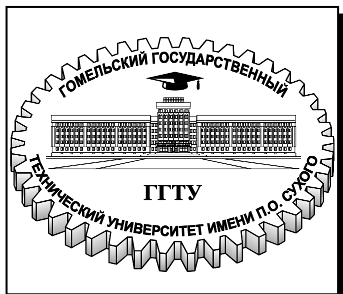

Министерство образования Республики Беларусь

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

Кафедра «Информационные технологии»

Т. Л. Романькова, Е. В. Коробейникова

КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ

**ПОСОБИЕ
по одноименному курсу
для студентов технических специальностей
дневной формы обучения**

Электронный аналог печатного издания

Гомель 2010

Библиотека ГГТУ им. П.О. Сухого

УДК 004.45+004.3(075.8)
ББК 32.973.26-018.1я73
Р69

Рекомендовано к изданию научно-методическим советом факультета автоматизированных и информационных систем ГГТУ им. П. О. Сухого (*протокол № 5 от 30.06.2008 г.*)

Рецензент: нач. сектора программных средств АСУ вычислительного центра ГГТУ им. П. О. Сухого *Н. С. Шестакова*

Романькова, Т. Л.

Р69 Конструирование программ и языки программирования : пособие по одноим. курсу для студентов техн. специальностей днев. формы обучения / Т. Л. Романькова, Е. В. Коробейникова. – Гомель : ГГТУ им. П. О. Сухого, 2010. – 43 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: http://lib.gstu.local. – Загл. с титул. экрана.

ISBN 978-985-420-888-6.

Состоит из семи разделов. Каждый раздел включает теоретические сведения и примеры. В теоретических сведениях дана методика решения задач, указаны необходимые методы языка C#. В практической части описан пошаговый алгоритм решения задач, приведены примеры в виде документов языка программирования.

Для студентов технических специальностей дневной формы обучения.

УДК 004.45+004.3(075.8)
ББК 32.973.26-018.1я73

ISBN 978-985-420-888-6

© Романькова Т. Л., Коробейникова Е. В., 2010
© Учреждение образования «Гомельский государственный технический университет имени П. О. Сухого», 2009

Библиотека ГГТУ им. П. О. Сухого

ВВЕДЕНИЕ

Объектно-ориентированное программирование (ООП) основано на «трех китах» – трех важнейших принципах, придающих объектам новые свойства. Этими принципами являются инкапсуляция, наследование и полиморфизм.

Инкапсуляция есть объединение в единое целое данных и алгоритмов обработки этих данных. В рамках ООП данные называются полями объекта, а алгоритмы – объектными методами. Инкапсуляция позволяет в максимальной степени изолировать объект от внешнего окружения. Она существенно повышает надежность разрабатываемых программ, т. к. локализованные в объекте алгоритмы обмениваются с программой сравнительно небольшими объемами данных, причем количество и тип этих данных обычно тщательно контролируются. В результате замена или модификация алгоритмов и данных, инкапсулированных в объект, как правило, не влечет за собой плохо прослеживаемых последствий для программы в целом (в целях повышения защищенности программ в ООП почти не используются глобальные переменные). Другим немаловажным следствием инкапсуляции является легкость обмена объектами, переноса их из одной программы в другую.

Наследование есть свойство объектов порождать своих потомков. Объект-потомок автоматически наследует от родителя все поля и методы, может дополнять объекты новыми полями и заменять (перекрывать) методы родителя или дополнять их. Принцип наследования решает проблему модификации свойств объекта и придает ООП в целом исключительную гибкость. При работе с объектами программист обычно подбирает объект, наиболее близкий по своим свойствам для решения конкретной задачи, и создает одного или нескольких потомков от него, которые «умеют» делать то, что не реализовано в родителе. Последовательное проведение в жизнь принципа «наследуй и изменяй» хорошо согласуется с поэтапным подходом к разработке крупных программных проектов и во многом стимулирует такой подход.

Полиморфизм – это свойство родственных объектов (т. е. объектов, имеющих одного общего родителя) решать схожие по смыслу проблемы разными способами. В рамках ООП поведенческие свойства объекта определяются набором входящих в него методов. Изменяя алгоритм того или иного метода в потомках объекта, программист может придавать этим потомкам отсутствующие у родителя специфические свойства. Для изменения метода необходимо перекрыть его

Библиотека ГГТУ им. П.О.Сухого

в потомке, т. е. объявить в потомке одноименный метод и реализовать в нем нужные действия. В результате в объекте-родителе и объекте-потомке будут действовать два одноименных метода, имеющие разную алгоритмическую основу и, следовательно, придающие объектам разные свойства. Это и называется *полиморфизмом объектов*.

C# относительно новый язык программирования, который характеризуется следующими преимуществами:

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

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

Центральной частью каркаса Net является его общеязыковая исполняющая среда, известная как Common Language Runtime (CLR), или .NET runtime. Код, исполняемый под управлением CLR, часто называется *управляемым кодом*. Однако перед тем как код сможет исполняться CLR, любой исходный текст должен быть скомпилирован. Компиляция в .NET состоит из 2 шагов:

1. Компиляция исходного кода в IL.
2. Компиляция IL в специфичный для платформы код с помощью CLR.

Наибольшим преимуществом управляемого кода является то, что вы получаете возможность использовать библиотеку базовых классов .NET. Базовые классы .NET представляют огромную коллекцию классов управляемого кода, которые позволяют решать практически любые задачи, которые раньше можно было решать с помощью Windows API. Язык C# можно использовать для создания консольных приложений, Windows приложений, Web-приложений.

1. СОЗДАНИЕ КОНСОЛЬНОГО ПРИЛОЖЕНИЯ НА C#

1.1. Запуск Visual Studio и создание проекта

Все приложения, разрабатываемые в Visual Studio, организуются в проекты. В проекте содержатся исходные и исполняемые файлы, а также другие части приложения. Для начала нужно загрузить Visual Studio. Как только Visual Studio запустится, на экране появится стартовая страница (рис. 1).

Библиотека ГГТУ им. П.О. Сухого

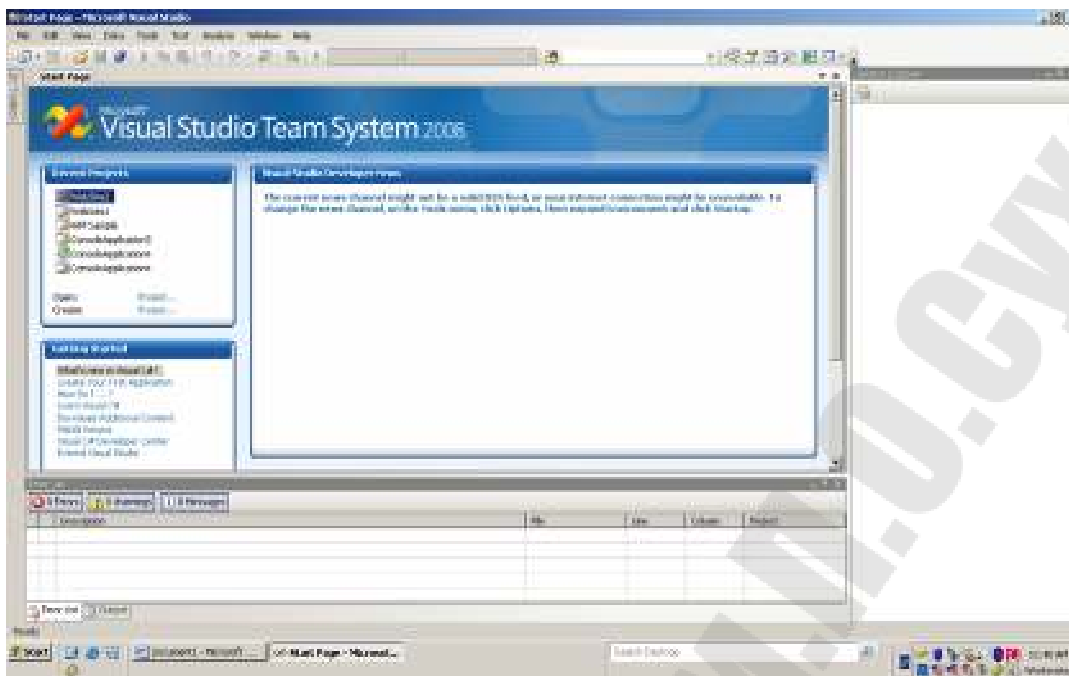

Рис. 1. Внешний вид стартовой страницы при запуске

На стартовой странице отображены все созданные проекты. Можно открыть или создать проект, щелкнув на ссылках Open Project (Открыть проект) и Create Project (Создать проект).

При создании нового проекта Visual Studio отображает диалоговое окно, в котором можно выбрать тип создаваемого файла. В этом окне вводится имя проекта и каталог, в котором будут находится файлы задаваемого проекта.

Для создания консольного приложения в окне Project types нужно выбрать раздел Visual C#, а в окне шаблонов – шаблон Console Application. В поле Name можно ввести, например, MyConsoleApplication, а в поле Location указать папку, в которой проект будет сохранен (рис. 2).

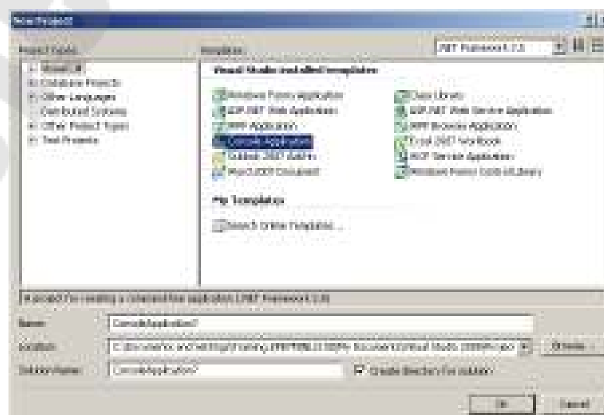

Рис. 2. Диалоговое окно New Project

Библиотека БГТУ им. П.О. Сухого

После того как проект создан, будет отображено основное пространство разработчика (рис. 3).

Это пространство – та среда, в которой происходит дальнейшая разработка проекта. Среда изначально создает некий код, который является скелетом программы (рис. 3). Этот код используется как начало собственной программы.

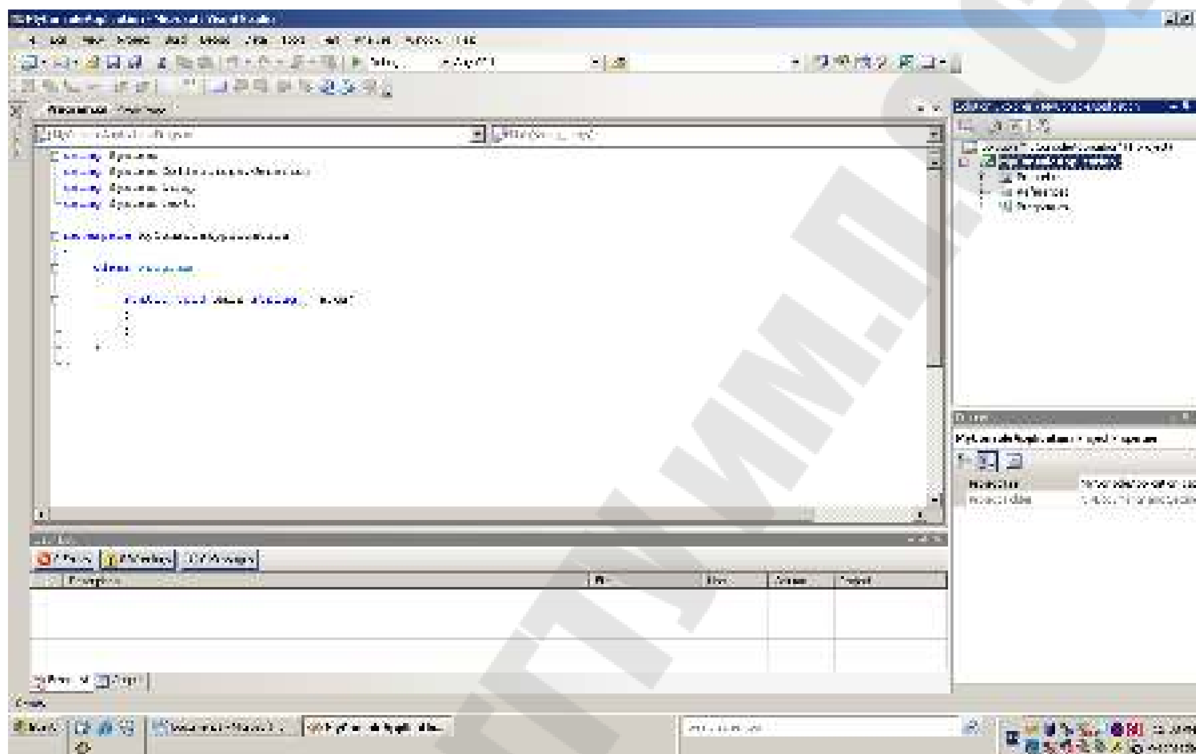

Рис. 3. Среда разработки Visual Studio

Интегрированная среда разработки IDE (Integrated Development Envirionment) Visual Studio является многооконной, настраиваемой, обладает большим набором возможностей. В окне Solution Explorer представлена структура построенного решения. В окне Properties можно увидеть свойства выбранного элемента решения. В окне документов отображается выбранный документ, в данном случае, программный код класса **проекта** – MyConsoleApplication.Program. В этом окне можно отображать и другие документы, список которых показан в верхней части окна. Построенное решение содержит, естественно, единственный заданный нами проект – MyConsoleApplication. Класс проекта погружен в пространство имен, имеющее по умолчанию то же имя, что и проект. Итак, при создании нового проекта автоматически создается достаточно сложная вложенная структура – решение, содержащее проект, содержащий пространство имен, содержащее

Библиотека ГГТУ им.П.О.Сухого

класс, содержащий точку входа (процедура Main()). Пространству имен может предшествовать одно или несколько предложений using, где после ключевого слова следует название пространства имен – из библиотеки FCL или из проектов, связанных с текущим проектом.

Каркас платформы .NET Framework располагает большим набором полезных функций. Каждая из них является членом какого-либо класса. Классы группируются по пространствам имен. Это означает, что в общем случае имя класса может иметь сложную структуру – состоять из последовательности имен, разделенных между собой точками. Последнее имя в этой последовательности собственно и является именем класса. Классы, имена которых различаются лишь последними членами (собственно именами классов) последовательностей, считаются

1.2. Компиляция и выполнение проекта

Перед тем как выполнить программу ее необходимо откомпилировать. Поскольку программа в Visual Studio организована в проект, нужно откомпилировать весь проект. Этот процесс называется *сборкой проекта*. Для того чтобы собрать проект, нужно выбрать команду Build – >Build Solution (Сборка – >Собрать решение). При этом исходный файл Program.cs будет скомпилирован в исполняемый.

После этого программу можно запустить. Для этого из меню Debuge – >Start Withaut Debugging (Отладка – >Запустить в обычном режиме). При запуске программы Visual Studio открывает для нее новое окно консоли (рис. 4). Чтобы завершить программу, нажмите в окне консоли любую клавишу.

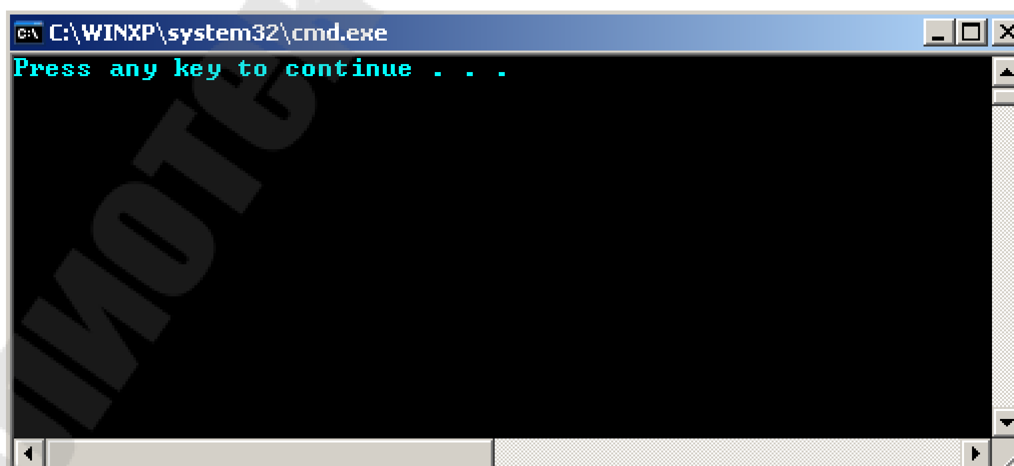

Рис. 4. Запущенная программа

2. ТИПЫ ДАННЫХ

2.1. Переменные и константы

Синтаксис объявления переменных выглядит следующим образом:
DataType Name;

Пример
int i;
float r;

Компилятор не позволит вам использовать переменную, пока она не будет инициализирована значением, но объявление выделяет в стеке 4 байта для хранения в них значений. После того как переменная объявлена, ей можно присваивать значения, используя операцию =.

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

const int a=100;

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

2.2. Категории типов данных

В языке C# различают две категории типов:

1. Типы значений.
2. Ссылочные типы.

Концептуальная разница состоит в том, что тип значений хранит данные непосредственно, а ссылочный тип хранит ссылку на значение.

Эти типы хранятся в разных местах памяти: типы значений сохраняются в области известной как стек, а ссылочные типы – в области, называемой *управляемой кучей*.

Пример

```
// i и j  оба имеют тип int
i=20;
j=20;
```

Эти операторы в результате создадут две области памяти, каждая из которых будет хранить значение 20.

Предположим, что вы определили класс с именем Matrica.

```
Matrica X,Y;
X=new Matrica();
```


Библиотека ГГТУ им. П.О. Сухого

X.Value=30; // Value – поле определенное в классе в Matrix.

Y=X;

Y.Value=50;

После выполнения этого кода будет существовать один объект типа Matrica, а *X* и *Y* будут указывать на область памяти где хранится этот объект.

Если переменная является ссылкой и она ни на что не указывает, то ее значение равно null.

Типы CTS

Базовые предопределенные типы описанные в C# не являются внутренними к языку, но являются частью платформы .NET Framework. Когда вы объявляете **int** в C#, то на самом деле объявляется экземпляр структуры .NET-System.Int32. Это означает то, что вы можете трактовать все примитивные типы так, как если бы они были классами, поддерживающими определенные методы. Например, чтобы преобразовать **int** в **string**, можно записать:

String s = i.ToString();

В C# имеется 15 предопределенных типов, 13 типов значений и 2 ссылочных типа (string, object).

Предопределенные типы значений

Встроенные типы представляют примитивы, такие как целые числа и числа с плавающей запятой, символьные и булевские типы.

Язык C# поддерживает восемь предопределенных целочисленных типов, перечисленных в табл. 1.

Таблица 1

Целые типы данных

Имя	Тип CTS	Описание	Диапазон
sbyte	System.Sbyte	8-битное целое значение	–128:127
short	System.Int16	16-битное целое значение	–32768:32767
int	System.Int32	32-битное целое значение	–2147483648:2147483647
long	System.Int64	64-битное целое значение	–9223372036854775808: 9223372036854775807
byte	System.Byte	8-битное целое без знака	0:255

Библиотека ГГТУ им.П.О.Сухого

Окончание табл. 1

Имя	Тип CTS	Описание	Диапазон
ushort	System.Unit16	16-битное целое без знака	0:65535
uint	System.Unit32	32-битное целое без знака	0:4294967295
ulong	System.Unit34	64-битное целое без знака	0:18446744073709551615

Язык C# поддерживает три типа с плавающей точкой, приведенные в табл. 2.

Таблица 2

Вещественные типы данных

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

Тип bool в C# используется для хранения булевских значений – **true** или **false** (табл. 3).

Таблица 3

Логический тип данных

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

Для хранения одиночных символов C# поддерживает тип данных **char** (табл. 4).

Таблица 4

Символический тип данных

Имя	Тип CTS	Значение
Char	System.Char	Представляет отдельный 16-битовый символ

Библиотека ГГТУ им. П.О.Сухого

Предопределенные ссылочные типы

Язык C# содержит два предопределенных ссылочных типа (табл. 5).

Таблица 5

Предопределенные ссылочные типы

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

Тип **Object** – первичный родительский тип, от которого наследуются все внутренние и пользовательские типы. Тип Object используется для двух целей:

1. Можно использовать ссылку на **Object** для связи с объектам любого конкретного подтипа.
2. Тип **Object** реализует множество базовых методов общего назначения.

Строки – это последовательность символов Unicode. Строки – это на самом деле объекты, содержащие множество методов работы со строками.

2.3. Приведение типов

Приведение типов – один из аспектов безопасности языка. Используемые в программе типы характеризуются собственными диапазонами значений, которые определяются свойствами типов, в том числе и размером области памяти, предназначенной для кодирования значений соответствующего типа. При этом области значений различных типов пересекаются. Многие значения можно выразить более чем одним типом. Например, значение 4 можно представить как значение типа sbyte, byte, short, ushort, int, uint, long, ulong. При этом в программе все должно быть устроено таким образом, чтобы логика преобразования значений одного типа к другому типу была бы понятной, а результаты этих преобразований – предсказуемы. В одном выражении могут быть сгруппированы операнды различных типов. Однако возможность подобного «смешения» при определении значения выражения приводит к необходимости применения дополнительных усилий по приведению значений операндов к «общему типу». Иногда приведение значения к другому типу происходит автоматически. Та-

Библиотека БГТУ им. В.Г. Шухова

кие преобразования называются *неявными*. Но в ряде случаев преобразование требует дополнительного внимания со стороны программиста, который должен явным образом указывать необходимость преобразования, используя выражения приведения типа или обращаясь к специальным методам преобразования, определенным в классе System.Convert, которые обеспечивают преобразование значения одного типа к значению другого (в том числе значения строкового типа к значениям базовых типов).

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

Различаются:

1. *Расширяющее преобразование* – значение одного типа преобразуется к значению другого типа, которое имеет такой же или больший размер. Например, значение, представленное в виде 32-разрядного целого числа со знаком, может быть преобразовано в 64-разрядное целое число со знаком. Расширяющее преобразование считается безопасным, поскольку исходная информация при таком преобразовании не искажается. Некоторые расширяющие преобразования типа могут привести к потере точности. На рис. 5 описаны варианты преобразований, которые иногда приводят к потере информации.

Int32	Float
UInt32	Float
Int64	Float, Double
UInt64	Float, Double
Decimal	Float, Double

Рис. 5. Варианты преобразований

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

CLR разрешает привести тип объекта к его собственному типу или любому из его базовых типов. Для приведения типа к производ-

Библиотека ГГТУ им.П.О.Сухого

ному от него типу необходимо ввести операцию явного приведения типов – неявное преобразование приведет к ошибке.

Пример

```
class Employee {
. . .
}
class App {
public static void Main( ) {
Object o = new Employee( );
Employee e = (Employee) o;
}
}
```

Если попытаться привести к типу объект, который относится к типу, не являющимся нашим типом или производным от него, то *CLR* не выполняет приведение типов и генерирует исключение *System.InvalidCastException*.

Приведение типов с помощью операторов is и as

Оператор *is* проверяет совместимость объекта с данным типом, а в качестве результата выдает значение типа *Boolean*: *true* или *false*. Этот оператор никогда не генерирует исключение. Если ссылка на объект равна *null*, оператор *is* всегда возвращает *false*, так как нет объекта, для которого нужно определить тип.

Приведением типов занимается оператор *as*. Он отличается от приведения типа только тем, что никогда не генерирует исключение. Если приведение типа невозможно, результатом является *null*.

Если не сравнить полученный оператором результат с *null* и попытаться работать с пустой ссылкой, возникнет исключение *NullReferenceException*.

Пример использования операторов *is* и *as*:

```
if (o is Student) { Student s = (Student) o; . . . }
Student s = o as Student;
if (s ! = null) { . . . }
```

Второй код эффективнее, т. к. в первом случае *CLR* проверяет совместимость с типом *Student* дважды: в операторе *is* и в теле оператора *if* (при приведении типа). Во втором же случае *CLR* проверяет совместимость o с типом *Student* только раз, а *if* лишь сравнивает *s* с *null* – такая проверка намного эффективнее, чем определение типа объекта.

Библиотека ГГТУ им.П.О.Сухого

3. ПРОСТЕЙШИЙ ВВОД-ВЫВОД

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

Console.WriteLine (параметр);

Пример

Console.WriteLine («Здравствуйте, я ваша тетя!»);

или

Console.Write («Здравствуйте, я ваша тетя!»);

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

Пример

```
int x = 3;
Console.WriteLine("x="+x.);
```

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

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

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

Пример

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

Для каждого параметра можно задать ширину поля вывода.

Пример

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

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

Библиотека ГГТУ им.П.О.Сухого

Пример

```
Console.Write("Как вас зовут?\n");
string   name = Console.ReadLine();
```

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

```
Console.Write("Введите символ\n");
char  name = (char) Console.Read();
```

Ввод чисел с клавиатуры можно осуществить в два этапа:

- ввести число в виде строки символов;
- преобразовать строку в переменную соответствующего типа.

Преобразование выполняется с помощью класса **Convert** из пространства имен System, используя соответствующие методы класса **ToInt32(s)**, **ToDouble(s)**, **ToByte(s)** и т. д.

Здесь s – параметр типа string.

Пример

```
string s;
s = Console.ReadLine();
byte h = Convert.ToByte(s);
```

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

Пример

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

Аналогично выполняется преобразование к типам int, double и т. д.

4. КЛАССЫ, ОСНОВНЫЕ ЭЛЕМЕНТЫ КЛАССА

4.1. Основные понятия и описание классов

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

В класс могут входить описанные ниже элементы.

- ***константы***, которые хранят неизменяемые значения, связанные с классом;
- ***поля***, содержащие данные класса;
- ***методы***, реализующие действия, выполняемые классом или экземпляром;

Библиотека ГГТУ им. П.О.Сухого

- ***свойства***, которые определяют характеристики класса в совокупности со способами их задания и получения (методами записи и чтения);
- ***конструкторы***, выполняющие действия по инициализации экземпляров или класса в целом;
- ***деструкторы*** определяют действия, выполняемые перед тем, как объект будет уничтожен;
- ***индексаторы*** обеспечивают возможность доступа к элементам класса по их порядковому номеру;
- ***операции*** задают действия с объектами с помощью знаков операций;
- ***события***, на которые может реагировать объект, определяют уведомления, которые может генерировать класс;
- ***типы*** – внутренние по отношению к классу типы данных.

Класс описывается следующим образом:

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

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

public – доступ не ограничен;

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

internal – доступ только из данной программы;

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

Для не вложенных классов используются только спецификаторы **public** и **internal**.

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

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

Например, пусть имеется следующее описание класса:

class Primer1{}

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

Библиотека ГГТУ им.П.О.Сухого

Primer1 p1 = new   Primer1();

Primer1   p2 = new Primer1();

Программа на C# состоит из взаимодействующих между собой классов. В каждом приложении обязательно должен присутствовать класс, содержащий метод **Main**. Именно с этого метода начинается выполнение программы.

4.2. Поля и константы класса

Данные, инкапсулированные в классе, представлены полями и константами.

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

Синтаксис описания элемента данных:

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

Спецификаторы полей и констант:

public – доступ к элементу не ограничен;

protected – доступ только из данного и производных классов;

internal – доступ только из данной сборки;

private – доступ только из данного класса;

static – одно поле для всех экземпляров класса;

readonly – поле доступно только для чтения.

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

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

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

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

Обращение к статическому полю класса:

Имя_класса. имя_поля

Обращение к константе класса:

Имя_класса. имя_константы

Обращение к обычному полю (полю экземпляра) класса:

Имя_объекта. имя_поля

Библиотека ГГТУ им. П.О.Сухого

Например, пусть в приложении описан класс Primer1:

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

Обратиться к полям этого класса можно следующим образом:

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

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

0
2,5
0
2,5
1
5
A
пустая строка

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

Библиотека ГГТУ им. П.О. Сухого

4.3. Методы

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

Синтаксис метода:

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

Спецификаторы метода:

public – доступ не ограничен;
protected – доступ только из данного и производных классов;
internal – доступ только из данной сборки;
private – доступ только из данного класса (по умолчанию);
static – статический метод.

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

Тип в заголовке метода определяет тип результата работы метода.

Для передачи значения выражения в качестве результата работы метода используется оператор

return ***выражение*;**

Пример

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

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

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


Библиотека ГГТУ им. П.О.Сухого

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

Пример

```
static  int   maximum(int  x, int  y)            // метод, возвращающий
                                                 // максимальное из двух целых чисел
{
   if  (x > y)  return x;
    else        return y;
}
```

Обращение к статическому методу класса:

***Имя_класса. имя_метода*([*аргументы*])**

Обращение к нестатическому методу класса:

***Имя_объекта. имя_метода*([*аргументы*])**

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

Пример

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

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

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

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

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

или

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

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

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


Библиотека ГГТУ им. П.О.Сухого

В C# предусмотрены следующие виды параметров:
- параметры-значения;
- параметры-ссылки;
- выходные параметры;
- параметры-массивы.

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

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

Пример

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

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

Библиотека ГГТУ им. П.О.Сухого

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

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

Пример

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

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

Пример

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


Библиотека ГГТУ им.П.О.Сухого

4.4. Конструкторы

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

Описание:

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

Обычно используется спецификатор public.

Конструктор не возвращает значение.

Класс может иметь несколько конструкторов для инициализации объектов разными способами.

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

Для инициализации статических данных класса можно создать статический конструктор.

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

Описание статического конструктора:

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

Приведенный на рис. 6 пример демонстрирует создание объектов класса с помощью различных конструкторов.

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

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

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

Рис. 6. Пример программы для создания объектов класса (1-й фрагмент; продолжение и окончание см. на с. 24 и 25)

Библиотека ГГТУ им. П.О.Сухого

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

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

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

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

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

 class Program
     {

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

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

Рис. 6. Продолжение
(начало см. на с. 23, окончание – на с. 25)

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

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

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

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

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

Рис. 6. Окончание (начало см. на с. 23 и 24)

4.5. Сбор мусора. Деструкторы

Каждому объекту класса при создании выделяется память в динамической области памяти (*хипе*). В C# имеется система сбора мусора, которая автоматически возвращает память для повторного исполь-

зования. Эта система действует незаметно для программиста, активизируется только по необходимости и точно невозможно узнать, когда происходит сбор мусора.

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

Синтаксис деструктора:

~ *Имя_класса* ( )
{ *тело деструктора*}

4.6. Свойства класса

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

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

Оба аксессора **не могут отсутствовать**.

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

При обращении к свойству автоматически вызываются аксессоры чтения и установки.

Обращение к свойству:

имя_объекта.имя_свойства

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

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

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

Библиотека ГГТУ им. П.О. Сухого

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

Обращение к свойству может быть такое:

```
Console.WriteLine("Радиус= {0,5:f3}   ", C1.Radius);
// В этом случае выполняется аксессор get
C3.Radius = 100;
// В этом случае выполняется аксессор set
```

Аксессор set можно дополнить проверкой значения на положительность:

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

5. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

5.1. Понятие исключения, основные стандартные исключения

Все исключения являются подклассами класса **Exception** пространства имен System. Исключения генерирует среда программирования или программист. В табл. 6 представлены наиболее часто используемые исключения.

Таблица 6

Стандартные исключения

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

Библиотека ГГТУ им. П.О. Сухого

Свойства класса Exception:
Message – текстовое описание ошибки;
TargetSite – метод, выбросивший исключение.

5.2. Оператор try

Исключения перехватываются и обрабатываются оператором **try**.

try
{*контролируемый блок*}
catch (*тип1* [*имя1*]) { *обработчик исключения1* }
catch (*тип2* [*имя2*]) { *обработчик исключения2* }
. . .
catch { *обработчик исключения* }
finally {*блок завершения*}

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

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

При возникновении ошибки при выполнении операторов контролируемого блока генерируется исключение.

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

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

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

Форма обработчика

catch (*тип*) { *обработчик исключения* }

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

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


Библиотека ГГТУ им.П.О.Сухого

Форма обработчика

catch (*тип имя*) { *обработчик исключения* }

используется, когда имя параметра используется в теле обработчика.

Пример

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

При попытке деления на 0 выведется сообщение:

Attempted to divide by zero. Деление на 0

Форма обработчика

catch { обработчик исключения }

применяется для перехвата всех исключений, независимо от их типа. ***Он может быть только один в операторе try и должен быть помещен после остальных catch-блоков.***

Пример

```
try
     { int v = Convert.ToInt32(Console.ReadLine()); }
catch { Console.WriteLine("Ошибка!!!"); }
```

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

5.3. Генерирование исключений вручную

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

Формат ее записи таков:

throw [*параметр*];

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

Пример

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

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

Библиотека ГГТУ им. П.О. Сухого

throw ;

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

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

     }

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

Рис. 7. Пример генерирования исключения вручную

Один try-блок можно вложить в другой.

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

6. ИСПОЛЬЗОВАНИЕ МАССИВОВ В C#

6.1. Основные понятия

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

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

При создании всем элементам массива присваиваются по умолчанию нулевые значения.

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

Библиотека ГГТУ им. П.О.Сухого

6.2. Одномерный массив

Одномерный массив можно описать одним из описанных ниже способов.

***тип*[ ] *имя_массива*;**

Например,

double[ ] y, z;

В этом случае память под элементы массивов не выделена.

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

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

Например,

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

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

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

Например,

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

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

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

В этом случае **new** подразумевается.

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

Например,

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

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

Обращение к элементу массива:

***имя массива* [*индекс*]**

Например, x[3], MyArray[10].

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

Библиотека ГГТУ им. П.О.Сухого

```
Console.WriteLine("Введите количество элементов");
int n=Convert.ToInt32(Console.ReadLine());
double[ ] x = new double[n];
for (int  i = 0; i < n; ++i)
{
   Console.Write("x[" + i + "]=");
   x[i] = Convert.ToDouble(Console.ReadLine());
}
```

Для просмотра всех элементов из некоторой группы данных: массива, списка и других существует удобный оператор цикла **foreach**.

Синтаксис:

foreach (*тип имя_переменной* in *имя_массива*)
**   *тело цикла*;**

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

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

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

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

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

6.3. Свойства и методы класса Array для обработки одномерных массивов

Элементы класса **Array** для работы с одномерными массивами представлены в табл. 7–9.

Таблица 7

Свойства класса System.Array

Свойство	Описание
Length	Количество элементов массива
Rank	Количество размерностей массива

Библиотека ГГТУ им.П.О.Сухого

Таблица 8

Статические методы класса System.Array

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

Таблица 9

Нестатические методы класса System.Array

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

Библиотека ГГТУ им. П.О.Сухого

6.4. Многомерные массивы

6.4.1. Виды многомерных массивов

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

Вот как объявляется многомерный прямоугольный массив:

***тип*[, … ,] *имя* = new *тип*[*размер_1*, ..., *размер_N*] ;**

Пример

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

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

6.4.2. Двумерные прямоугольные массивы

Двумерный массив можно описать одним из следующих способов:

***тип*[,]   *имя_массива*;**

Пример

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

В этом случае память под элементы массивов не выделена.

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

Пример,

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

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

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

В списке инициализаторов значения сгруппированы в фигурных скобках по строкам. Например,

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

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

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

Операция **new** подразумевается.

***тип*[,] *имя_массива* = new   *тип*[*разм1*, *разм2*] {*список_инициализаторов*};**

Библиотека ГГТУ им.П.О.Сухого

Например,

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

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

Обращение к элементу матрицы:

***имя_массива* [*индекс1*, *индекс2*]**

Например, x[3,4], MyArray[1,0].

Следующий фрагмент кода демонстрирует вывод матрицы на экран в консольном режиме.

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

Рис. 8. Вывод матрицы

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

Например, пусть требуется найти максимальный элемент матрицы:

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

В табл. 10 приводятся нестатические методы класса **Array** для работы с двумерными массивами.

Таблица 10

Нестатические методы класса Array для работы с двумерными массивами

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

Библиотека ГГТУ им. П. О. Сухого

6.4.3. Ступенчатые массивы

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

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

Описание:

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

Например, int[ ] [ ] z = new[10] [ ];

Здесь *размер* означает количество строк в массиве.

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

Пример

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

В данном случае x.Lenght равно 3, x[0].Lenght равно **4**, x[1].Lenght равно **3**, x[2].Lenght равно **5**.

Другой способ:

***тип*[ ] [ ] *имя* = {*создание_массива1*, *создание_массива2*,…, *создание_массиваN*};**

Например,

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

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

имя[индекс1] [индекс2]

Например, **x[2] [9] или a[i] [j]**

Пример

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

Консольное приложение для решения данной задачи приведено ниже.

Библиотека ГГТУ им.П.О.Сухого

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ConsoleApplication1
{
  class Student
    {
        int[ ] ocenki;            // закрытые поля класса: массив оценок,
        string  fam, gruppa; // фамилия, группа,
        static  double mrot; // минимальны размер оплаты труда.
        public Student(string fam, string gruppa, int n) // конструктор
          { this.fam = fam; this.gruppa = gruppa;
            ocenki = new int[n]; }
public string Fam   // свойство доступа к  полю Фамилия
        {
          set { fam = value; }
          get { return fam; }
        }
public string Gruppa   // свойство доступа к  полю Группа
        {
          set { gruppa = value; }
          get { return gruppa; }
        }
static  Student( ) // статический конструктор класса
        { mrot = 200000; }
 public void vvod_oc() //метод  для вывода оценок
    {
        Console.WriteLine("Введите оценки студента " +
                                       fam + " за сессию");
        for (int i = 0; i < ocenki.Length;++i )
           {ocenki[i]= int.Parse(Console.ReadLine()); }
     }

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


Библиотека ГГТУ им. П.О.Сухого

```
      return S / ocenki.Length;}
  }
public double stip  // свойство только для чтения,
                              // возвращающее размер стипендии
 {
  get { if (Dvoechnik) return 0;
        else return
         (Sr_b > 8) ? (2 * mrot) :
      (Sr_b > 6 ? 1.8 * mrot : 1.25 * mrot);
      }
  }
public bool Dvoechnik  //свойство только для чтения,
                       // возвращающее true, если у студента есть двойки
  { get
      {if (Array.IndexOf(ocenki, 2)>=0)
               return true;
         else
               return false;
        } }}
class Program
   {
      static void Main(string[ ] args)
        {   Console.WriteLine("Сколько групп?");
             int m=int.Parse(Console.ReadLine());
// создание ступенчатого массива объектов класса Student
             Student[ ][ ]   fakultet = new  Student[m][ ];
      for (int i = 0; i < m; ++i)
                {
                  Console.WriteLine("Какая группа?");
                   string grp = Console.ReadLine();
Console.WriteLine("Сколько в группе студентов ?");
                   int ks = Convert.ToInt32(Console.ReadLine());
                  fakultet[i]=new Student[ks];
                   for (int j = 0; j < ks; ++j)
                     {   Console.WriteLine("Фамилия студента ? ");
 fakultet[i][j] = new Student(Console.ReadLine( ), grp, 4);
                        fakultet[i][j].vvod_oc( );
                      }
                   }
```


Библиотека ГГТУ им. П.О. Сухого

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

7. ИНДЕКСАТОРЫ КЛАССА

7.1. Одномерные индексаторы

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

Индексаторы могут характеризоваться одной или несколькими размерностями.

Одномерные индексаторы описываются следующим образом:

Библиотека ГГТУ им. П.О. Сухого

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

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

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

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

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

Например, пусть в классе **Massiv** определены закрытое поле
double[ ] x;
и индексатор

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

И пусть **Y** – объект класса **Massiv**.

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

Y[1] = 5.3;
Console.WriteLine(Y[i]);

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

Библиотека ГГТУ им. П.О. Сухого

7.2. Многомерные индексаторы

Многомерные индексаторы описываются следующим образом:

***спецификатор тип* this [*тип1 индекс1*, *тип2 индекс2*, …, *типN индексN*]**

{

get {*код аксессора для получения данных*}

set {*код аксессора для установки данных*}

}

Например, если в классе объявлен массив

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

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

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


Библиотека ГГТУ им. П.О.Сухого

Литература

1. Павловская, Т. А. C#. Программирование на языке высокого уровня : учеб. для вузов / Т. А. Павловская. – Санкт-Петербург : Питер, 2007. – 432 с.

2. Биллиг, В. А. Основы программирования на C# / В. А. Биллинг. – Москва : Интернет-университет информационных технологий – ИНТУИТ.ру, 2006. – 488 с.

3. Шилдт, Г. С. C# : учеб. курс / Г. С. Шилдт. – Санкт-Петербург : Питер, 2002. – 512 с.

4. Шилдт, Г. С. Полный справочник по C# / Г. С. Шилдт. – Москва : Издат. дом «Вильямс», 2004. – 752 с.

5. Фролов, А. В. Язык C# : самоучитель / А. В. Фролов. – Москва : Диалог-МИФИ, 2003. – 560 с.

Библиотека ГГТУ им. П. О. Сухого

Содержание

Введение 3
1. Создание консольного приложения на C# 4
1.1. Запуск Visual Studio и создание проекта 4
1.2. Компиляция и выполнение проекта 7
2. Типы данных 8
2.1. Переменные и константы 8
2.2. Категории типов данных 8
2.3. Приведение типов 11
3. Простейший ввод-вывод 14
4. Классы, основные элементы класса 15
4.1. Основные понятия и описание классов 15
4.2. Поля и константы класса 17
4.3. Методы 19
4.4. Конструкторы 23
4.5. Сбор мусора. Деструкторы 25
4.6. Свойства класса 26
5. Обработка исключительных ситуаций 27
5.1. Понятие исключения, основные стандартные исключения 27
5.2. Оператор try 28
5.3. Генерирование исключений вручную 29
6. Использование массивов в C# 30
6.1. Основные понятия 30
6.2. Одномерный массив 31
6.3. Свойства и методы класса Array для обработки одномерных массивов 32
6.4. Многомерные массивы 34
7. Индексаторы класса 39
7.1. Одномерные индексаторы 39
7.2. Многомерные индексаторы 41
Литература 42

Библиотека ГГТУ им. П.О.Сухого

Учебное электронное издание комбинированного распространения

Учебное издание

Романькова Татьяна Леонидовна
Коробейникова Евгения Васильевна

КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Пособие по одноименному курсу для студентов технических специальностей дневной формы обучения

Электронный аналог печатного издания

Редактор *М. В. Аникеенко*
Компьютерная верстка *Е. В. Темная*

Подписано в печать 22.12.09.
Формат 60x84/16. Бумага офсетная. Гарнитура «Таймс».
Ризография. Усл. печ. л. 2,56. Уч.-изд. л. 2,7.
Изд. № 114.
E-mail: ic@gstu.gomel.by
http://www.gstu.gomel.by

Издатель и полиграфическое исполнение:
Издательский центр учреждения образования
«Гомельский государственный технический университет
имени П. О. Сухого».
ЛИ № 02330/0549424 от 08.04.2009 г.
246746, г. Гомель, пр. Октября, 48.

Библиотека ГГТУ им.П.О.Сухого