый класс будет по-своему. Тогда в описание родительского класса включается метод без реализации – *абстрактный* метод. Введение такого метода гарантирует его наличие у всех потомков, тем самым достигается универсальность работы с потомками.

Метод без реализации в предке помечается спецификатором `abstract`, метод в потомке, наполняющий реализацией абстрактный метод предка, помечается спецификатором `override`.

Логично предположить, что в данной ситуации объект родительского типа создавать вообще нельзя, поскольку часть его реализации не определена. Значит, нужно запретить создавать объекты классов, не имеющих части реализации. Такие классы помечаются спецификатором `abstract` и называются *абстрактными* классами. То есть абстрактный класс – это класс, у которого отсутствует часть его реализации.

Пример.

```
abstract class BaseClass // родительский класс
{
   public abstract Print();//абстрактный метод родительского класса
}

class MyFirstClass: BaseClass // дочерний класс
{
   public override Print() //переопределяющий метод дочернего класса
   {
      Console.WriteLine(”MyFirstClass string”);
   }
}
class MySecondClass: BaseClass // дочерний класс
{
   public override Print() //переопределяющий метод дочернего класса
   {
      Console.WriteLine(”MySecondClass string”);
   }
}
…
BaseClass[] bc = new BaseClass[2];
bc[0] = new MyFirstClass();
bc[1] = new MySecondClass();
for (int i = 0; i < bc.Length; i++)
bc[i].Print();                    //MyFirstClass string
                                  //MySecondClass string
…
```

Класс, содержащий хотя бы один абстрактный метод, должен быть абстрактным. Обратное неверно! То есть если класс объявлен абстрактным,

то это не значит, что в нем обязательно есть хотя бы один абстрактный метод! Просто объекты таких классов создавать нельзя, и этот момент отслеживается на уровне компиляции. Зато, как видно из примера, под ссылкой абстрактного типа можно хранить объект любого из дочерних типов, которые и наполняют абстракцию реализацией. Причем каждый дочерний объект под одним внешним представлением работает по-своему.

Следует заметить, что класс-потомок не обязан в обязательном порядке переопределять абстрактный метод класса-предка. Если реализации абстрактного метода в классе-потомке нет, это значит, что он в своем составе содержит абстрактный метод, и тогда сам обязан быть маркирован как абстрактный. Более того, ключевое слово `abstract` не включается в сигнатуру метода. Это значит, что в потомке можно сделать абстрактным неабстрактный метод, унаследованный от предка. То есть полностью исключить реализацию метода, описанную в предке, и предложить уже своим потомкам написать собственное наполнение своего абстрактного метода, бывшего когда-то неабстрактным.

Бесплодные классы.

Абстрактные классы предназначены для порождения потомков. В противоположность им существуют классы, наследовать от которых запрещено. Чтобы запретить создавать наследников от класса его нужно пометить спецификатором `sealed`. Тогда любая попытка указать `sealed`-класс в качестве класса-предка будет вызывать ошибку компиляции.

Наследование – это возможное нарушение инкапсуляции, поскольку элемент родительского класса, даже приватный, доступен в потомке по ссылке `base`. Поэтому альтернативой наследованию является описание бесплодных классов и использование модели «Включение/делегирование».

Модель «Включение/делегирование»

Эта модель применяется для имитации наследования – например, в случае невозможности наследования от классов, запрещенных к наследованию (бесплодных). Класс, от которого хотелось бы создать потомка с расширенным внешним представлением, включается в качестве поля в описание расширяющего класса (этап включения). Затем описываются методы расширяющего класса, соответствующие по интерфейсу методам включенного класса. И вызовы методов включающего класса делегируют свои вызовы методам включенного класса (этап делегирования). То есть, по сути, тело метода расширяющего класса содержит вызов аналогичного метода расширяемого класса. При этом, в отличие от наследования, не происходит нарушения инкапсуляции. Кроме того, делегирующие методы могут содержать дополнительную функциональность, тем самым имитируя перегрузку методов включаемого класса. На модели «включение/делегирование» построены реализации многих паттернов проектирования.

Пример.

```
sealed class Included
{
   int count; //приватное поле
   public Included(int c) //конструктор с параметром
   {
     count = c;
   }
   public int Count //свойство для доступа к полю
   {
     get
     {
       return count;
     }
     set
     {
       count = value;
     }
   }
   public IncCount() //метод увеличения значения поля на 1
   {
     count++;
   }
   public DecCount() //метод уменьшения значения поля на 1
   {
     count--;
   }
}
class Including
{
   Included incl; //включение объекта
   string description; //приватное поле
   public Including(int c, string d) //конструктор с параметрами
   {
     incl = new Included(c);
     description = d;
   }
   public int Count //свойство для доступа к полю включенного объекта
   {
     get
     {
       return incl.Count;
     }
     set
     {
       incl.Count = value;
     }
   }
   public int Description //свойство для доступа к полю
   {
     get
     {
       return description;
     }
```

```
      set
      {
        description = value;
      }
    }
public IncCount() //метод увеличения значения поля на 1
    {
      incl.Count++; //делегирование вызова метода
    }
    public DecCount() //метод уменьшения значения поля на 1
    {
      incl.Count--; //делегирование вызова метода
    }
}
…
Including o = new Including(10, ”Количество”);
Console.WriteLine(o.Description + ” ” + o.Count);
o.IncCount();
Console.WriteLine(o.Description + ” ” + o.Count);
o.DecCount();
Console.WriteLine(o.Description + ” ” + o.Count);
…
```

В данном примере класс `Including` имитирует наследование от класса `Included`, копируя и расширяя его внешний интерфейс. При вызове методы `IncCount` и `DecCount` объекта класса `Including` вызывают методы `IncCount` и `DecCount` включённого объекта класса `Included`, имитируя их наследование переопределение. Однако обращение к состоянию включённого объекта возможно только через его внешнее представление, что никоим образом не нарушает принцип инкапсуляции. Следовательно, в некоторых случаях для защиты собственного кода предпочтительнее использование модели «Включение/делегирование», чем прямого наследования.

Интерфейсы.

Рассмотрим базовое понятие объектно-ориентированного программирования, с помощью которого реализуется полиморфическое использование объектов – понятие *интерфейса*.

Интерфейс – это полностью абстрактный класс, содержащий набор семантически связанных абстрактных элементов, предназначенных для реализации в наследующих классах. Каждый класс, наследующий от интерфейса (говорят, что класс *реализует* интерфейс) должен самостоятельно переопределить каждый их унаследованных элементов. Естественно, каждый класс реализует унаследованные элементы по-своему. Принято, что класс переопределяет ВСЕ элементы реализуемого интерфейса, хотя строго этого не требуется. Однако если класс реализует не все унаследованные от интерфейса элементы, это значит, что в классе остались абстрактные элементы, а такой класс обязан быть абстрактным, что не приветствуется.

В языке C# один класс может реализовывать несколько итерфейсов.

Синтаксис интерфейса:

```
<спецификатор доступа> interface <имя>
{
     <набор элементов>
}
```

Имя интерфейса составляется по нотации Паскаля следующим образом:

`I<сущностной корень>able`

Например, `IComparable`, `ICloneable`.

Элементами интерфейса могут быть методы (и их специальные виды – свойства и индексаторы), определяющие внешнее представление класса и объекта. Все элементы интерфейса по умолчанию имеют спецификаторы `public abstract`, поэтому их явное указание при описании интерфейса не нужно.

В интерфейсе, как правило, описывается небольшое количество элементов, объединенных единой семантикой. Суть интерфейса – заявить некую функциональность и обязать наследующий класс эту функциональность исполнять, то есть иметь в своем составе реализацию методов, заявленных в интерфейсе для дальнейшего полиморфического использования объектов реализующих классов. Например, если класс реализует интерфейс `IComparable`, то его экземпляры умеют сравниваться между собой.

Пример:

```
public interface ISummable //интерфейс
{
   int Sum(int a, int b); //абстрактный метод
}
public class Class1 : ISummable //класс, реализующий интерфейс
{
   int a;
   int b;
   public Class1(int a, int b)
   {
     this.a = a;
     this.b = b;
   }
   public int Sum() //реализация метода, унаследованного от интерфейса
   {
     return a + b;
   }
}
public class Class2 : ISummable //класс, реализующий интерфейс
{
   int a;
   int b;
   int c;
   public Class2(int a, int b, int c)
```

```
    {
      this.a = a;
      this.b = b;
      this.c = c;
    }
    public int Sum() //реализация метода, унаследованного от интерфейса
    {
      return a + b + c;
    }
}
…
Class1 i1 = new Class1(1,2);
Class2 i2 = new Class2(3,4,5);
ISummable i = i1; //ссылка типа интерфейс направлена на реальный объект
Console.WriteLine("Сумма равна " + i.Sum());
i = i2; //ссылка типа интерфейс направлена на другой реальный объект
Console.WriteLine("Сумма равна " + i.Sum());
…
```

Результатом работы данного примера будет следующий вывод:

```
Сумма равна 3
Сумма равна 12
```

Как видно из примера, результат работы метода зависит от реального типа объекта, вызывающего метод на исполнение, а не от типа ссылки на объект. То есть под однотипными интерфейсными ссылками могут храниться объекты различных реальных классов. И метод, заявленный в реализуемом интерфейсе, вызывается на исполнение одинаково (внешнее представление одинаково), а исполняется по-разному (внутренняя реализация в разных классах различна). Следует также отметить, что одинаковость внешнего представления объектов различных классов гарантируется только в той части, которая заявлена в том интерфейсе, который классы одновременно реализуют. Если к объектам таких классов обращаться по ссылке типа реализуемого интерфейса, то они будут иметь абсолютно одинаковое внешнее представление. Это и есть реализация полиморфизма – объекты на самом деле разные, но выглядят одинаковыми.

Используем описанную ранее иерархию классов в следующем коде:

```
ISummable[] mas = new ISummable[3];
mas[0] = new Class1(10, 20);
mas[1] = new Class2(1, 2, 3);
mas[2] = new Class2(4, 5, 6);
foreach (ISummable k in mas)
{
   Console.WriteLine("Сумма равна " + k.Sum());
}
```

На экран будет выведено следующее:

```
Сумма равна 30
Сумма равна 6
Сумма равна 15
```

Несмотря на то, что объекты `mas[0]` и `mas[1]` на самом деле принадлежат разным классам, если рассматривать их с точки зрения их интерфейса, то они абсолютно одинаковы, их можно хранить в массиве и обращаться к ним единым образом.

Получение ссылки типа интерфейс.

Получение интерфейсной ссылки на реальный объект возможно несколькими способами.

1. Присвоение ссылке интерфейсного типа ссылки на реальный объект (как в приведенном выше примере) – неявное приведение типа.

```
Class1 i1 = new Class1(1,2);
ISummable i = i1; //ссылка типа интерфейс направляется на реальный объект
```

Аналогичный результат будет получен и при вызове операции явного приведения типа.

```
Class1 i1 = new Class1(1,2);
ISummable i = (ISummable)i1;
```

Операции явного и неявного приведения типа могут привести к выбросу исключения `InvalidCastException` в том случае, если преобразование невозможно выполнить.

2. Использование операции приведения типа `as`.

```
ISummable i = i1 as ISummable;
```

Эта операция не приводит к выбросу исключений, если класс объекта `i1` не реализует интерфейс `ISummable`. В этом случае результат выполнения операции будет равен `null`. После выполнения операции `as` ее результат нужно проверить.

```
if (i == null) … //класс не реализует интерфейс
else …           //можно использовать ссылку i
```

3. Использование операции проверки принадлежности типу `is`. Этот способ является расширением способа 1 и позволяет избежать выброса ненужных исключений.

```
if (i1 is ISummable) // класс реализует интерфейс
ISummable i = i1;    //преобразование выполнится корректно
else …               //класс не реализует интерфейс
```

Ссылки типа интерфейс можно принимать в качестве параметров в методы и возвращать как результат выполнения метода.

Реализовывать один и тот же интерфейс могут классы, сами связанные наследованием. Если класс-предок реализовывал некий интерфейс, и класс-потомок реализует тот же интерфейс, то возможны два варианта реализаций:

1. Не переопределять методы интерфейса в классе-потомке. Тогда их реализация наследуется из класса-предка.

2. Реализовать в классе-потомке методы интерфейса. Тогда их реализации, унаследованные от класса-предка, игнорируются.

Второй вариант реализации является предпочтительным, поскольку в иерархии наследования явно указано, что класс-потомок сам реализует интерфейс.

Явная реализация интерфейса.

Поскольку в языке поддерживается множественное наследование от интерфейсов (один класс может реализовывать несколько интерфейсов), а элементы в каждом интерфейсе семантически связаны, то может возникнуть ситуация, когда в реализуемых классом интерфейсах есть методы, точно совпадающие по сигнатуре.

Пусть класс реализует два интерфейса, в которых есть метод, совпадающий по сигнатуре с методом из второго интерфейса. В этом случае может быть принято решение о единственной реализации классом обоих методов. В классе описывается одна реализация метода. Именно эта реализация и будет вызываться во всех случаях, независимо от типа ссылки на реальный объект – ссылки типа первый интерфейс, ссылки типа второй интерфейс или ссылки реального типа.

Но бывают ситуации, когда, несмотря на то, что сигнатура методов, принадлежащих разным интерфейсам, совпадает, их семантика может отличаться. То есть реализация методов должна быть различной. Тогда и применяется явная реализация интерфейсов. То есть при реализации методов в классе в заголовке метода явно указывается, какому интерфейсу будет принадлежать реализация метода.

Пример.

```
public interface ISummable //интерфейс
{
   int Sum(int a, int b); //абстрактный метод
}
public interface IDoubleSummable //интерфейс
{
   int Sum(int a, int b); //абстрактный метод
}
public class MyClass : ISummable, IDoubleSummable
{
   int a;
   int b;
   public MyClass(int a, int b)
   {
     this.a = a;
     this.b = b;
   }
   public int ISummable.Sum() //реализация метода из интерфейса ISummable
   {
     return a + b;
   }
   public int IDoubleSummable.Sum() //реализация метода из интерфейса
   {                                //IDoubleSummable
     return a*a + b*b;
   }
```

```
}
…
MyClass i = new MyClass(1,2);
ISummable isum = i;
Console.WriteLine("Сумма равна " + isum.Sum());
IDoubleSummable idoub = i;
Console.WriteLine("Сумма равна " + idoub.Sum());
…
```

Результат работы будет следующим:

```
Сумма равна 3
Сумма равна 5
```

Следует отметить, что при явной реализации интерфейса соответствующие версии методов будут вызываться только тогда, когда на объект указывает ссылка соответствующего интерфейсного типа. Во внешнем представлении самого класса метода не существует, несмотря на то, что в классе описано даже целых две версии метода, вызов любой из них у ссылки реального типа на объект невозможен.

То есть оператор

```
Console.WriteLine("Сумма равна " + i.Sum());
```

вызовет ошибку компиляции, потому что метода `Sum`, принадлежащего самому классу `MyClass`, не существует.

Наследование для интерфейсов.

Интерфейс – это класс, значит, он может быть включен в иерархию наследования, причем не только, как предок. Для интерфейсов в языке определено множественное наследование. Интерфейс не может наследовать от класса, но может наследовать от нескольких интерфейсов. В интерфейсе-потомке содержатся все методы, унаследованные от интерфейсов-предков. Также как и в классе, новый метод взамен унаследованного нужно описывать с помощью ключевого слова `new`. Чтобы обратиться к скрытому теперь методу интерфейса-потомка необходимо явное преобразование к типу интерфейса-потомка. Кроме того, наследующий интерфейс не может расширять область видимости базового интерфейса.

При наследовании интерфейсов следует помнить, что наследуются только заголовки методов, а не их реализации, то есть наследуется внешний интерфейс. Так что особого смысла в наследовании интерфейсов от интерфейсов ВООБЩЕ НЕТ.

Стандартные интерфейсы.

В языке представлено несколько стандартных интерфейсов. Их реализация позволяет пользовательским классам участвовать в стандартных механизмах и схемах наравне со стандартными классами.

Интерфейс IComparable

Описан в пространстве имен `System.Collections`. Реализуется тогда, когда объекты класса нужно уметь сравнивать между собой на

«больше» и «меньше» (на равенство и неравенство объекты сравнимы при переопределении метода `Equals` из класса `Object`).

Интерфейс содержит один метод:

```
int CompareTo(object o);
```

Метод должен возвращать 1 в том случае, если объект, у которого вызывается метод, «больше» чем объект, переданный в качестве параметра; -1 в том случае, если объект, у которого вызывается метод, «меньше» чем объект, переданный в качестве параметра; 0 в том случае, если объекты «равны». Следует отметить, что отношения «больше», «меньше» и «равны» следует определить, исходя из семантики задачи.

Пример.

```
class MyClass : IComparable
{
   int count;
   string description;
   public MyClass(int c, string d)
   {
     count = c;
     description = d;
   }
   public int CompareTo(object o) //метод интерфейса IComparable
   {
     MyClass mo = o as MyClass;
     if (this.count > mo.count) return 1;
     else
       if (this.count < mo.count) return -1;
       else
         return 0;
   }
}
…
MyClass m1 = new MyClass(1, "Первый объект");
MyClass m2 = new MyClass(2, "Второй объект");
if (m1.CompareTo(m2) > 0)
   Console.WriteLine("Первый объект больше второго");
   else
     if (m1.CompareTo(m2) < 0)
       Console.WriteLine("Первый объект меньше второго");
     else Console.WriteLine("Объекты равны");
…
```

В данном примере объект класса `MyClass` содержит два поля – целого и строкового типа. В методе, унаследованном от интерфейса `IComparable`, определим критерий сравнения объектов на «больше» и «меньше» При сравнении объектов большим будет считаться тот объект, у которого значение целого поля `count` больше.

Объекты класса MyClass теперь сравнимы между собой. Реализация интерфейса IComparable позволяет использовать объекты реализующего класса в стандартных технологиях, основанных на сравнении объектов:

```
MyClass[] mas = new MyClass[5];
mas[0] = new MyClass(5, "5");
mas[1] = new MyClass(1, "1");
mas[2] = new MyClass(7, "7");
mas[3] = new MyClass(3, "3");
mas[4] = new MyClass(2, "2");
Array.Sort(mas);
```

Сортировка по возрастанию массива объектов типа MyClass стала возможна стандартным методом сортировки, поскольку теперь объекты сравнимы друг с другом (реализуют интерфейс IComparable). Вид сортировки зависит от единственного в классе переопределения метода CompareTo.

Интерфейс IComparer.

Интерфейс предназначен для задания критерия сортировки, позволяет описать различные правила сравнения объектов.

Интерфейс принадлежит пространству имен System.Collections. В нем задан один метод:

```
int Compare (object o1, object j2);
```

Метод должен возвращать 1 в том случае, если первый параметр «больше» второго; -1 в том случае, если первый параметр «меньше» второго; 0 в том случае, если объекты «равны». Так же, как и в методе CompareTo, отношения «больше», «меньше» и «равны» следует определить, исходя из семантики задачи.

Порядок применения критерия сортировки:

1. Описывается вспомогательный класс, реализующий интерфейс IComparer – класс, объект которого будет являться критерием сравнения объектов пользовательского класса. Таких вспомогательных классов может быть несколько для реализации различных критериев сортировки.

2. Объект критерия передается в качестве параметра в стандартный метод сортировки массива пользовательских объектов. Для различных видов сортировки необходим свой объект критерия.

Пример.

```
class MyClass //пользовательский класс
{
   int count;
   string description;
   public MyClass(int c, string d)
   {
     count = c;
     description = d;
   }
   public int Count
```

```
    {
      get
      {
        return count;
      }
      set
      {
        count = value;
      }
    }
    public string Description
    {
      get
      {
        return description;
      }
      set
      {
        description = value;
      }
    }
}
class SortByCountUp : IComparer //класс критерия сортировки
{                                //по возрастанию поля count
    int IComparer.Compare(object o1, object o2)
    {
      MyClass mc1 = o1 as MyClass;
      MyClass mc2 = o2 as MyClass;
      if (mc1.Count > mc2.Count) return 1;
      else
        if (mc1.Count < mc2.Count) return -1;
        else return 0;
    }
}
class SortByCountDown : IComparer //класс критерия сортировки
{                                  //по убыванию поля count
    int IComparer.Compare(object o1, object o2)
    {
      MyClass mc1 = o1 as MyClass;
      MyClass mc2 = o2 as MyClass;
      if (mc1.Count < mc2.Count) return 1;
      else
        if (mc1.Count > mc2.Count) return -1;
        else return 0;
    }
}
class SortByDescriptionUp : IComparer //класс критерия сортировки
{                                    //по возрастанию поля description
    int IComparer.Compare(object o1, object o2)
    {
      MyClass mc1 = o1 as MyClass;
      MyClass mc2 = o2 as MyClass;
      return mc1.Description.CompareTo(mc2.Description);
```

```
    }
}
class SortByDescriptionDown : IComparer //класс критерия сортировки
{                                       //по убыванию поля description
    int IComparer.Compare(object o1, object o2)
    {
      MyClass mc1 = o1 as MyClass;
      MyClass mc2 = o2 as MyClass;
      return mc2.Description.CompareTo(mc1.Description);
    }
}
…
MyClass[] mas = new MyClass[5];
mas[0] = new MyClass(5, "5");
mas[1] = new MyClass(1, "1");
mas[2] = new MyClass(7, "7");
mas[3] = new MyClass(3, "3");
mas[4] = new MyClass(2, "2");
Array.Sort(mas, new SortByCountUp());
Array.Sort(mas, new SortByCountDown());
Array.Sort(mas, new SortByDescriptionUp());
Array.Sort(mas, new SortByDescriptionDown());
…
```

В последних четырех строках кода в примере массив объектов сортируется по различным критериям. Но массив всегда сортируется по возрастанию. А вот смысл слова «возрастание» и задается в критерии сортировки.

Перегрузка операций отношения

Операции класса были рассмотрены в теме 4. Нерассмотренным остался вопрос о перегрузке операций сравнения для объектов пользовательских классов. Перегрузка таких операций должна учитывать правила сравнения объектов на «больше» и «меньше» (метод `CompareTo` из интерфейса `IComparable`), а также признак эквивалентности объектов, устанавливаемый методом `Equals`, унаследованным из класса `Object`, который был рассмотрен в теме 5. Поэтому перегрузка операций отношения обычно использует перечисленные заранее устанавливаемые критерии и правила.

Операции сравнения – «>», «<», «>=», «<=», «==», «!=» - относятся к бинарным операциям, и возвращают результат типа `bool`.

Пример.

```
class MyClass : IComparable
{
   int count;
   string description;
   public MyClass(int c, string d)
   {
     count = c;
     description = d;
```

```
    }
    public int CompareTo(object o)
    {
      MyClass mo = o as MyClass;
      if (this.count > mo.count) return 1;
      else
        if (this.count < mo.count) return -1;
      else return 0;
    }
    public override bool Equals(object o)
    {
      MyClass mc = (MyClass)o;
      if   (this.count   ==   mc.Count   &&   this.description   ==
      mc.Description) return true;
      else return false;
    }
    public override int GetHashCode() //метод переопределяется с Equals
    {
      return count.GetHashCode();
    }
    // переопределение операций отношения
    public static bool operator ==(MyClass mc1, MyClass mc2)
    {
      return mc1.Equals(mc2);
    }
    public static bool operator !=(MyClass mc1, MyClass mc2)
    {
      return !mc1.Equals(mc2);
    }
    public static bool operator >(MyClass mc1, MyClass mc2)
    {
      if (mc1.CompareTo(mc2) > 0) return true;
      else return false;
    }
    public static bool operator <(MyClass mc1, MyClass mc2)
    {
      if (mc1.CompareTo(mc2) < 0) return true;
      else return false;
    }
    public static bool operator >=(MyClass mc1, MyClass mc2)
    {
      if (mc1.CompareTo(mc2) > 0 || mc1.Equals(mc2)) return true;
      else return false;
    }
    public static bool operator <=(MyClass mc1, MyClass mc2)
    {
      if (mc1.CompareTo(mc2) < 0 || mc1.Equals(mc2)) return true;
      else return false;
    }
}
…
MyClass mc1 = new MyClass(5, "5");
MyClass mc2 = new MyClass(2, "2");
```

```
if (mc1 == mc2)
   Console.WriteLine("Объекты равны");
if (mc1 != mc2)
   Console.WriteLine("Объекты не равны");
if (mc1 > mc2)
   Console.WriteLine("Первый объект больше второго");
if (mc1 < mc2)
   Console.WriteLine("Первый объект меньше второго");
if (mc1 >= mc2)
   Console.WriteLine("Первый объект больше второго или равен ему");
if (mc1 <= mc2)
   Console.WriteLine("Первый объект меньше второго или равен ему");
…
```

Результат выполнения данного кода выглядит следующим образом:

```
Объекты не равны
Первый объект больше второго
Первый объект больше второго или равен ему
```

Следует отметить, что теперь операции сравнения для объектов класса `MyClass` переопределены. Результат выполнения операций сравнения на равенство или сравнения на неравенство может отличаться от результата, принятого по умолчанию для операций сравнения объектов ссылочных типов.

Клонирование объектов

Поскольку операция присваивания для ссылочных типов данных не приводит к созданию новой копии объекта, то эту возможность можно реализовать специальным образом.

Клонирование – это создание копии объекта. Различают два вида клонирования объектов – поверхностное и глубокое.

При поверхностном клонировании объекта создается его точная копия. При этом все значащие поля копируются в клон, а вот поля, являющиеся ссылками на другие объекты, также просто копируются в клон – то есть остаются направленными на старые объекты, новых объектов для полей-ссылок поверхностное клонирование не создает. Такой тип полезен в том случае, если есть уверенность в отсутствии у клонируемого объекта полей, ссылающихся на другие объекты. Поверхностное клонирование реализовано в методе `MemberwiseClone()` в классе `Object`.

Для создания полного независимого – глубокого – клона объекта нужно реализовывать так называемое глубокое клонирование. Алгоритм глубокого клонирования не специфицирован и описывается разработчиком. При этом класс должен реализовывать интерфейс `ICloneable`, в котором описан единственный метод

```
object Clone();
```

Именно в этом методе и реализуется алгоритм глубокого клонирования. Реализация классом интерфейса `ICloneable` говорит о том, что объекты класса умеют создавать свои глубокие клоны. Следует отметить, что в заголовке метода `Clone` в качестве возвращаемого результата указан

тип `object`. Это означает, что при получении глубокого клона необходимо его явное преобразование к искомому типу.

Пример.

```
class MyClass : ICloneable
{
   int[] mas;
   int length;
   public MyClass(int l)
   {
     mas = new int[l];
     length = mas.Length;
   }
   public int this[int i]
   {
     get
     {
       return mas[i];
     }
     set
     {
       mas[i] = value;
     }
   }
   public int Length
   {
     get
     {
       return length;
     }
   }
   public object LightClone()
   {
     return this.MemberwiseClone();
   }
   public object Clone()
   {
     return new MyClass1(this.length);
   }
}
…
MyClass o = new MyClass(3); //создание объекта
for (int i = 0; i < o.Length; i++) //задание состояния объекта
{
   o[i] = i;
}
Console.WriteLine("Клонируемый объект:");
for (int i = 0; i < o.Length; i++) //вывод на экран состояния объекта
{
   Console.WriteLine(o[i]);
}
MyClass o1= (MyClass)o.LightClone();//создание поверхностного клона
o1[0] = 10;                          //изменение состояния клона
```

```
Console.WriteLine("Клонируемый объект:");
for (int i = 0; i < o.Length; i++)
{
   Console.WriteLine(o[i]); //вывод состояния исходного объекта
}
…
```

Вывод на экран будет следующим:

```
Клонируемый объект:
0
1
2
Клонируемый объект:
10
1
2
```

Как видно из примера, клонируемый объект и его поверхностный клон ссылаются на один и тот же массив целых чисел. Поэтому изменение состояния одной копии влечет за собой автоматическое изменение состояния второй копии.

Заменим в примере создание поверхностного клона созданием глубокого клона:

```
MyClass o = new MyClass(3); //создание объекта
for (int i = 0; i < o.Length; i++) //задание состояния объекта
{
   o[i] = i;
}
Console.WriteLine("Клонируемый объект:");
for (int i = 0; i < o.Length; i++) //вывод на экран состояния объекта
{
   Console.WriteLine(o[i]);
}
MyClass o2= (MyClass)o.Clone(); //создание глубокого клона
o2[0] = 10;                          //изменение состояния клона
Console.WriteLine("Клонируемый объект:");
for (int i = 0; i < o.Length; i++)
{
   Console.WriteLine(o[i]); //вывод состояния исходного объекта
}
…
```

Результат работы этого примера будет следующим:

```
Клонируемый объект:
0
1
2
Клонируемый объект:
0
1
2
```

Как видно из результата работы, глубокий клон и клонируемый объект совершенно не зависят друг от друга.

Тема 8. Особые классы.

Структуры.

Структура – это набор разнотипных данных, имеющий одно имя и позволяющий доступ через это имя к составным частям структуры. По сути структура очень похожа на класс.

Структура и представляет собой особый класс, имеющий ряд отличительных характеристик:

- структура является значимым типом, то есть хранится в стэке и обрабатывается при присваивании и копировании по правилам для значащих типов;
- структуры не участвуют в наследовании, то есть не могут порождать потомков, и сами являются прямыми наследниками класса `Object`;
- структуры могут реализовывать интерфейсы;
- структуры не могут иметь спецификатор `abstract`;
- в структуре могут быть описаны все элементы, присущие классам, только в структуре не может быть деструктора и конструктора без параметров;
- спецификатор доступа `protected` для структур не используется, т.к. у структуры не может быть потомков, в основном структуры снабжаются спецификатором `public`, а спецификаторы `internal` и `private` используются для вложенных структур;
- для элементов структур также не используются спецификаторы `protected` и `protected internal`.;
- для полей структуры нельзя задавать значения по умолчанию (а вот статическим полям значения по умолчанию задать можно);
- методы структур не могут быть помечены спецификаторами `abstract` или `virtual`, однако структуры, являясь потомком класса `Object`, могут переопределять его виртуальные методы с использованием спецификатора `override`.

Синтаксис структуры:

```
<атрибуты> <спецификаторы> struct <Имя> : <Интерфейсы>
{
   <тело структуры>;
}
```

Пример. Структура описывает студента. Для сокращения кода структуры объявим ее поля публичными (более правильно сделать поля приватными и описать для доступа к ним публичные свойства).

```
struct Student
{
   public id;//номер зачетной книжки
   public surname;//фамилия
```

```
   public Student(int num, string fam)//конструктор с параметрами
   {
      id = num;
      surname = fam;
   }
   public override string ToString()
   {
      return ”Номер зачетки: ”+ id + ”, фамилия: ” + surname;
   }
}
…
Student Vasya = new Student(1, ”Иванов”);
Student Kolya = new Student(2, ”Петров”);
Student Petya=Vasya;//копируется не ссылка на объект, а его значение
Petya.id = 3;
Console.WriteLine(”Студенты:\n” + Vasya + ”\n” + Kolya + ”\n” +
Petya);
…
```

На экран будет выведен следующий результат:

```
Студенты:
Номер зачетки: 1, фамилия: Иванов
Номер зачетки: 2, фамилия: Петров
Номер зачетки: 3, фамилия: Иванов
```

При присваивании структуры создается ее копия в стеке, поэтому ссылки `Petya` и `Vasya` ссылаются на РАЗНЫЕ объекты (изменения, вносимые в один объект, не отражаются на другом, что имело бы место быть при аналогичной работе с классами). При передаче в методы структуры также передаются по значению, их также можно передать в метод с помощью ключевых слов `ref` и `out`.

Работа со структурами в стэке производится быстрее, чем работа с объектами в динамической памяти.

Перечисления.

Перечисление – набор связанных между собой именованных констант, при этом конкретные значения этих констант не особо важны, важно само имя константы и ее принадлежность перечислению.

Конечно, за перечислением спрятан целочисленный тип, поэтому конкретное значение каждой константы можно задать из диапазона значений того целочисленного типа, на котором базируется описываемое перечисление.

Синтаксис перечисления:

```
<атрибуты> <спецификаторы> enum <Имя> : <базовый тип>
{
   <константы перечисления>
}
```

Атрибуты и спецификаторы перечисления аналогичны атрибутам и спецификаторам класса.

Переменная типа перечисление в качестве значения может принимать одну из констант перечисления.

Базовым типом перечисления может быть один из целочисленных типов (кроме `char`). Если базовый тип не указывается в описании перечисления, то по умолчанию базовым типом считается тип `int`. Каждая константа в перечислении имеет свое базовое целочисленное значение. Целочисленные значения константам в перечислении задаются упорядоченно, начиная с 0. Эти значения можно задать и явным образом

Тело перечисления – перечисление его констант с возможным явным указанием их целочисленных значений, если значение константы не указано явно, то оно по умолчанию равно увеличенному на 1 значению предыдущей константы. В перечислении не может быть констант с одинаковыми целочисленными значениями. Также по умолчанию все константы в перечислении публичны.

Пример.

```
enum Color //перечисление возможных цветов
{Red, Green, Blue, Black, White, Yellow}
class Figure
{
   public double area; //площадь фигуры
   public Color color; //цвет фигуры
…
}
…
Figure circle = new Figure();
circle.area = 2.54;
circle.color = Color.Red; //для фигуры задается красный цвет
…
```

С переменными типа перечисление возможно выполнение следующих арифметических операций: «+», «-», «++», «--», «>», «<», «>=», «<=», «==», «!=», «&», «|», «~», «^» и операция получения размера в байтах `sizeof`. Конечно, операция выполняется над целочисленным значением константы перечисления.

Переменные перечислимых типов можно использовать в арифметических выражениях и операциях присваивания, только требуется явное преобразование к целому типу. В общем случае, переменной перечислимого типа можно присвоить любое значение базового целочисленного типа, в том числе и не входящего в значения констант перечисления, но в этом нет особого смысла, поскольку у такого значения нет имени в перечислении.

Все перечисления являются наследниками своего родительского типа `Enum` из пространства имен `System`. Состав класса `Enum`:

Элемент	Описание	Вид
GetName	Получение имени константы по	Статический метод

	ее значению	
GetNames	Получение массива имен констант в перечислении	Статический метод
GetValues	Получение массива значений констант в перечислении	Статический метод
IsDefined	Возвращает true, если строкового имя константы определена в перечислении	Статический метод
GetUnderlyingType	Возвращает имя базового типа (по умолчанию `Int32`)	Статический метод

```
Console.WriteLine(Enum.GetName(typeof(Color), 3)); //black
if (! Enum.IsDefined(typeof(Color), ”Brown”))
Console.WriteLine(”Константы Brown нет в перечислении Color”);
```

Все статические методы принимают в качестве параметра представление типа перечисления (ссылку типа `Type` на описание перечисления, которую можно получить с помощью операции `typeof`).

Тема 9. Коллекции.

Коллекции – это массивы объектов с расширенной функциональностью. Для универсальности коллекции хранят объекты типа `object`. Коллекции описаны в пространстве имен `System.Collections`.

Коллекции бывают разной структуры и назначения. Функциональность коллекций разделена по стандартным интерфейсам. Применимость коллекции для решения конкретной задачи хранения данных и организации работы с ними определяется набором тех интерфейсов, которые коллекция реализует.

В пространстве имен `System.Collections` описаны следующие стандартные интерфейсы (о некоторых из них уже шла речь в теме 7):

Интерфейс	Описание
ICollection	Определяет общие характеристики коллекций – счетчики, связи, реализует интерфейс IEnumerable
IComparer	Определяет критерий сравнения двух объектов
IDictionary	Позволяет объекту представить его содержимое в виде пар «имя/значение», реализует интерфейс IEnumerable
IDictionaryEnumerator	Позволяет перечислять объекты в коллекции, реализующей интерфейс IDictionary
IEnumerable	Возвращает объект типа IEnumerator для коллекции
IEnumerator	Позволяет перечислять объекты с помощью цикла foreach
IList	Предоставляет возможность добавления, удаления, индексирования элементов коллекции, определяет размер коллекции и доступ на чтение/запись элементов коллекции, реализует интерфейс IEnumerable

Рассмотрим классы стандартных коллекций.

Интерфейс	Описание	Реализуемые интерфейсы
ArrayList	Динамически изменяемый массив объектов	IEnumerable, ICollection, ICloneable, IList
HashTable	Набор пар «ключ/значение», ключом является хэш-код объекта. Типы объектов коллекции должны переопределять метод GetHashCode	IEnumerable, ICollection, ICloneable, IDictionary

Queue	Очередь типа FIFO (first in – first out)	IEnumerable, ICollection, ICloneable
SortedList	Позволяет обратиться к коллекции пар по номеру (индексу)	IEnumerable, ICollection, ICloneable, IDictionary
Stack	Очередь типа LIFO (last in – fisrt out)	IEnumerable, ICollection, ICloneable

Коллекция ArrayList.

Коллекция представляет собой массив с изменяющимся во времени количеством элементов. Хранит элементы любого типа под ссылками типа `object`. По умолчанию создается из 16 элементов, количество элементов можно передать в конструкторе.

Состав класса:

Элемент	**Описание**	**Вид**
Capacity	Возможное количество элементов коллекции (емкость коллекции)	Свойство для чтения и записи
Count	Фактическое количество элементов в коллекции	Свойство для чтения
Item	Чтение и запись элемента по индексу	Свойство
Add	Добавление элемента в конец коллекции	Метод
AddRange	Добавление другой коллекции в конец коллекции	Метод
BinarySearch	Двоичный поиск в сортированной коллекции	Метод
Clear	Удаление всех элементов коллекции	Метод
Clone	Поверхностное клонирование коллекции	Метод
CopyTo	Копирование целиком или части коллекции в другую коллекцию	Метод
GetRange	Получение части коллекции	Метод
IndexOf	Поиск первого вхождения элемента в коллекцию. Если искомого элемента в коллекции нет, то метод возвращает -1	Метод
Insert	Вставка элемента в указанную позицию коллекции (по индексу)	Метод
InsertRange	Вставка другой коллекции по указанному индексу в коллекцию	Метод
LastIndexOf	Поиск последнего вхождения элемента в коллекцию. Если искомого элемента в	Метод

	коллекции нет, то метод возвращает -1	
Remove	Удаление первого вхождения элемента из коллекции	Метод
RemoveAt	Удаление элемента из коллекции по индексу	Метод
RemoveRange	Удаление группы элементов из коллекции	Метод
Reverse	Реверс коллекции	Метод
SetRange	Установка значений для группы элементов в коллекции	Метод
Sort	Сортировка коллекции по возрастанию	Метод
TrimToSize	Уменьшение емкости коллекции до фактического количества элементов	Метод

Методы, увеличивающие количество элементов в коллекции, при превышении текущей емкости приводят к ее увеличению в 2 раза. А методы уменьшения количества элементов не уменьшают емкость коллекции. Возможна ситуация, когда емкость многократно превышает фактическое число элементов. Тогда применяется метод `TrimToSize`().

Коллекция `ArrayList` реализована через закрытый массив типа `Array` элементов типа `Object`. Если в коллекции хранятся элементы значащих типов, то для них автоматически создается объектная оболочка, помещающаяся в коллекцию (boxing – упаковка). При извлечении из коллекции необходима их распаковка – unboxing. Многократно повторяемая упаковка-распаковка снижает быстродействие. Поэтому коллекция в основном используется для обработки ссылочных объектов. Но поскольку объекты хранятся по ссылками типа `Object`, то при извлечении их из коллекции необходимо явное преобразование.

Пример.

```
…
ArrayList ar = new ArrayList();
ar.Add(25);
ar.Add(3.14);
ar.Add(”Строка”);
ar.Add(new int[3]{1, 2. 3});
int a = (int)ar[0];
int[] mas = (int[])ar[3];
ar.Remove(”Строка”);
ar.RemoveAt(1);
…
```

Тема 10. Объекты файловой системы.

К объектам файловой системы относятся каталоги (папки, директории) и файлы. В языке объекту файловой системы ставится в соответствие объект определенного класса. Через этот объект можно управлять объектом файловой системы.

Для работы с каталогами предназначен класс `DirectoryInfo`, для работы с файлами – класс `FileInfo`. Оба класса являются наследниками абстрактно класса `FileSystemInfo`, каждый из них по-своему реализует унаследованные абстрактные методы.

Свойства класса FileSystemInfo:

`Attributes`	Читает или устанавливает атрибуты, связанные с текущим файлом, представленным в перечне `FileAttributes`
`CreationTime`	Читает или устанавливает время создания для текущего файла или каталога
`Exists`	Может использоваться для выяснения того, существует ли данный файл или каталог
`Extension`	Читает расширение файла
`FullName`	Получает полный путь каталога или файла
`LastAccessTime`	Читает или устанавливает время последнего доступа к текущему файлу или каталогу
`LastWriteTime`	Читает или устанавливает время последнего сеанса записи в текущий файл или каталог
`Name`	Для файлов получает имя файла. Для каталогов получает имя последнего каталога в иерархии, если такая иерархия существует. Иначе получает имя каталога

Класс DirectoryInfo наследует все свойства класса FileSystemInfo и имеет собственные элементы:

Члены	Описание
`Create()` `CreateSubdirectory()`	Создает каталог (или множество подкаталогов) в соответствии с заданным именем пути
`Delete()`	Удаляет каталог и все его содержимое
`GetDirectories()`	Возвращает массив строк, представляющих все подкаталоги текущего каталога
`GetFiles()`	Получает массив типов `FileInfo`, представляющих множество файлов данного каталога
`MoveTo()`	Перемещает каталог и его содержимое в место, соответствующее заданному новому пути
`Parent`	Получает каталог родителя указанного пути
`Root`	Получает корневую часть пути

Атрибуты объекта файловой системы описаны в перечислении FileAttributes:

Значение	Описание
Archive	Используется приложениями при выполнении резервного копирования, а в некоторых случаях — при удалении старых файлов
Compressed	Файл является сжатым
Directory	Объект файловой системы является каталогом
Encrypted	Файл является зашифрованным
Hidden	Файл является скрытым
Normal	Файл находится в обычном состоянии, и для него установлены любые другие атрибуты. Этот атрибут не может использоваться с другими атрибутами
Offline	Файл, расположенный на сервере, кэширован в хранилище на клиентском компьютере. Возможно, что данные этого файла уже устарели
ReadOnly	Файл доступен только для чтения
System	Файл является системным

Класс FileInfo/

Объект класса соответствует файлу на дисковом накопителе. Состав класса:

Член	Описание
AppendText()	Создает тип StreamWriter (будет описан позже) для добавления текста в файл
CopyTo()	Копирует существующий файл в новый файл
Create()	Создает новый файл и возвращает тип FileStream (будет описан позже) для взаимодействия с созданным файлом
CreateText()	Создает тип StreamWriter, который записывает новый текстовый файл
Delete()	Удаляет файл, к которому привязан экземпляр FileInfo
Directory	Получает экземпляр каталога родителя
DirectoryName	Получает полный путь к каталогу родителя
Length	Получает размер текущего файла или каталога
MoveTo()	Перемещает указанный файл в новое место, имеет опцию для указания нового имени файла
Name	Получает имя файла
Open()	Открывает файл с заданными возможностями чтения/записи и совместного доступа
OpenRead()	Создает FileStream с доступом только для чтения
OpenText()	Создает тип StreamReader (будет описан позже) для чтения из существующего текстового файла
OpenWrite()	Создает FileStream с доступом только для записи

Перечисление FileMode

Описывает режимы открытия файла.

Значение	Описание
Append	Открыть файл, если он существует, и установить текущий указатель в конец файла. Если файл не существует, создать новый файл
Create	Создать новый файл. Если в каталоге уже существует файл с таким же именем, он будет стерт
CreateNew	Создать новый файл. Если в каталоге уже существует файл с таким же именем, возникает исключение IOException
Open	Открыть существующий файл
OpenOrCreate	Открыть файл, если он существует. Если нет, создать файл с таким именем
Truncate	Открыть существующий файл. После открытия он должен быть обрезан до нулевой длины

Перечисление FileAccess

Описывает режимы доступа к файлу.

Значение	Описание
Read	Открыть файл только для чтения
ReadWrite	Открыть файл для чтения и записи
Write	Открыть файл только для записи

Перечисление FileShare

Описывает режимы совместного доступа к файлу.

Значение	Описание
None	Совместное использование открытого файла запрещено. Запрос на открытие данного файла завершается сообщением об ошибке
Read	Позволяет открывать файл для чтения одновременно нескольким пользователям. Если этот флаг не установлен, запросы на открытие файла для чтения завершаются сообщением об ошибке
ReadWrite	Позволяет открывать файл для чтения и записи одновременно нескольким пользователям
Write	Позволяет открывать файл для записи одновременно нескольким пользователям

Классы Directory и File

Данные классы в основном дублируют возможности классов `DirectoryInfo` и `FileInfo`, но поддерживают обработку объектов файловой системы через статические методы.

Тема 11. Схема потоков данных.

Все классы для работы с потоками данных содержатся в пространстве имен System.IO.

В объектно-ориентированных языках, в том числе и в C#, передача данных (ввод/вывод) осуществляется по определенной схеме – схеме потоков. Данные представляются в виде абстрактного линейного потока, передаваемого от источника к получателю. При этом источник не знает, кто является получателем данных, и работает только с потоком. Получатель также не имеет информации об источнике, и работает только с потоком данных. Такая абстракция позволяет сделать универсальной передачу данных независимо от типа получателя, и универсальным прием данных, независимо от источника. При этом и источник и получатель работают с абстракцией потока.

Потоки делятся на потоки ввода (потоки, с которыми работает получатель) и потоки вывода (потоки, с которыми работает источник). Таким образом, ввод и вывод логически отделены друг от друга и по большому счету никак не связаны. По типу передаваемых данных потоки бывают байтовыми и текстовыми.

То есть существует 4 больших группы потоков – байтовые ввода, байтовые вывода, текстовые ввода (их называют потоками чтения) и текстовые вывода (их называют потоками записи).

Каждая из этих групп имеет базовый абстрактный класс, в котором описана основная функциональность соответствующей группы потоков, наследники этих абстрактных классов реализуют ввод или вывод для конкретных хранилищ данных.

Следует отметить, что после окончания работы поток нужно обязательно закрыть.

Байтовые потоки данных.

В языке C# байтовый поток ввода и байтовый поток вывода объединены в единую сущность. То есть речь ведется не с потоком ввода и с потоком вывода по отдельности, а с байтовым потоком ввода/вывода. В связи с этим возникает необходимость управления курсором в потоке.

Базовая функциональность байтовых потоков ввода/вывода описана в абстрактном классе Stream.

Состав класса Stream.

Элемент	Описание
BeginRead, BeginWrite	Начать асинхронный ввод или вывод
CanRead, CanSeek, CanWrite	Свойства, определяющие, какие операции поддерживает поток: чтение, прямой доступ и/или запись
Close	Закрыть текущий поток и освободить связанные с ним ресурсы (сокеты, указатели на файлы и т. п.)

EndRead, EndWrite	Ожидать завершения асинхронного ввода; закончить асинхронный вывод
Flush	Записать данные из буфера в связанный с потоком источник данных и очистить буфер. Если для данного потока буфер не используется, то этот метод ничего не делает
Length	Возвратить длину потока в байтах
Position	Возвратить текущую позицию в потоке
Read, ReadByte	Считать последовательность байтов (или один байт) из текущего потока и переместить указатель в потоке на количество считанных байтов
Seek	Установить текущий указатель потока на заданную позицию
SetLength	Установить длину текущего потока
Write, WriteByte	Записать последовательность байтов (или один байт) в текущий поток и переместить указатель в потоке на количество записанных байтов

Класс FileStream

Класс является наследником класса Stream и реализует всю его функциональность для взаимодействия с файлами. Файл при этом интерпретируется как линейный поток байтов, каждый байт которого может быть прочитан или записан. При этом реализуется возможность управления курсором в файле. Для задания режимов работы с файлами используются стандартные перечисления `FileMode`, `FileAccess` и `FileShare`, рассмотренные в теме 10.

Класс FileStream также позволяет создать новый файл на диске, или открыть существующий (то есть в этом смысле дублирует функциональность классов FileInfo и File).

```
FileStream fs = new FileStream(”test.dat”,
FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.None);
byte[] x = new byte[10];
for (int i = 0; i < 10, i++)
{
   x[i] = i;
}
fs.Write(x); //в поток побайтно записывается массив байт
fs.Write(x,0,5); //записывается 5 элементов массива, начиная с нулевого
for (byte i = 0; i < 256; i++)
{
   fs.WriteByte(i); //записываются числа от 0 до 255 в виде байтов
}
fs.Position = 0; //перемещаем курсор в начало файлового потока
byte[] y = new byte[15];
fs.Read(y,0,15); //из потока считывается 15 элементов в массив
byte z = fs.ReadByte();
fs.Close();
```

Метод `Seek()` используется для установки курсора в потоке на байт с определенным номером (еще один способ управления свойством `Position`). Этот метод имеет два параметра – количество байт, которые

нужно отступить и точку отсчета, от которой нужно отступить указанное число байт. Точками отсчета являются константы из перечисления `SeekOrigin`:

- `Begin` – от начала файлового потока:
- `Current` – от текущей позиции курсора в потоке;
- `End` – от конца потока.

При работе с файлами возможен выброс следующих исключений:

`FileNotFoundException` – попытка открыт несуществующий файл;

`DirectoryNotFoundException` – обращение к несуществующей директории;

`ArgumetException` – неверно задан режим открытия потока;

`IOException` – ошибка ввода/вывода.

Эти исключения нужно отлавливать.

Класс MemoryStream.

Класс является наследником класса `Stream` и реализует работу с потоком, организованным в динамический памяти. Используется в основном как временное хранилище двоичных данных, которые должны быть в итоге записаны в поток, связанный с постоянным хранилищем (например, в файл). Работа с таким потоком очень похожа на работу с файловым потоком (у них один предок). Дополнительные элементы класса `MemoryStream`:

Элемент	**Описание**	**Вид**
Capacity	Количество байт, выделенных под поток (объем потока)	Свойство
GetBuffer	Возвращает массив байт, на котором был организован поток	Метод
ToArray	Записывает все содержимое потока в массив байтов, независимо от положения курсора в потоке	Метод
WriteTo	Записывает все содержимое потока в другой поток типа Stream	Метод

Класс BufferedStream.

Этот поток является дополнительным буфером для данных. Используется для уменьшения количества обращений к реальному источнику или получателю данных, поскольку операции ввода и вывода относятся к медленно исполняемым операциям. Буферизованный поток является *потоком-оберткой*, то есть надстраивается над другим потоком типа `Stream`. Его функциональность унаследована от родительского класса `Stream`. Буферизованные потоки можно выстраивать в цепочки.

Пример:

```
FileStream fs = new FileStream(”test.dat”); //файловый поток
BufferredStream bs = new BufferedStream(fs); //поток-обертка
byte[] x = new byte[10];
bs.Write(x); //запись в буфер буферизованного потока
bs.Close(); //запись буфера в поток fs
fs.Close(); //запись данных на диск в файл
```

Текстовые потоки данных.

Потоки предназначены для работы с символьными данными в кодировке Unicode. Кодировку потока можно настроить с помощью констант перечисления `Encoding` из пространства имен `System.Text`.

Текстовые потоки разделены на потоки чтения и потоки записи. Каждая из групп имеет свой абстрактный базовый класс. Схема наследования для текстовых потоков приведена на рисунке.

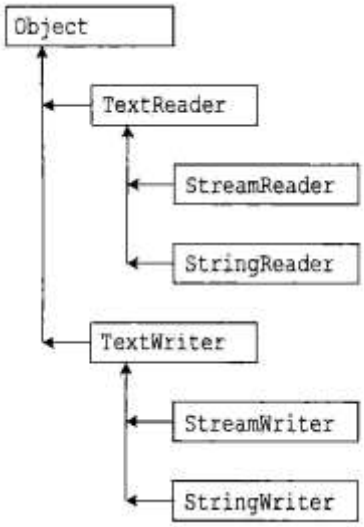


Рисунок 1 – Иерархия наследования для текстовых потоков данных

Класс TextWriter.

Является базовым абстрактным классом для текстовых потоков записи. Состав класса:

Элемент	Описание
Close	Закрыть файл и освободить связанные с ним ресурсы. Если в процессе записи используется буфер, он будет автоматически очищен
Flush	Очистить все буферы для текущего файла и записать накопленные в них данные в место их постоянного хранения. Сам файл при этом не закрывается
NewLine	Используется для задания последовательности символов, означающих начало новой строки. По умолчанию используется последовательность «возврат каретки» — «перевод строки» (\r\n)
Write	Записать фрагмент текста в поток
WriteLine	Записать строку в поток и перейти на другую строку

Класс TextReader.

Является базовым абстрактным классом для символьных потоков чтения. Состав класса:

Элемент	Описание
Peek	Возвратить следующий символ, не изменяя позицию указателя в файле
Read	Считать данные из входного потока
ReadBlock	Считать из входного потока указанное пользователем количество символов и записать их в буфер, начиная с заданной позиции
ReadLine	Считать строку из текущего потока и возвратить ее как значение типа string. Пустая строка (null) означает конец файла (EOF)
ReadToEnd	Считать все символы до конца потока, начиная с текущей позиции, и возвратить считанные данные как одну строку типа string

Наследниками описанных абстрактных классов являются потоки чтения и записи в текстовый файл – `StreamWriter` и `StreamReader` соответственно. Они реализуют функциональность своего предка для работы с файлом как с текстовым потоком записи или чтения и имеют дополнительную функциональность, связанную с созданием и открытием файлов.

Пример:

```
…
FileInfo f = new FileInfo(”my.txt”);
StreamWriter sw = f.CreateText();
sw.WriteLine(”Новый файл”);
for (int i = 0; i < 10; i++)
{
   sw.Write(i + ”  ”);
}
sw.Write(sw.NewLine);
sw.Close();
StreamReader sr = f.OpenText();
string s = null;
while (s = sr.ReadLine() != null)
   Console.WriteLine(s);
sr.Close();
…
```

Нужно отметить, что методы `Write` и `WriteLine` класса `StreamWriter` имеют множество перегрузок для записи в файл значений разных типов (как и в классе `Console`).

В классе `StreamWriter` описано булевское свойство `AutoFlush`, если установить его в true, то очищение буфера потока будет производиться после каждой записи в поток. То есть каждый метод записи будет передавать данные через внутренний буфер потока прямо в файл.

Наследниками абстрактных классов `TextWriter` и `TextReader` являются классы `StringWriter` и `StringReader` соответственно.

Аналогично потоку MemoryStream, эти классы позволяют работать с символьными потоками в динамической памяти.

Пример.

```
…
StringWriter stw = new StringWriter();
stw.WriteLine(”Поток в динамической памяти”);
for (int i = 0; i < 10; i++)
{
   stw.Write(i + ” ”);
}
stw.Write(stw.NewLine);
str.Close();
Console.WriteLine(stw.ToString());
StringReader str=new StringReader(”Поток в динамической памяти”);
string s = null;
while (s = str.ReadLine() != null)
   Console.WriteLine(s);
str.Close();
…
```

Потоки в динамической памяти можно интерпретировать как временные хранилища данных, которые после формирования их содержимого нужно отправить в постоянное хранилище.

В классе StringWriter определен метод получения еще одного временного хранилища символьных данных – объекта класса StringBuilder – метод GetStringBuilder.

Нужно отметить, что текстовые потоки не поддерживают произвольный доступ, и не имеют в своем составе методов управления курсором.

Двоичные потоки-обертки

Часто в байтовые (бинарные, двоичные) потоки приходится записывать значения простых типов данных. В байтовых потоках запись и чтение производится побайтно, однако значение простого типа данных может занимать гораздо больше одного байта. Значит, приходится преобразовывать это значение к набору байтов и записывать их в байтовый поток. При чтении реализуется обратное – чтение набора байт в необходимом количестве и конфигурирование из них многобайтного значения. Алгоритмы сборки-разборки могут реализовываться программистом самостоятельно, однако для универсальности и корректности прямых и обратных преобразований используются специальные классы BinaryWriter и BinaryReader.

Это байтовые потоки данных, являющиеся прямыми наследниками класса Object, которые относятся к потокам-оберткам, то есть надстраиваются над любым потоком типа Stream. Их основная функциональность – запись или чтение значений простых типов данных в двоичном виде в байтовый поток, над которым надстроен поток-обертка.

Состав класса BinaryWriter:

Элемент	Описание
BaseStream	Базовый поток, с которым работает объект BinaryWriter
Close	Закрыть поток
Flush	Очистить буфер
Seek	Установить позицию в текущем потоке
Write	Записать значение в текущий поток

Состав класса BinaryReader:

Элемент	Описание
BaseStream	Базовый поток, с которым работает объект BinaryReader
Close	Закрыть поток
PeekChar	Возвратить следующий символ без перемещения внутреннего указателя в потоке
Read	Считать поток байтов или символов и сохранить в массиве, передаваемом как входной параметр
ReadXXXX	Считать из потока данные определенного типа (например, ReadBoolean, ReadByte, ReadInt32 и т. д.)

Пример:

```
…
FileStream fs = new FileStream(”my.dat”, FileMode.OpenOrCreate,
FileAccess.ReadWrite, FileShare.None);
BinaryWriter bw = new BinaryWriter(fs);
bw.WriteString(”String в двоичном формате”);
int a = 10;
float b = 2.5;
bool c = false;
char[] chars = {’a’, ’b’, ’c’};
bw.Write(a);
bw.Write(b);
bw.Write(c);
bw.Write(chars);
bw.BaseStream.Position = 0;
BinaryReader br = new BinaryReader(fs);
int t = 0;
while (br.PeekChar() != -1)
{
   Console.Write(br.ReadByte());
   t++;
   if (t == 5)
   {
      t = 0;
      Console.WriteLine(); //каждые 5 байт переводим курсор на новую строку
   }
}
bw.Close();
br.Close();
f.Close();
…
```

С помощью потока-обертки из байтового файла можно считать необходимое количество байт и автоматически сформировать значение необходимого типа. Однако не следует забывать, что чтение все равно производится побайтно, то есть считать можно совсем не ту информацию, которая была записана, если при чтении не знать порядок записи.

Сериализация объектов.

При работе с байтовыми потоками состояние объекта пишется в поток поэлементно, с последующим поэлементным же считыванием. Однако в некоторых ситуациях удобно записывать в поток объект целиком. Процесс преобразования объекта в линейный поток байтов называется байтовой *сериализацией*. Обратный процесс считывания из линейного потока и формирования объекта называется *десериализацией*.

Объект может содержать ссылки на другие объекты, которые в свою очередь содержат ссылки на другие объекты (и т.д.). Для описания этих сложных связей необходимо построение так называемого *графа объектов* (или дерева объектов). При сериализации сохраняется сам граф объектов и состояние всех объектов, входящих в граф. При десериализации по считанному графу объектов восстанавливаются их связи и состояния.

Построение графа объектов и настройка сериализации производится автоматически. Для того, чтобы класс был сериализуем, нужно указать для него атрибут `Serializable`. Классы, помеченные этим атрибутом, могут быт сериализованы, то есть представлены в виде потока байтов.

Процессом сериализации занимается специальный внешний объект класса `Formatter`.

Существует два типа сериализации – байтовая и в XML формат. В данной теме речь идет о байтовой сериализации.

Двоичный форматер описан в пространстве имен `System.Runtime.Serialization.Formatters.Binary` и принадлежит классу `BinaryFormatter`.

Главными в этом классе являются два метода:

`Serialize`() – сериализация объекта в байтовый поток;
`Deserialize`() – десериализация объекта из байтового потока.

Пример:

```
[Serializable]
class Class1
{
   int n;
…
}

[Serializable]
class Class2
{
   Class1 c1 = new Class1();
```

```
…
}
…
Class2 c2 = new Class2();
FileStream fs = File.Create(”my.bin”);
BinaryFormatter bf = new BinaryFormatter();
bf.Serialize(fs, c2);
fs.Close();
fs = File.OpenRead(”my.bin”);
Class2 c3 = (Class2)bf.Deserialize(fs);
fs.Close();
…
```

Нужно отметить, что форматтер десериализует объект из потока и возвращает на него ссылку типа `Object`, поэтому необходимо явное преобразование к реальному типу объекта.

Если какие-то поля класса не нужно сохранять при сериализации, то их помечают атрибутом `NonSerialized`. При десериализации в несериализуемые поля объекта записываются значения по умолчанию. Не сериализуются также статические поля, поскольку они не принадлежат объекту.

Тема 12. Делегаты. События.

Делегаты.

Делегат - это специальный класс, объект которого предназначен для хранения ссылок на методы. Делегат обычно передаётся в качестве параметра и приводит к вызову всех методов, на которые ссылается. Синтаксис делегата описывает сигнатуру методов, которые могут быть вызваны с его помощью.

Синтаксис делегата:

```
<атрибуты> <спецификаторы> delegate <тип> <имя>(<список параметров>)
```

Спецификаторы могут быть следующими - `new`, `public`, `protected`, `internal`, `private`.

Тип – это тип возвращаемого значения из методов, вызываемых с помощью делегата, список параметров – перечисление параметров этих методов с указанием их типов.

Делегат может хранить ссылки на несколько методов, их сигнатуры должны совпадать, методы вызываются поочерёдно. Наследовать от делегата запрещено, объявлять делегат можно как в классе, так и непосредственно в пространстве. Для использования делегата нужно создать его экземпляр и зарегистрировать в нём методы. При вызове экземпляра делегата вызываются все методы, ссылки на которые он хранит. Делегаты позволяют определить исполняемый метод во время выполнения программы, а не на этапе компиляции, позволяет создавать методы, вызывающие другие методы. Делегат предназначен для реализации механизма событий.

Пример:

```
delegate int Del(int a, int b)
class Class
{
     public int Sum(int a, intb)
     {
        return a+b;
     }
     public int Sub(int  a, int b)
     {
        return a+b;
     }
}
class Program
{ static void Main()
     {
     int a=5;
     int b=2;
     Del d;
     Class1 c=new Class1();
     D=new Del(c.Sum);
     int e =d(a,b);
     d=new Del(c.Sub);
```

```
     int f=d(a,b);
     }
}
```

Использование делегата - тот же вызов метода. Если в делегате хранятся ссылки на несколько методов, то они выполняются последовательно, и изменения, вносимые одним методом, влияют на последующие. Формирование списка методов, вызываемых с помощью делегата, производится с помощью операции сложения. Удаление из списка – с помощью операции вычитания (либо методы `Combine` и `Remove` у объекта делегата).

Пример

```
class Class
{public static int Sum(int a, int b)
     {Console.WriteLine("Сумма равна {0},a+b");
     return a+b;
     }
public static int Sub(int a, int b)
     {Console.WriteLine("Разность равна {0}",a-b);
     }
}
delegate int Del(int a, int b);
class Program
{static void Main()
     {int a=10;
     int b=6;
     Del d=new Del(Class1.Sum);
     d+=new Del(Class.Sub);
     int c=d(a,b);
     Console.WriteLine("Результат выполнения делегата {0}",c);
     d-=new Del(Class.Sub);
     int e=d(a,b);
     Console.WriteLine("Результат выполнения делегата {0}",e);
     }
}
```

В делегат можно передавать обычные методы (по имени объекта) и статические (по имени класса), сигнатура методов должна полностью соответствовать делегату. Если параметры передаются не по значению, а по ссылке, то изменение состояния параметра в одном методе из списка влияет на параметр, передаваемый в последующие методы, поскольку методы в списке вызова делегата выполняются последовательно, набор параметров в методы передаётся также последовательно. Если метод возвращает значение, то из делегата будет возвращено то, которое возвращено из последнего метода в списке делегата. Поэтому рекомендуется использовать методы , возвращающие не void. Если вызвать на исполнение делегат, в списке которого нет методов, то будет сгенерировано исключение `NullReferenceException`. Если в методе из списка делегата возникло исключение, то

1) исключение, обработанное в том же методе, никак не влияет на выполнение последующих методов;

2) если исключение в методе не обработано, то последующие методы не выполняются, а производится поиск обработчика исключения в методе, вызывающем делегат.

Делегаты в параметрах методов

Делегат – это ссылка на методы, и объект специального класса, его можно передавать в методы. Таким образом создаются универсальные методы, которые вызывают другие методы, причём заранее неизвестно, какие, они передаются в виде делегата.

Пример

```
delegate double Function(double x)
class Class1
{public static void CountFunction(Function f,double a,double b,
double dx)
     {for ()double x=a; x<=b; x+=dx
          {Console.WriteLine("x={0,5:0.##}
y={1,5:0.##}",x,f(x));}
     }
public static void Sinus(double x)
     { return Math.Sin(x);}
public static void Cosinus(double x)
     { return Math.Cos(x);}
public static void Const(double x)
     { return 3;}
}
class Program
{ static void Main()
     { Console.WriteLine("Функция синуса");
     Function d=new Function(Class1.Sinus);
     Class1.CountFunction(d,-2,2,0.1);
     d=new Function(Class1.Cosinus);
     Console.WriteLine("Функция косинуса");
     Class1.CountFunction(d,-2,2,0.05);
     Console.WriteLine("Функция y=3");
     Class1.CountFunction(new Function(class1.Count),0,5,1);
     }
}
```

На данной технологии основан механизм обратных вызовов: описывается метод, содержащий бизнес-логику, этот метод через делегат передаётся в стандартную функцию, а она вызывает делегат, то есть передаёт вызов методу с бизнес-логикой (обратный вызов).

Операции с делегатами

Делегаты можно сравнивать на равенство и неравенство. Делегаты считаются равными, если они содержат ссылки на одни и те же методы в одном и том же порядке, либо не содержат методы. Делегаты, имеющие один тип возвращаемого значения и одинаковые списки параметров, не регулирующиеся по имени класса также можно сравнивать, на равенство и

неравенство, хотя такие делегаты считаются принадлежащими разным типам. Делегат является неизменяемым типом данных, то есть при его изменении создаётся новый экземпляр, а старый теряется. Делегаты одного типа можно складывать и вычитать.

…….

```
Del d1=new Del(Class1.Sum);
Del d2=new Del(Class1.Sub);
Del d3=new Del(Class1.Sub);
d3=d1+d2;
d3+=d1;
d3+=d2;
d3-=d1;
```

……

Пример

```
delegate void Del();
class Class1
{ public static void Method1()
     { Console.WriteLine("Method1");}
public static void Method2()
     { Console.WriteLine("Method2");}
public static void Method3()
     { Console.WriteLine("Method3");}
}
class Program
{ static void Main()
     { Del d=new Del(Class1.Method1);
     d+=new Del(Class1.Method2);
     d+=new Del(Class1.Method3);
     d();
     { Console.WriteLine(c.Message);}
// альтернатива:
foreach (Del d1 in d.GetInckabicntist())
     { try
          { d1();}
      catch (Exception c)
          { Console.WriteLine(c.Message);}
     }
}
}
```

<u>События</u>

События – это элемент класса, позволяющий послать другим объектам извещение об изменении своего состояния, после чего в объектах-получателях запускаются обработчики данного события. Обработчики обязательно должны быть зарегистрированы в источнике события. События основываются на делегатах, через делегат вызываются методы обработчики событий. Порядок создания события следующий:

1) Описывается делегат, который задаёт сигнатуру образчиков, данного события;
2) Описывается само событие в классе-источнике;

3) Описывается метод, генерирующий событие (в классе-источнике);
4) В классах-получателях описываются методы-обработчики события, совпадающие по сигнатуре с делегатом;
5) Методы-обработчики регистрируются в событиях (добавляются в событие через делегат);
6) Инициализируется событие.

Синтаксис события

```
<атрибуты><спецификаторы>event<тип><имя>;
```

спецификаторы: new, public, protected, internal, private, static, virtual, sealed, override, abstract.

Тип – тип делегата, на котором основано событие, то есть через этот делегат регистрируются обработчики события.

```
…..
delegate void Del();
class Class
{ public event Del.Ev;
…….
}
……
```

Для событий определены только операции += (добавление метода-обработчика в список обработчиков событий) и - = (удаление метода-обработчика из списка).

Пример:

```
public delegate void Del();
class Ist
{ public event Del Event;
public void EventGen();
     { Console.WriteLine("Генерация события");
if (Sob!=null) Sob();}
}
class Obr1
{ public void EventHard1()
     {Console.WriteLine("Обработка состояния 1") ;
}
 class Obr2
{ public void EventHand2()
     {Console.WriteLine("Обработка состояния 2") ;
}
class Obr3
{ public void EventHand3()
     {Console.WriteLine("Обработка состояния 3") ;
}
class Program
{ static void Main()
     { Ist i=new Ist();
     Obr1 01=new Obr1();
     Obr1 02=new Obr2();
     Obr1 03=new Obr3();
     i.Event +=new Del(01.EventHand1);
```

```
        i.Event +=new Del(02.EventHand2);
        i.Event +=new Del(03.EventHand1);
        i.Event +=new Del(03.EventHand3);
        i.Event Gen();
        }
}
```

Вне класса, в котором описано событие, с ним можно производить только добавление или удаление обработчиков сложением и вычитанием.

Внутри класса, в котором описано событие, с ним производятся те действия, которые допустимы для делегата (сравнение, присваивание). Событие – это поле типа делегат. Поле может быть статическим.

Механизм событий применяется в программировании под Windows.

Тема 13. Программирование под Windows.

В основу программирования под Windows положен принцип событийного управления. Это значит, что и сама система, и приложения после запуска ожидают действий пользователя и реагируют на них заранее заданным образом. Любое действие пользователя (нажатие клавиши на клавиатуре, щелчок кнопкой мыши, перемещение мыши) называется событием. Событие воспринимается операционной системой и преобразуется в сообщение — запись, содержащую необходимую информацию о событии (например, какая клавиша была нажата, в каком месте экрана произошел щелчок мышью). Сообщения могут поступать не только от пользователя, но и от самой системы, а также от активного или других приложений. Определен достаточно широкий круг стандартных сообщений, образующий иерархию, кроме того, можно определять собственные сообщения.

Сообщения поступают в общую очередь, откуда распределяются по очередям приложений. Каждое приложение содержит цикл обработки сообщений, который выбирает сообщение из очереди и через операционную систему вызывает подпрограмму, предназначенную для его обработки. Таким образом, Windows-приложение состоит из главной программы, обеспечивающей инициализацию и завершение приложения, цикла обработки сообщений и набора обработчиков событий.

Среда Visual Studio содержит удобные средства разработки Windows-приложений, выполняющие вместо программиста рутинную работу — создание шаблонов приложения и форм, заготовок обработчиков событий, организацию цикла обработки сообщений и т. д.

Приложение представляет собой шаблон (визуальную форму) – окно, предназначенное для размещения визуальных *компонент* (кнопок, меток, текстовых окон и т.д.). Размещение компонент на форме производится средствами визуальной среды разработки.

Процесс создания Windows-приложения состоит из трех основных этапов:

1. Разработка иерархии классов, описывающих предметную область решаемой задачи и реализующих необходимую функциональность обработки данных, бизнес-логику. Этот этап называется разработкой модели данных и является независимым.

2. Визуальное проектирование, то есть задание внешнего облика приложения из существующего набора визуальных компонент. Основная цель данного этапа – формирование полного и адекватного пользовательского представления для описанной в первом этапе модели данных. Этот этап называется разработкой вида (или представления). Данный этап зависит от предыдущего.

3. Определение поведения приложения путем написания обработчиков событий от визуальных компонент вида. Цель этого этапа – сформировать взаимодействие вида с пользователем. Через это взаимодействие

пользователь может вызывать на исполнение бизнес-методы модели данных, изменять состояние хранимых в ней данных. Этот этап называется разработкой контроллера. Контроллер обеспечивает корректное взаимодействие между моделью и видом.

Три описанных этапа соответствуют шаблону проектирования программных систем, называемому «Модель-Вид-Контроллер» или Model-View-Controller (MVC). Этот шаблон описывает правила разработки программных приложений вообще. Вторая часть шаблона – Вид – в данной теме описывается именно как приложение в стиле Windows, а третья часть – Контроллер – логично вытекает из второй.

Пользовательское представление – Вид – в Windows-приложении представляет собой набор взаимосвязанных визуальных компонент, размещенных на форме – так называемый фрейм, или каркас. Набор визуальных компонент сейчас довольно широк и постоянно расширяется бесплатными и платными библиотеками.

Классы стандартных визуальных компонент собраны в пространстве имен `System.Windows.Forms`. Состав пространства имен:

Класс	Назначение
Application	Класс Windows-приложения. При помощи методов этого класса можно обрабатывать Windows-сообщения, запускать и прекращать работу приложения и т. п.
ButtonBase, Button, CheckBox, ComboBox, DataGrid, GroupBox, ListBox, LinkLabel, PictureBox	Примеры классов, представляющих элементы управления (компоненты): базовый класс кнопок, кнопка, флажок, комбинированный список, таблица, группа, список, метка с гиперссылкой, изображение
Form	Класс формы — окно Windows-приложения
ColorDialog, FileDialog, FontDialog, PrintPreviewDialog	Примеры стандартных диалоговых окон для выбора цветов, файлов, шрифтов, окно предварительного просмотра
Menu, MainMenu, MenuItem, ContextMenu	Классы выпадающих и контекстных меню
Clipboard, Help, Timer, Screen, ToolTip, Cursors	Вспомогательные типы для организации графических интерфейсов: буфер обмена, помощь, таймер, экран, подсказка, указатели мыши
StatusBar, Splitter, ToolBar, ScrollBar	Примеры дополнительных элементов управления, размещаемых на форме: строка состояния, разделитель, панель инструментов и т. д.

Класс Control.

Класс является базовым для всех визуальных компонент, каждый класс которых наследует перечисленные ниже элементы класса `Control`:

Основные свойства:

Свойство	Описание
Anchor	Определяет, какие края элемента управления будут привязаны к краям родительского контейнера. Если задать привязку всех краев, элемент будет изменять размеры вместе с родительским

BackColor, BackgroundImage, Font, ForeColor, Cursor	Определяют параметры отображения рабочей области формы: цвет фона, фоновый рисунок, шрифт, цвет текста, вид указателя мыши
Bottom, Right	Координаты нижнего правого угла элемента. Могут устанавливаться также через свойство Size
Top, Left	Координаты верхнего левого угла элемента. Эквивалентны свойству Location
Bounds	Возвращает объект типа Rectangle (прямоугольник), который определяет размеры элемента управления
ClientRectangle	Возвращает объект Rectangle, *определяющий размеры рабочей области элемента*
ContextMenu	Определяет, какое контекстное меню будет выводиться при щелчке на элементе правой кнопкой мыши
Dock	Определяет, у какого края родительского контейнера будет отображаться элемент управления
Location	Координаты верхнего левого угла элемента относительно верхнего левого угла контейнера, содержащего этот элемент, в виде структуры типа Point. Структура содержит свойства X и Y
Height, Width	Высота и ширина элемента
Size	Высота и ширина элемента в виде структуры типа Size. Структура содержит свойства Height и Width
Created, Disposed, Enabled, Focused, Visible	Возвращают значения типа bool, определяющие текущее состояние элемента: создан, удален, использование разрешено, имеет фокус ввода, видимый
Handle	Возвращает дескриптор элемента (уникальное целочисленное значение, сопоставленное элементу)
ModifierKeys	Статическое свойство, используемое для проверки состояния модифицирующих клавиш (Shift, Control, Alt). Возвращает результат в виде объекта типа Keys
MouseButtons	Статическое свойство, проверяющее состояние клавиш мыши. Возвращает результат в виде объекта типа MouseButtons
Opacity	Определяет степень прозрачности элемента управления. Может изменяться от 0 (прозрачный) до 1 (непрозрачный)
Parent	Возвращает объект, родительский по отношению к данному (имеется в виду не базовый класс, а объект-владелец)
Region	Определяет объект Region, при помощи которого можно управлять очертаниями и границами элемента управления
TabIndex, TabStop	Используются для настройки последовательности перемещения с помощью клавиши Tab по элементам управления, расположенным на форме

Основные методы:

Метод	**Описание**
Focus	Установка фокуса ввода на элемент[1]
GetStyle, SetStyle	Получение и установка флагов управления стилем элемента. Используются значения перечисления ControlStyles (см. далее)
Hide, Show	Управление свойством Visible (Hide — скрыть элемент, Show — отобразить элемент)

Invalidate	Обновление изображения элемента путем отправки соответствующего сообщения в очередь сообщений. Метод перегружен таким образом, чтобы можно было обновлять не всю область, занимаемую элементом, а лишь ее часть
On*XXXX*	Методы-обработчики событий (OnMouseMove, OnKeyDown, OnResize, OnPaint и т. п.), которые могут быть замещены в производных классах
Refresh	Обновление элемента и всех его дочерних элементов
SetBounds, SetLocation, SetClientArea	Управление размером и положением элемента

События:

Событие	Описание
Click, *DoubleClick*, *MouseEnter*, *MouseLeave*, MouseDown, MouseUp, MouseMove, MouseWheel	События от мыши
KeyPress, KeyUp, KeyDown	События от клавиатуры
BackColorChanged, ContextMenuChanged, FontChanged, Move, Paint, Resize	События изменения элемента
GotFocus, Leave, LostFocus	События получения и потери фокуса ввода

Обработка событий.

События класса `Control` могут быть сгенерированы любым визуальным компонентом. Другой визуальный компонент (в том числе и сама форма) должен на сгенерированное событие реагировать.

Подход к обработке событий был описан в теме 12. Используется следующая схема:

1. Компонент, являющийся элементом пользовательского интерфейса, генерирует событие, спровоцированное пользователем, то есть служит источником события. При этом средствами среды исполнения вместо системного события взаимодействия пользователя с устройством ввода-вывода генерируется одно наиболее подходящее событие класса Control.

2. Компонент, являющийся слушателем события, должен быть зарегистрирован в источнике для получения сообщения о наступлении события.

3. В момент генерирования события в источнике зарегистрированный слушатель этого события синхронизирует свое состояние, получая от источника объект, содержащий параметры наступившего события. То есть слушатель исполняет свой метод-обработчик наступившего события.

Схема реализуется с помощью описанного в теме 12 механизма делегатов и событий.

В языке C#, для того, чтобы метод компонента мог быть зарегистрирован как обработчик какого-либо события, он должен удовлетворять сигнатуре обработчика, заданной соответствующим делегатом. То есть для различных стандартных категорий событий описаны

стандартные делегаты, задающие сигнатуру обработчиков событий этой категории:

`MouseEventHandler` – делегат для обработчиков событий мыши;

`KeyEventHandler` – делегат для обработчиков событий клавиатуры;

`PaintEventHandler` – делегат для обработчиков событий перерисовки;

`EventHandler` – делегат для обработчиков событий общего вида (является родительским для первых трех).

Каждому из четырех делегатов соответствует класс, объект которого содержит параметры наступившего события – `MouseEventArgs`, `KeyEventArgs`, `PaintEventArgs`, `EventArgs` (родительский для первых трех).

Сигнатура обработчика события:

```
<спецификатор> void <ИмяОбъекта_OnИмяСобытия> (object sender, объект параметров события)
{…}
```

Спецификатор – `public`, `private`, `protected` и т.д.

ИмяОбъекта – имя того объекта, для которого регистрируется обработчик, то есть объекта – источника события.

Имя события – одно из стандартных событий компонента-источника.

Объект параметров события – объект класса `EventArgs` или производный от него

Пример:

```
private void Form1_OnDoubleClick (object sender, EventArgs e)
public void Button1_OnMouseClick(object sender,MouseEventArgs e)
```

Методы-обработчики события регистрируются в компоненте-источнике через делегат, соответствующий категории прослушиваемого события.

Свойства класса `MouseEventArgs`:

- `Button` – определяет, какая кнопка мыши нажата
- `Clicks` – определяет, сколько раз нажата и отпущена кнопка мыши
- `Delta` – информация о повороте колесика мыши
- `X` – координата X указателя мыши во время щелчка
- `Y` – координата Y указателя мыши во время щелчка

Свойства класса `KeyEventArgs`:

- `Alt` – определяет, нажата ли клавиша Alt
- `Control` – определяет, нажата ли клавиша Control
- `Handled` – информация о том, обрабатывается ли данное события
- `KeyCode` – код клавиши при событиях KeyDown, KeyUp
- `KeyData` – данные о нажатых клавишах и их комбинациях со служебными клавишами
- `Modifiers` – информация о нажатых управляющих клавишах
- `Shift` – определяет, нажата ли клавиша Shift

Объект, содержащий параметры события принимается в метод-обработчик, его свойства доступны в методе-обработчике соответствующего события.

Иерархия визуальных компонент довольно обширна, классы визуальных компонент относятся к справочной информации и могут быть рассмотрены самостоятельно.

Класс Form.

Класс описывает окно, на котором могут быть размещены другие визуальные и невизуальные компоненты.

Все окна можно делить на модальные и немодальные. Модальное окно не позволяет пользователю переключаться на другие окна того же приложения, пока не будет завершена работа с текущим окном. Системное модальное окно не позволяет пользователю переключаться на другие окна вообще, пока не будет завершена работа с текущим окном. Немодальное окно позволяет переключение на другие окна.

Каждое приложение содержит главное окно – главную форму, и дочерние окна. При окончании работы с главной формой все ее дочерние формы закрываются автоматически.

Состав класса Form.

Основные свойства:

Свойство	Описание
AcceptButton	Позволяет задать кнопку или получить информацию о кнопке, которая будет активизирована при нажатии пользователем клавиши Enter
ActiveMDIChild, IsMDIChild, IsMDIContainer	Свойства предназначены для использования в приложениях с многодокументным интерфейсом (MDI)
AutoScale	Позволяет установить или получить значение, определяющее, будет ли форма автоматически изменять свои размеры, чтобы соответствовать высоте шрифта, используемого на форме, или размерам размещенных на ней компонентов
FormBorderStyle	Позволяет установить или получить стиль рамки вокруг формы (используются значения перечисления FormBorderStyle)
CancelButton	Позволяет задать кнопку или получить информацию о кнопке, которая будет активизирована при нажатии пользователем клавиши Esc
ControlBox	Позволяет установить или получить значение, определяющее, будет ли присутствовать стандартная кнопка системного меню в верхнем левом углу заголовка формы
Menu, MergedMenu	Используются для установки или получения информации о меню на форме
MaximizeBox, MinimizedBox	Определяют, будут ли на форме присутствовать стандартные кнопки восстановления и свертывания в правом верхнем углу заголовка формы

ShowInTaskbar	Определяет, будет ли форма отображаться на панели задач Windows
StartPosition	Позволяет получить или установить значение, определяющее исходное положение формы в момент выполнения программы (используются значения перечисления FormStartPosition). Например, для отображения формы в центре экрана устанавливается значение CenterScreen
WindowState	Определяет состояние отображаемой формы при запуске (используются значения перечисления FormWindowState)

Методы:

Метод	Описание
Activate	Активизирует форму и помещает в нее фокус ввода
Close	Закрывает форму
CenterToScreen	Помещает форму в центр экрана
LayoutMDI	Размещает все дочерние формы на родительской в соответствии со значениями перечисления LayoutMDI
OnResize	Может быть замещен для реагирования на событие Resize
Show	Отображает форму (унаследовано от Control)
ShowDialog	Отображает форму как диалоговое окно (подробнее о диалоговых окнах рассказывается в следующем разделе)

События:

Событие	Описание
Activate	Происходит при активизации формы (когда она выходит в активном приложении на передний план)
Closed, Closing	Происходят во время закрытия формы
MDIChildActive	Возникает при активизации дочернего окна

Диалоговые окна.

Диалоговые окна, как следует из их названия, предназначены для организации диалога с пользователем. Они должны обладать определенным набором характеристик. Специального класса для построения диалоговых окон в языке нет. Используется объект класса `Form` со следующими параметрами:

- размеры окна не изменяются – свойство `FormBorderStyle = FixedDialog`:
- кнопок сворачивания и восстановления в правом верхнем углу нет – свойства `MaximazeBox = false`, `MinimazeBox = false`;
- наличие диалоговых кнопок, после нажатия которых устанавливается конкретное значение возвращаемого результата (задается в свойстве `DialogResult` у кнопки);
- для отображения окна используется метод `ShowModal()`, возвращающий значение нажатой кнопки. Значение является константой перечисления `DialogResult.`

Перечисление `DialogResult`:

Значение	Описание	Значение	Описание
None	Окно не закрывается	Ignore	Нажата кнопка Ignore
OK	Нажата кнопка OK	Yes	Нажата кнопка Yes
Cancel	Нажата кнопка Cancel	No	Нажата кнопка No
Abort	Нажата кнопка Abort	Retry	Нажата кнопка Retry

Класс Application.

Класс содержит статические элементы для управления приложением в целом. Состав класса `Application`:

Элемент класса	Тип	Описание
AddMessageFilter, RemoveMessageFilter	Методы	Позволяют перехватывать сообщения и выполнять с этими сообщениями нужные предварительные действия. Для того чтобы добавить фильтр сообщений, необходимо указать класс, реализующий интерфейс IMessageFilter[1]
DoEvents	Метод	Обеспечивает способность приложения обрабатывать сообщения из очереди сообщений во время выполнения какой-либо длительной операции
Exit	Метод	Завершает работу приложения
ExitThread	Метод	Прекращает обработку сообщений для текущего потока и закрывает все окна, владельцем которых является этот поток
Run	Метод	Запускает стандартный цикл обработки сообщений для текущего потока
CommonAppDataRegistry	Свойство	Возвращает параметр системного реестра, который хранит общую для всех пользователей информацию о приложении
CompanyName	Свойство	Возвращает имя компании
CurrentCulture	Свойство	Позволяет задать или получить информацию о естественном языке, для работы с которым предназначен текущий поток
CurrentInputLanguage	Свойство	Позволяет задать или получить информацию о естественном языке для ввода информации, получаемой текущим потоком
ProductName	Свойство	Позволяет получить имя программного продукта, которое ассоциировано с данным приложением
ProductVersion	Свойство	Позволяет получить номер версии программного продукта
StartupPath	Свойство	Позволяет определить имя выполняемого файла для работающего приложения и путь к нему в операционной системе
ApplicationExit	Событие	Возникает при завершении приложения
Idle	Событие	Возникает, когда все текущие сообщения в очереди обработаны и приложение переходит в режим бездействия
ThreadExit	Событие	Возникает при завершении работы потока в приложении. Если работу завершает главный поток приложения, это событие возникает до события ApplicationExit

Введение в графику.

Каждый визуальный компонент является контейнером, на котором могут быть отрисованы графические примитивы.

Для вывода линий, текста и геометрических фигур используется объект класса `Graphics`, описанного в пространстве имен `System.Drawing`.

Существует несколько способов создания ссылки на такой объект:

1. Создание объекта с помощью метода `CreateGraphics`, описанного в классах компонент:

```
Graphics g = this.CreateGraphics();
```

2. Получение ссылки на объект из объекта параметров события перерисовки:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
}
```

3. Построение объекта на изображении:

```
BitMap bm = new BitMap(”d:\\picture.bmp”);
Graphics g = Graphics.FromImage(bm);
```

Третий способ обычно применяется для редактирования изображений.

Объект типа `Graphics` служит для вывода графических примитивов. В соответствующем пространстве имен также описаны вспомогательные классы – `Font`, `Pen`, `Brush`, `Color`.

Пример.

```
…
Graphics g = this.CreateGraphics();
Pen pen = new Pen(Color.Black);
g.DrawEllipse(pen, new Rectangle(1, 1, 100, 100));
g.DrawLine(pen, 1, 1, 100, 100);
g.DrawRectangle(pen, 1, 1, 20, 20);
…
```

Графические объекты занимают много динамической памяти, поэтому зачастую для них используют метод неявного освобождения ресурсов:

```
using (Pen pen = new Pen(Color.Red))
{
…
}
```

После выполнения блока графический объект, используемый под ключевым слово `using`, удаляется из динамической памяти автоматически.

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Самарский государственный аэрокосмический университет
имени академика С.П. Королёва (национальный исследовательский университет)» (СГАУ)

Факультет информатики
Кафедра программных систем

Дегтярева О.А.

Задания к лабораторным работам по дисциплине «Основы программирования»

Направление 010300.62 Фундаментальная информатика и информационные технологии

Курс 1
Семестр 1,2

Самара 2012

Лабораторная работа № 1

Задание 1

Протабулировать функцию в заданном диапазоне с заданным шагом. Шаг и диапазон табуляции задаются пользователем. В программе реализовать простое меню пользователя.

Задание 2

В беззнаковом целом числе поменять местами тетрады в каждом байте. Число вводится с клавиатуры пользователем. Предусмотреть следующие варианты ввода и вывода – десятичный ввод и десятичный вывод; двоичный ввод и двоичный вывод. При организации двоичного ввода и вывода можно воспользоваться классом String. В программе реализовать простое меню пользователя.

Лабораторная работа №2

Задание 1.

Создать класс «счетчик» со следующей структурой:
-поле состояния счетчика;
-методы для установки и чтения состояния счетчика;
-метод инкрементирования состояния счетчика;
-метод декрементирования состояния счетчика;
Класс описывает циклический счетчик. Проверять принадлежность состояния счетчика диапазону 0-100.
В классе Program протестировать функциональность описанного класса.

Задание 2.

Создать класс «Дробь» со следующей структурой:
-поле «числитель»;
-поле «знаменатель»;
-метод чтения числителя;
-метод чтения знаменателя;
-метод установки числителя;
-метод установки знаменателя с контролем неравенства
знаменателя нулю;
-статические методы сложения, умножения, вычитания и
деления дробей, принимающих в качестве параметров
две дроби и возвращающих результирующую дробь;
-реализовать приватный метод сокращения дроби.
В классе Program протестировать функциональность описанного класса.

Задание 3

Описать класс – многочлен типа ax^2+bx+c:
В классе должно быть описано следующее:
- поля – переменные вещественных типов – коэффициенты многочлена;
- конструктор с параметрами;
- конструктор по умолчанию;
- методы установки и чтения для каждого параметра;
- метод, возвращающий массив решений соответствующего квадратного уравнения;
- метод, возвращающий массив координат вершины соответствующей параболы;
- статические методы сложения, вычитания и умножения многочленов с созданием нового возвращаемого объекта.
Примечание. При умножении полиномов 2-й степени возможно получение результата, степень которого превышает степень описываемого класса. В этом случае из метода умножения нужно вернуть ссылку null. А после вызова метода умножения в методе Main() проверить равенство возвращенного результата null/ В этом случае вывести сообщение о том, что степень полинома больше 2-й.
В классе Program протестировать функциональность описанного класса.

Лабораторная работа №3

Реализовать решение следующих задач:

-подсчитать в строке число букв;

-подсчитать в строке среднюю длину слова (средняя длина – вещественное число);

-заменить в строке все вхождения заданного пользователем слова новым заданным пользователем словом (заменяются в строке именно слова, а не просто вхождения подстроки, слово должно быть обрамлено разделителями или пробелами, регистр игнорировать. Например, в строке «Привет – самое распространенное приветствие, привет также можно передать» заменить слово «привет» на слово «здорово». Результат «здорово – самое распространенное приветствие, здорово также можно передать»),

-подсчитать в строке количество вхождений заданной подстроки (именно подстроки, а не слова),

-проверить, является ли строка датой (формат даты – ДД.ММ.ГГ или ДД.ММ.ГГГГ),

-проверить, является ли строка палиндромом.

Лабораторная работа № 4

Задание 1.

Описать класс «вектор» с именем “ArrayVector” со следующей структурой:

- поле – массив элементов вещественного типа (координаты вектора в пространстве);
- конструктор с параметром – длиной массива;
- метод установки элемента массива по индексу SetElement (параметры метода – индекс элемента и устанавливаемое значение);
- метод чтения элемента массива по индексу GetElement (параметр метода – индекс элемента)
- метод получения модуля вектора GetNorm;
- метод подсчета суммы всех отрицательных элементов массива с четными номерами (SumNegativesFromChetIndex);
- метод подсчета суммы тех элементов массива, которые имеют нечетные номера и одновременно больше среднего значения всех модулей элементов массива (SumLagersFromNechetIndex);
- метод подсчета произведения всех четных элементов (MulChet);
- метод подсчета произведения всех нечетных элементов, не делящихся на три (MulNechet);
- метод сортировки массива по возрастанию SortUp;
- метод сортировки массива по убыванию SortDown;

Протестировать работу описанного класса в классе Program, отлавливая все возможные исключения

Задание 2.

Описать класс пользовательского исключения IndexOutOfBordersException, наследного от класса Exception. Выбрасывать это исключение в случае выхода за границу вектора индекса в методах доступа к элементам вектора.

Протестировать работу в классе Program.

Лабораторная работа №5

Задание 1.

Привести класс «вектор» с именем “ArrayVector” к следующей структуре:
- поле – массив элементов вещественного типа
  (координаты вектора в пространстве);
- конструктор с параметром-длиной массива;
- конструктор без параметра, задающий длину массива 5;
- индексатор для организации доступа к элементам массива, выбрасывающий исключение IndexOutOfBordersException;
- метод получения модуля вектора GetNorm;
- свойство для чтения числа координат вектора;

Задание 2.

Описать класс «список» с именем «LinkedListVector», содержащий массив элементов вещественного типа в виде динамического односвязного списка. Каждая элемент массива представляет собой отдельный объект «узел » класса «Node». Структура класса «Node»:
- поле – элемент вещественного типа (по умолчанию = 0);
- поле – ссылка на элемент класса Node (по умолчанию = null);

Структура класса «LinkedListVector»:
- поле – ссылка на начало списка (на объект класса Node);
- конструктор с параметром – длиной списка;
- конструктор без параметра, задающий длину списка 5;
- индексатор для организации доступа к элементам массива, выбрасывающий исключение IndexOutOfBordersException;
- метод получения модуля вектора GetNorm;
- свойство для чтения числа координат вектора;

Задание 3.

Описать класс с именем «Vectors», содержащий публичные статические методы
- сложения двух векторов Sum (принимает в качестве параметра 2 объекта ArrayVector, возвращает новый объект ArrayVector);
- скалярного произведения двух векторов Scalar (принимает в качестве параметра 2 объекта ArrayVector, возвращает вещественное число);
- получения модуля вектора GetNorm (принимает в качестве параметра объект ArrayVector, возвращает вещественное число);

Описать класс исключения IncompatibleVectotsException, наследующий от класса Exception. Выбрасывать данное исключение в методах Sum и Scalar в случае невозможности проведения указанных действий над векторами.

Задание 4.

Описать интерфейс «Vector», содержащий следующие элементы:
- индексатор для организации доступа к элементам массива
- свойство для чтения числа координат вектора;
- метод получения модуля вектора GetNorm;

Сделать классы ArrayVector и LinkListVector реализующими интерфейс Vector и привести их в соответствие с описанной структурой наследования.

Поменять в классе со статическими методами Vectors типы параметров так, чтобы методы работали со ссылками типа интерфейс Vector.

Задание 5.

В классах ArrayVector и LinkListVector переопределить унаследованный от класса Object метод ToString(), который преобразует вектор в строку формата «число координат вектора координаты вектора». Использовать этот метод для вывода информации о векторе на экран.

В классах ArrayVector и LinkListVector переопределить унаследованный от класса метод Equals() таким образом, чтобы он сравнивал на равенство любой объект интерфейса Vector. Вектора считаются равными, если они равны по числу координат и равны покоординатно.

Сделать интерфейс Vector реализующим интерфейс ICloneable. В классах ArrayVector и LinkListVector реализовать метод Clone(), реализующий глубокое клонирование объектов.

Протестировать работу приложения в классе Program.

Лабораторная работа №6

Задание1.

Создать перечисление должностей Vacancies {Manager, Boss, Clerk, Salesman, etc.}

Задание 2.

Создать структуру «Employee» состоящую из:
- поля name строкового типа;
- поля vacancy типа Vacancies;
- поля зарплата целого типа;
- поля дата приема на работу типа int[3].

Задание 3.

В классе Program создать массив сотрудников. Длина массива задается пользователем, заполнение массива производится им же. Выполнить следующее:
- вывести полную информацию обо всех сотрудниках;
- найти в массиве всех менеджеров, зарплата которых больше средней зарплаты всех клерков, вывести на экран полную информацию о таких менеджерах отсортированную по их фамилии.
- распечатать информацию обо всех сотрудниках, принятых на работу позже босса, отсортированную в алфавитном порядке по фамилии сотрудника.

Примечание: в исходном массиве сотрудники хранятся в неотсортированном виде.

Лабораторная работа №7

Задание 1.

В классе Vector добавить следующие методы:

- записи вектора в байтовый поток
  void outputVector(Vector v, Stream out)
- чтения вектора из байтового потока
  Vector inputVector(Stream in),

Записанный вектор должен представлять собой последовательность чисел, первым из которых является размерность вектора, а остальные числа являются значениями координат вектора. Передачу данных простых типов осуществить с помощью потоков-оберток BinaryReader и BinaryWriter.

Проверить возможности методов в классе Program, в качестве байтового потока используя файловый поток (создать файл данных в текущей папке).

Задание 2.

Добавить в класс Vector следующие методы:

- записи вектора в символьный поток
  void writeVector(Vector v, TextWriter out),
- чтения вектора из символьного потока
  Vector readVector(TextReader in).

В данном случае рекомендуется считать, что один вектор записывается в одну строку (числа разделены пробелами, возможно использовать переопределенный метод ToString()). Для чтения вектора из символьного потока рекомендуется использовать метод Split() класса String.

Проверить возможности методов в классе Program, в качестве текстового потока используя файловый поток (создать текстовый файл в текущей папке).

Задание 3.

Модифицировать классы ArrayVector и LinkListVector таким образом, чтобы они были сериализуемыми.

Продемонстрировать возможности сериализации (в классе Program), записав в файл объект, затем считав и сравнив с исходным.

Примечание. Для сравнения объектов переопределить метод Equals(), унаследованный от класса Object.

Протестировать работу приложения в классе Program.

Лабораторная работа №8

Организовать работу меню в программе из лабораторных работ №5 и №7 с помощью механизма делегатов и событий.
- описать делегат без параметров, возвращающий void .
- в классе Program описать статическими методами все пункты меню. Сигнатура методов соответствует сигнатуре, заданной делегатом.
- в методе Main() описать переменную события типа делегат.
- добавлять через делегат в зависимости от выбора пользователя в качестве обработчика события статические методы, реализующие пункты меню.
- после регистрации соответствующего обработчика генерировать событие. Делегат автоматически вызовет тот статический метод, который является на данный момент обработчиком сгенерированного события.

Продемонстрировать результат работы.

Лабораторная работа №9

Выполнить лабораторные работы 5 и 7 как Windows-приложение.
Разработать графический интерфейс пользователя, адекватно отражающий функциональность приложения.
Приложение должно быть спроектировано в рамках архитектуры «Модель-Вид-Контроллер».

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Самарский государственный аэрокосмический университет
имени академика С.П. Королёва (национальный исследовательский университет)» (СГАУ)

Факультет информатики
Кафедра программных систем

Дегтярева О.А.

Задания к практическим работам по дисциплине «Основы программирования»

Направление 010300.62 Фундаментальная информатика и информационные технологии

Курс 1
Семестр 1

Самара 2012

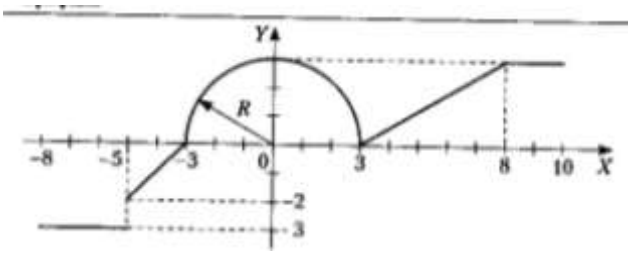
Практическая работа №1

Задание 1.
Изучение основ работы в среде разработки VisualStudio.

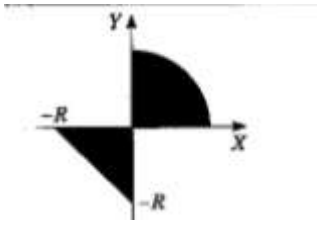
Создание проекта, сохранение проекта, создание, сохранение и добавление исходных кодов. Пункты главного меню среды.

Задание 2.
Изучение основ синтаксиса языка C#.

2.1 Написать программу, которая по значению аргумента X выводит значение функции Y, заданной на графике. Параметр R=3. График функции рисовать не нужно.



2.2
Написать программу, определяющую, попадает ли точка с введенными координатами в заштрихованную область. Координаты X и Y вводятся вручную. Параметр R вводится вручную. График функции рисовать не нужно.



2.3
Найти сумму ряда с заданной точностью

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \ldots$$

2.4
Смоделировать бросание двух кубиков. Если сумма выпавших очков четная – игрок выиграл, если не четная – проиграл.

Практическая работа №2

Задание 1.
Обработка одномерных массивов

При написании программ обработки одномерных массивов необходимо реализовывать следующее:
1. Задание количества элементов массива с клавиатуры.
2. Задание элементов массива с клавиатуры.
3. Возможность просмотра и изменения массива – полностью и по номеру элемента.
4. Простейший интерфейс пользователя.

1.1 Написать программу, которая в одномерном массиве ищет максимальный элемент и выводит на экран его значение и номер.

1.2 Написать программу, которая выводит все элементы одномерного массива, которые равны своему предыдущему элементу. Если таковых нет – выводится информационное сообщение об этом.

1.3 Написать программу поиска в одномерном массиве первого нулевого элемента. Если такового нет – выводится информационное сообщение об этом.

1.4 Написать программу подсчета среднего арифметического всех положительных элементов одномерного массива. Если положительных элементов в массиве нет – выводится информационное сообщение об этом.

Задание 2.
Обработка прямоугольных двумерных массивов (матриц)

При написании программ обработки матриц необходимо реализовывать следующее:
1. Задание размерности матрицы с клавиатуры.
2. Задание элементов матрицы с клавиатуры.
3. Возможность просмотра и изменения матрицы – полностью и по номеру элемента.
4. Простейший интерфейс пользователя.

2.1 Написать программу, которая из матрицы целых чисел получает одномерный массив, состоящий из сумм положительных элементов соответствующих строк матрицы.

2.2 Написать программу, которая из матрицы целых чисел получает массив, содержащий `true`, если в соответствующей строке матрицы есть хотя бы один нулевой элемент и `false` в противоположном случае.

2.3 Написать программу, реализующую умножение двух целочисленных матриц. При этом контролировать размерность исходных матриц. Если произведение получить невозможно – выводить информационное сообщение об этом.

2.4 Написать программу получения из квадратной матрицы целых чисел соответствующую по размерам матрицу, в которой элемент равен `true`, если соответствующие строка и столбец исходной матрицы являются ортогональными векторами, и `false` в противоположном случае.

Задание 3.
Обработка двумерных ступенчатых массивов.

Написать программу сортировки двумерного ступенчатого массива по возрастанию. При этом сохранить размерность ступенчатого массива. Например:

Исходный массив
```
1  4  0  -12
0  3
7  1  10
5
```
Результирующий массив:
```
-12  0  0  1
  1  3
  4  5  7
 10
```

Практическая работа №3

Работа с коллекциями.

Реализовать массив объектов в виде коллекции ArrayList. Проверить возможности сортировки коллекции по различным критериям с помощью интерфейсов IComparable, IComparer. Реализовать алгоритм глубокого клонирования объектов коллекции.

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Самарский государственный аэрокосмический университет
имени академика С.П. Королёва (национальный исследовательский университет)» (СГАУ)

Факультет информатики
Кафедра программных систем

Дегтярева О.А.

Темы для подготовки к зачету по дисциплине «Основы программирования»

Направление 010300.62 Фундаментальная информатика и информационные технологии

Курс 1
Семестр 1

Самара 2012

1 блок. Синтаксис языка C#.
Лексемы, идентификаторы, константы, правила именования. Типы данных, значимые и ссылочные типы данных, преобразование типов. Операции, операторы, конструкции программирования. Простейший ввод и вывод, форматный вывод, класс Console.

2 блок. Объектно-ориентированное программирование.
Принципы ООП – инкапсуляция, наследование, полиморфизм. Общий состав класса – конструкторы, поля, методы. Перегрузка, сокрытие, переопределение элементов класса. Спецификаторы, спецификаторы доступа.

3 блок. Состав класса в языке C#.
Конструкторы, поля, методы, специальные методы – свойства-аксессоры, операции класса (унарные, бинарные, отношения/сравнения, преобразования), индексаторы, деструкторы, вложенные классы, метод Main().

4 блок. Стандартные классы.
Массивы – одномерные, двумерные, ступенчатые, массивы объектов. Работа с массивами. Класс Array. Строки и символы, классы Char, String, StringBuilder. Работа со строками. Класс Random. Класс Object.

5 блок. Исключительные ситуации.
Исключения, отлов, выброс, проброс исключений. Правила обработки исключений. Класс Exception.

6 блок. Реализация принципов ООП в C#.
Реализация инкапсуляции, спецификаторы доступа. Реализация наследования – основные термины, наследование полей, конструкторов, методов и т.д. Реализация полиморфизма – виртуальные методы, абстрактные методы и абстрактные классы, бесплодные классы, модель «включение-делегирование»
Интерфейсы. Состав интерфейса, спецификаторы элементов интерфейса, получение ссылки типа интерфейс, наследование от интерфейсов, наследование для интерфейсов. Стандартные интерфейсы IComparable. IComparer, ICloneable.

7 блок. Особые классы.
Структуры, особенности описания и хранения, работа со структурами. Перечисления, описание, базовый тип перечисления, операции с перечислениями. Коллекции, особенности, предназначение, типы коллекций.

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Самарский государственный аэрокосмический университет
имени академика С.П. Королёва (национальный исследовательский университет)» (СГАУ)

Факультет информатики
Кафедра программных систем

Дегтярева О.А.

Темы для подготовки к экзамену по дисциплине «Основы программирования»

Направление 010300.62 Фундаментальная информатика и информационные технологии

Курс 1
Семестр 2

Самара 2012

1 блок. Синтаксис языка C#.
Лексемы, идентификаторы, константы, правила именования. Типы данных, значимые и ссылочные типы данных, преобразование типов. Операции, операторы, конструкции программирования. Простейший ввод и вывод, форматный вывод, класс Console.

2 блок. Объектно-ориентированное программирование.
Принципы ООП – инкапсуляция, наследование, полиморфизм. Общий состав класса – конструкторы, поля, методы. Перегрузка, сокрытие, переопределение элементов класса. Спецификаторы, спецификаторы доступа.

3 блок. Состав класса в языке C#.
Конструкторы, поля, методы, специальные методы – свойства-аксессоры, операции класса (унарные, бинарные, отношения/сравнения, преобразования), индексаторы, деструкторы, вложенные классы, метод Main().

4 блок. Стандартные классы.
Массивы – одномерные, двумерные, ступенчатые, массивы объектов. Работа с массивами. Класс Array. Строки и символы, классы Char, String, StringBuilder. Работа со строками. Класс Random. Класс Object.

5 блок. Исключительные ситуации.
Исключения, отлов, выброс, проброс исключений. Правила обработки исключений. Класс Exception.

6 блок. Реализация принципов ООП в C#.
Реализация инкапсуляции, спецификаторы доступа. Реализация наследования – основные термины, наследование полей, конструкторов, методов и т.д. Реализация полиморфизма – виртуальные методы, абстрактные методы и абстрактные классы, бесплодные классы, модель «включение-делегирование»
Интерфейсы. Состав интерфейса, спецификаторы элементов интерфейса, получение ссылки типа интерфейс, наследование от интерфейсов, наследование для интерфейсов. Стандартные интерфейсы IComparable. IComparer, ICloneable.

7 блок. Особые классы.
Структуры, особенности описания и хранения, работа со структурами. Перечисления, описание, базовый тип перечисления, операции с перечислениями. Коллекции, особенности, предназначение, типы коллекций.

8 блок. Работа с объектами файловой системы
Классы Directory и DirectoryInfo, особенности и отличия, классы File и FileInfo, особенности и отличия.

9 блок. Схема потоков данных.
Основные типы потоки данных, иерархия потоков данных в C#. Байтовые потоки Stream (FileStream, MemoryStream, BufferedStream). Текстовые потоки записи TextWriter (StreamWriter, StringWriter), текстовые потоки чтения TextReader (StreamReader, StringReader). Байтовые потоки BinaryReader, BinaryWriter, предназначение, особенности. Байтовая сериализация/десериализация.

10 блок. Событийное программирование.
Делегаты, основные принципы, область применения, передача в методы, операции с делегатами. События, основные принципы событийного программирования порядок работы с событиями.

11 блок. Программирование под Windows.
Основные этапы визуального программирования. Компоненты, типы компонент. Стандартные события, делегаты, синтаксис обработчиков стандартных событий, классы объектов параметров событий. Окна, типы окон. Классы Form, Application. Работа с графическими примитивами, класс Graphics.

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Самарский государственный аэрокосмический университет
имени академика С.П. Королёва (национальный исследовательский университет)» (СГАУ)

Факультет информатики
Кафедра программных систем

Дегтярева О.А.

Электронные тесты по дисциплине «Основы программирования»

Направление 010300.62 Фундаментальная информатика и информационные технологии

Курс 1
Семестр 1,2

Самара 2012

Самарский государственный аэрокосмический университет имени академика С.П. Королева
(национальный исследовательский университет)
Факультет информатики
Кафедра программных систем

Основы программирования

Направление 010300.62 «Фундаментальная информатика и информационные технологии»

Вариант 1.

1. Укажите, в каком описании констант нет ошибки:

1. const int a = 10, b = 20;
   const int c = a + b;
2. const int a = 10; const double b = 20;
   const int c = a + b;
3. const int a = 10, b = 20;
   const uint c = a + b;
4. const float a = 10, b = 20;
   const int c = a + b;

2. Чему будет равна переменная x после вызова метода Sum в классе Class2?

```
class Class1
{
   public static void Sum (int x, int y)
   {
      x +=y;
   }
}
```

```
class Class2
{   public static void Main ()
   {
      int x = 5;
      int y = 8;
      Class1.Sum (x, y);
   }}
```

1. 13 2. 5 3. 8 4. 40

3. Определить, в каком из вариантов правильно описана конструкция программирования:

1. For (int i = 0; i < 6; i++)
   {…}
2. for (int i = 0; i < 6; i++) do
   {…}
3. for (int i = 0; i < 6; i++)
   {…}
4. for int i := 1 to 10
   {…}

4. Чему равно b[1]?

```
{
   int[] a = new int[5];
   for (int i = 0; i<=4; i++)
      a[i] = i*2;
   int[] b = a;
   for (int i = 0; i<=4; i++)
      a[i] = i*4;
}
```

1. 2 2. 4 3. 8 4. 16

5. Сколько возвращаемых значений имеет конструктор класса?

1. 1 2. 2 3.3 4. Ни одного

6. Сколько индексаторов может быть у класса, в котором содержится два поля, являющихся массивами?

1. Ни одного 2. Один 3. Два

7. Чем отличаются классы String и StringBuilder?

1. Первый из них есть в языке, а второго нет.
2. Второй предназначен для изменения текстовых данных без порождения нового экземпляра
3. Первый предназначен для изменения текстовых данных без порождения нового экземпляра
4. Первый порождает особые объекты, второй – массивы символов

8. Что такое «распространение исключения»?

1. Генерирование исключения во всех частях программы, в которых может возникнуть ошибка.
2. Генерирование исключений одинакового типа в различных блоках исполняемой программы.
3. Передача исключения на более высокий уровень, если не найден соответствующий обработчик.

9. Продолжите предложение. Статический элемент класса

1. не содержит реализации
2. вызывается по имени класса
3. предполагает обращение с помощью свойства
4. нельзя вызывать в других классах

10. Какой оператор применяется для прекращения текущего витка цикла и перехода к новому витку?

1. return 2. goto 3. break 4. continue

11. Каким образом возможен доступ к полю родительского класса из дочернего класса, если в дочернем классе есть поле, совпадающее по имени с полем родительского класса?

1. С помощью ключевого слова new
2. С помощью ключевого слова this
3. С помощью ключевого слова base
4. С помощью ключевого слова override

12. Перегрузка метода – это

1. Объявление нескольких методов класса с одним и тем же именем, но с различными сигнатурами
2. Замещение версии метода, объявленной в базовом классе, новой, с точно такой же сигнатурой
3. Вызов метода родительского класса в классе-наследнике
4. Исключение, возникающее из-за слишком большого количества кода в методе

13. Какие данные представляют собой потоки StreamReader, StreamWriter?

1. Записей. 2. Бинарные. 3. Текстовые.

14. Чем отличаются классы FileInfo и File?

1. Оба предназначены для работы с файлом через статические методы.
2. Оба предназначены для работы с классом через методы экземпляра.
3. FileInfo - для работы с файлом через статические методы, а File - для работы с файлом через методы экземпляра.
4. FileInfo - для работы с файлом через методы экземпляра, а File - для работы с файлом через статические методы.

15. Что предоставляет пользователю свойство Capasity коллекции ArrayList?

1. Текущее число элементов в коллекции.
2. Максимальное число элементов, возможное хранить в коллекции ArraytList вообще.
3. Максимальное число элементов, возможное хранить в коллекции ArraytList на данном этапе работы программы.

Самарский государственный аэрокосмический университет имени академика С.П. Королева
(национальный исследовательский университет)
Факультет информатики
Кафедра программных систем

Основы программирования

Направление 010300.62 «Фундаментальная информатика и информационные технологии»

Вариант 2.

1. Укажите, в каком описании констант нет ошибки:

1. const float a = 10, b = 20;
   const int c = a + b;
2. const int a = 10; const double b = 20;
   const int c = a + b;
3. const int a = 10, b = 20;
   const byte c = a + b;
4. const byte a = 10, b = 20;
   const int c = a + b;

2. Определить, в каком из вариантов правильно описана конструкция программирования:

1. while (int i = 0; i < 6; i++)
   {…}
2. double i = -12.5;
   while (i < 0)
   {…}
3. double i = -12.5;
   while (i < 0) do
   {…}
4. int i = 0;
   while (int i <= 10) {…}

3. Что объявлено в следующем операторе:

int[]a;

1. массив элементов целого типа
2. ссылка на массив элементов целого типа
3. ступенчатый массив
4. массив элементов целого типа с длиной 0

4. Чему равно b[2]?

```
{
   double[] a = new double[4];
   for (int i = 0; i<=3; i++)
      a[i] = i+2;
   double[] b = a;
   for (int i = 0; i<=3; i++)
      a[i] += i;
}
```

1. 3 2. 4 3. 5 4. 6

5. Что называется «конструктором по умолчанию»?

1. Конструктор без параметров
2. Конструктор, вызываемый системой
3. Конструктор, в котором всем значимым полям присваивается 0, а ссылочным – null
4. Конструктор, в который в качестве параметров передаются 0 или null

6. Какие параметры метода описываются с помощью ключевого слова ref?

1. Параметры-значения 3. Параметры-ссылки 2. Выходные параметры 4. Параметры-массивы

7. Может ли у свойства класса быть параметр?

1. Да 2. Нет 3. Да, но только один 4. Да, но только целого типа

8. Могут ли блоки get и set свойства класса иметь собственные спецификаторы доступа?

1. Да 2. Нет 3. Да, не уже, чем у всего свойства 4. Да, не шире, чем у всего свойства

9. Чем в языке C# является строка?

1. Массивом символов
2. Массивом кодов Unicode
3. Экземпляром класса String

10. Выберите правильную конструкцию:

```
1. try                                  2. try                                  3. try
{…}                                     {…}                                     {…}
catch (DivideBeZeroException) {…}       catch {…}                               catch (DivideBeZeroException) {…}
catch {…}                               catch (DivideBeZeroException) {…}       catch (OverFlowException) {…}
catch (OverFlowException) {…}           catch (OverFlowException) {…}           catch {…}
```

11. Что такое виртуальный метод?

1. Метод без реализации 2. Метод без параметров 3. Метод, который возможно переопределить в потомках

12. Интерфейс:

1. Может расширять только один интерфейс
2. Должен расширять хотя бы один интерфейс
3. Может расширять несколько интерфейсов
4. Может расширять несколько интерфейсов, но все они должны реализовываться в общем классе

13. Перечисление – это:

1. Особый класс, содержащий конечный набор констант объектного типа
2. Особый класс, содержащий конечный набор констант целого типа
3. Особый класс, содержащий бесконечный набор именованных констант
4. Особый класс, содержащий конечный набор связанных между собой именованных констант,

14. Что такое десериализация?

1. Просмотр у программы серийного номера.
2. Преобразование объекта в последовательность байтов.
3. Преобразование последовательности байтов в объект.
4. Преобразование объекта в последовательность символов Unicode.
5. Преобразование последовательности символов Unicode в объект.

15. Что будет, если при работе метода из списка делегата возникло исключение?

1. Программа завершится с ошибкой.
2. Исключение обрабатывается, и методы из списка делегата выполняются дальше.
3. Исключение обрабатывается, а методы из списка делегата прекращают выполняться.
4. Ничего не будет.

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Государственное образовательное учреждение
высшего профессионального образования
"Самарский государственный аэрокосмический университет
имени академика С.П. Королёва
(национальный исследовательский университет)"

СОГЛАСОВАНО	**УТВЕРЖДАЮ**
Управление образовательных программ	Проректор по учебной работе
_____/А.В. Дорошин	_____/Ф.В. Гречников
"_____"_____ 20___ г.	"____" _____ 20___ г.

РАБОЧАЯ ПРОГРАММА

Наименование модуля (дисциплины)

Основы программирования

Цикл, в рамках которого происходит освоение модуля (дисциплины)	Б.3.Б.4		
Часть цикла	Базовая		
Код учебного плана	010300.2.62-2011-О-П-4г00м		
Факультет	6		
Кафедра	Программных систем		
Курс	1		
Семестр	1, 2		
Лекции (СЛ)	72		
Семинарские и практические занятия (СП)	18		
Лабораторные занятия (СЛР)	72	Экзамен	2
Курсовая работа (проект) (СКР)	0	Зачет	1
Самостоятельная работа (СРС)	198		
Всего	360		

Наименование стандарта, на основании которого составлена рабочая программа:

010300.62 Фундаментальная информатика и информацонные технологии

Соответствие содержания рабочей программы, условий ее реализации, материально-технической и учебно-методической обеспеченности учебного процеса по дисциплине всем требованиям государственных стандартов подтверждаем.

Составители:

Дегтярева Ольга Александровна, доц., к.т.н. _____ (подпись)

Заведующий кафедрой:

Коварцев Александр Николаевич проф., д.т.н. _____ (подпись)

Рабочая програма обсуждена на заседании кафедры

Программных систем

Протокол №___ от "____" _____ 20___ г.

1 Цели и задачи модуля (дисциплины), требования к уровню освоения содержания

1.1 Перечень развиваемых компетенций

ОК-11, ОК-12
ПК-1, ПК-18, ПК-19, ПК-27

1.2 Цели и задачи изучения модуля (дисциплины)

1. Создание у студентов базовой теоретической подготовки в области объектно-ориентированного программирования как одной из наиболее перспективных технологий программирования, а также технологий визуального и событийного программирования.
2. Формирование у студентов понимания основных методик создания объектно-ориентированных приложений, схем обмена данными, механизмов обработки ошибок.
3. Усвоение основных принципов и технологий создания приложений на языках высокого уровня.
4. Выработка у студентов приемов и навыков решения конкретных задач с помощью изученных технологий, помогающих студентам в дальнейшем решать инженерные задачи, связанные с проектированием и программированием.
5. Ознакомление студентов с современными информационными технологиями и выработка у студентов навыков применения изученного материала при решении практических задач.

1.3 Требования к уровню подготовки студента, завершившего изучение данного модуля (дисциплины)

Студенты, завершившие изучение данной дисциплины, должны
знать: базовые принципы разработки объектно-ориентированных приложений на языках высокого уровня, принципы создания визуальных приложений, многопоточных приложений, схемы потоков данных, используемых в объектно-ориентированных приложениях,
правила обработки исключительных ситуаций;
уметь: с помощью изученных средств и технологий программирования разрабатывать приложения для решения учебных и научно-исследовательских задач, имеющих простой и удобный пользовательский интерфейс,
а также проводить отладку и тестирование разработанного программного обеспечения.

1.4 Связь с предшествующими модулями (дисциплинами)

Курс основывается на представлениях и навыках, полученных в процессе изучения следующих дисциплин:
1) информатика
2) информационные технологии

1.5 Связь с последующими модулями (дисциплинами)

Курс является базовым для всех курсов, использующих автоматизированные и

программные методы моделирования, анализа и расчетов, используется в научно-исследовательской работе а также при подготовке выпускной квалификационной работы бакалавра.

2 Содержание рабочей программы (модуля)

Семестр 1		
СЛ 0,2 36 часов 1 кредитов	Активные 0,4	4. Состав класса в языке C#. Конструкторы, поля, методы, специальные методы – свойства, индексаторы, операции класса, индексаторы, деструкторы, вложенные классы, метод Main().
		5. Стандартные классы. Массивы – одномерные, двумерные, ступенчатые, массивы объектов. Работа с массивами. Класс Array. Строки и символы, классы Char, String, StringBuilder. Работа со строками. Класс Random. Класс Object.
		8. Особые классы. Структуры, особенности описания и хранения, работа со структурами. Перечисления, описание, базовый тип перечисления, операции с перечислениями.
		9. Коллекции, особенности, предназначение, типы коллекций.
	Интерактивные 0,2	6. Исключительные ситуации. Исключения, отлов, выброс исключений. Правила обработки исключений. Класс Exception. Пользовательские исключения.
		7. Реализация принципов ООП в C#. Инкапсуляция, спецификаторы доступа. Наследование полей, конструкторов, методов. Реализация полиморфизма – виртуальные методы, абстрактные методы и абстрактные классы, бесплодные классы, модель «включение-делегирование».
	Традиционные 0,4	1. Платформа .NET. Среда Microsoft Visual Studio. Общие принципы функционирования платформы .NET, CLR, CTS, сборка мусора, сборки, схема трансляции программ. Преимущества и недостатки платформы .NET. Среда разработки MS Visual Studio.
		2. Синтаксис языка C#. Лексемы, идентификаторы, константы, правила именования. Типы данных, значимые и ссылочные типы данных, преобразование типов. Операции, операторы, конструкции программирования. Простейший ввод и вывод, форматный вывод, класс Console.

		3. Объектно-ориентированное программирование. Принципы ООП – инкапсуляция, наследование, полиморфизм. Общий состав класса – конструкторы, поля, методы. Перегрузка, сокрытие, переопределение элементов класса. Спецификаторы, спецификаторы доступа.
СП 0,1 18 часов 0,5 кредитов	Активные 1	Практическая работа 1. Основы работы в среде Visual Studio. Настройка среды. Создание приложений.
		Практическая работа 2. Обработка массивов.
		Практическая работа 3. Работа с коллекциями.
	Интерактивные 0	
	Традиционные 0	
СЛР 0,2 36 часов 1 кредитов	Активные 0,4	Лабораторная работа 2. Простейшие классы. Описание класса, доступ к полям и методам, вызов методов, создание объектов.
		Лабораторная работа 3. Обработка строк.
		Лабораторная работа 6. Работа со структурами и перечислениями.
	Интерактивные 0,5	Лабораторная работа 4. Исключительные ситуации. Обработка ошибок. Создание пользовательских исключений.
		Лабораторная работа 5. Создание иерархий классов. Виртуальные и абстрактные методы. Интерфейсы.
	Традиционные 0,1	Лабораторная работа 1. Операторы и конструкции языка C#. Создание простейших приложений.
СКР 0 0 часов 0 кредитов	Активные 0	
	Интерактивные 0	
	Традиционные 0	
СРС 0,5 90 часов 2,5 кредитов	Активные 0,3	Подготовка к лабораторной работе 2.
		Подготовка к практической работе 2.
		Подготовка к практической работе 3.
		Подготовка к лабораторной работе 6.
	Интерактивные 0,4	Подготовка к лабораторной работе 4.
		Подготовка к лабораторной работе 5.
	Традиционные 0,3	Подготовка к лабораторной работе 1.

		Подготовка к практической работе 1.
		Подготовка к лабораторной работе 3.
Семестр 2		
СЛ 0,2 36 часов 1 кредитов	Активные 0,4	10. Работа с объектами файловой системы. Классы Directory и DirectoryInfo, особенности и отличия, классы File и FileInfo, особенности и отличия.
		11. Схема потоков данных. Основные типы потоков данных, иерархия потоков данных в C#. Байтовые потоки, текстовые потоки записи/чтения. Байтовые потоки-обертки. Байтовая сериализация/десериализация.
	Интерактивные 0,6	12. Событийное программирование. Делегаты, основные принципы, область применения, передача в методы, операции с делегатами. События, основные принципы событийного программирования порядок работы с событиями.
		13. Программирование под Windows. Основные этапы визуального программирования. Компоненты, типы компонент. Стандартные события, делегаты, синтаксис обработчиков стандартных событий, классы объектов параметров событий. Окна, типы окон. Классы Form, Applica
	Традиционные 0	
СП 0 0 часов 0 кредитов	Активные 0	
	Интерактивные 0	
	Традиционные 0	
СЛР 0,2 36 часов 1 кредитов	Активные 0,5	Лабораторная работа 7. Потоки данных. Текстовые и байтовые потоки. Сериализация.
		Лабораторная работа 8. Создание делегатов и событий. Вызов делегата. Обработка событий.
	Интерактивные 0,5	Лабораторная работа 9. Создание Windows-приложения для решения учебной расчетной задачи моделирования с использованием основных принципов объектно-ориентированного и визуального программирования.
	Традиционные 0	

СКР 0 0 часов 0 кредитов	Активные 0	
	Интерактивные 1	
	Традиционные 0	
СРС 0,6 108 часов 3 кредитов	Активные 0,5	Подготовка к лабораторной работе 7.
		Подготовка к лабораторной работе 8.
	Интерактивные 0,5	Подготовка к лабораторной работе 9.
		Подготовка к экзамену.
	Традиционные 0	

3 Инновационные методы обучения

1. Использование порталов дистанционного образования.
2. Проведение лекций с использованием современных мультимедийных демонстрационных средств.
3. Выполнение лабораторных работ с помощью современного программного обеспечения.
4. Выполнение практических работ с помощью современного программного обеспечения.
5. Отчет по лабораторным и практическим работам в виде круглых столов.

4 Технические средства и материальное обеспечение учебного процесса

1. Презентационные материалы в виде мультимедийных презентаций.
2. Аппаратура для демонстрации мультимедийных презентаций.
3. Компьютерный класс, используемый при проведении лабораторных и практических занятий.
4. Компьютерное программное обеспечение, необходимое для проведения лабораторных занятий.

5 Учебно-методическое обеспечение

5.1 Основная литература

1. Давыдов, Виктор Григорьевич.Программирование и основы алгоритмизации [Текст] : [учеб. пособие для вузов по специальности "Упр. и информатика в техн. системах"] / В. Г. Давыдов. - М. : Высш. шк., 2003. - 448 с.ГРНТИ:50.01УДК:004.42(075)Экземпляров всего:8
2. Окулов, Станислав Михайлович.Программирование в алгоритмах [Текст] / С. Окулов. - М. : Бином. Лаб. знаний , 2004. - 341 с.ГРНТИ:50.01УДК:004.021(075)Экземпляров всего:3
3. Биллиг, Владимир Арнольдович.Основы программирования на C# [Текст] : учеб. пособие / В. А. Биллиг. - М. : Интернет-ун-т информ. технологий : Бином. Лаб. знаний , 2006. - 483 с.ГРНТИ:50.05.09УДК:004.43(075)Экземпляров всего:15
4. Павловская, Татьяна Александровна.C++. Объектно-ориентированное программирование [Текст] : практикум : [учеб. пособие для вузов по направлению

подгот. дипломир. специалистов "Информатика и вычисл. техника"] / Т. А. Павловская, Ю. А. Щупак. - СПб. [и др.] : Питер : Питер принт, 2006. -

264 с. - (Учебное пособие).ГРНТИ:50.01УДК:004.652(075)Экземпляров всего:8
5. Просиз, Джефф. Программирование для Microsoft. NET [Текст] : [пер. с англ.] / Джефф Просиз. - М. : Рус. ред., 2003. - 678 с. + 1 эл. опт. диск (CD-ROM). - (Фундаментальные знания). ГРНТИ:50.05.09 УДК:004.43 Экземпляров всего:5

5.2 Дополнительная литература

1. Павловская, Татьяна Александровна.C/C++. Программирование на языке высокого уровня [Текст] : Учеб. для вузов по направлению "Информатика и вычисл. техника" / Т. А. Павловская. - СПб.и др. : Питер, 2002. - 460 с.ГРНТИ:50.01УДК:681.3УДК:681.3.06(075)Экземпляров всего:3

2. Павловская, Татьяна Александровна.C/C++. Структурное программирование [Текст] : практикум / Т. А. Павловская, Ю. А. Щупак. - СПб. : Питер : Питер принт, 2004. - 238 с. - (Учебное пособие).ГРНТИ:50.01УДК:004УДК:004.43(075)Экземпляров всего:4
3. Понамарев, Вячеслав Александрович.Программирование на C++/C# в Visual Studio .NET 2003 [Текст] / В. А. Понамарев. - СПб. : БХВ-Петербург, 2004. - 349 с. - (Мастер программ).ГРНТИ:50.05.09УДК:004.43Экземпляров всего:5
4. Круглински, Дэвид Джордж.Программирование на Microsoft Visual C++ 6.0 [Текст] / Д. Д. Круглински, С. Уингоу, Д. Шеферд = Programming Microsoft Visual C++ 6.0 / David J. Kruglinski, Scot Wingoaand, Goorge Shepherd : Пер. с англ. - СПб. [и др.] : Питер : Питер бук ; М. : Русская

редакция, 2001. - 853 с. + 1 эл. опт. диск (CD-ROM). - (Для профессионалов).ГРНТИ:50.05.09УДК:681.3.06:519Экземпляров всего:1

5.3 Электоронные источники и интернет ресурсы

1. www.microsoft.com/rus/net
2. Круглински, Дэвид Джордж.Программирование на Microsoft Visual C++ 6.0 [Электронный ресурс. Компакт-диск]/ Д. Д. Круглински, С. Уингоу, Д. Шеферд. - СПб. : Питер, 2001. - 1 CD-ROM. Шифр 681.3/К 840ГРНТИ:50.05.09
3. Просиз, Джефф.Программирование для Microsoft. NET [Электронный ресурс. Компакт-диск]: [пер. с англ.]/ Джефф Просиз. - М. : Рус. ред., 2003. - 1 CD-ROM. Шифр 004/П 823УДК:681.3.06:519

5.4 Методические указания и рекомендации

Текущий контроль знаний студентов в первом семестре проводится в форме отчета по лабораторным и практическим работам, включающем отчет по теории и работающую программу.
Обсуждение результатов работы производится в фоме круглого стола с обсуждением технологий, испольуемых в работе, а также исправлением возможных ошибок.

По итогам выполнения лабораторных и практических работ принимается решение о допуске или недопуске студента к зачету.
Текущий контроль знаний студентов во втором семестре проводится в форме отчета по лабораторным работам. По итогам выполнения лабораторных работ принимается решение о допуске или недопуске студента к экзамену.
Невыполненная лабораторная или практическая работа является основанием для недопуска студента к зачету в первом семестре.
Неудовлетворительный отчет по теории при наличии работающей программы не лишает студента права сдавать зачет, но может быть основанием для дополнительного вопроса или задачи на зачете.
Невыполненная лабораторная работа является основанием для недопуска студента к экзамену во втором семестре.
Неудовлетворительный отчет по теории при наличии работающей программы не лишает студента права сдавать экзамен, но может быть основанием для дополнительного вопроса или задачи.
Зачет и экзамен проводятся согласно положению о текущем и промежуточном контроле знаний студентов, утвержденному ректором университета.
Зачет/незачет в первом семестре ставится на основании письменного и устного ответов студента по билету. Билет включает два теоретических вопроса и задачу.
Оценка за экзамен во втором семестре ставится на основании письменного и устного ответов студента по билету. Билет включает два теоретических вопроса и задачу.