Теория и практика программирования

Шпилёв Пётр Валерьевич

Санкт-Петербургский государственный университет
Математико-механический факультет
Кафедра статистического моделирования

Лекции

Санкт-Петербург
2021 г.

e-mail: pitshp@hotmail.com

Литература


Б. Пахомов. C-C++ и MS Visual C++ 2008 для начинающих, СПб.:БХВ-Петербург, 2009.


Герберт Шилдт. C# 4.0. Полное руководство, М.: ООО "И.Д. Вильямс 2011


Н. Культин. Microsoft Visual C в задачах и примерах, СПб.:БХВ-Петербург, 2009.


Ч. Петцольд. Программирование с использованием MS WindowsForms, М.: Русская Редакция; СПб.: Питер, 2006.


Ч. Петцольд. Программирование для Microsoft Windows на C#., М.: Русская Редакция, 2002


Эндрю Троелсен. Язык программирования C# 2010 и платформа .NET 4, М.:ООО "И.Д. Вильямс 2011.


Симон Робинсон и др. C# для профессионалов, издательство «Лори», 2003

Лекция 1. Содержание

- Предыстория. Создание C#.
- Объектно-ориентированное программирование.
- Примеры.

Лекция 1. Содержание

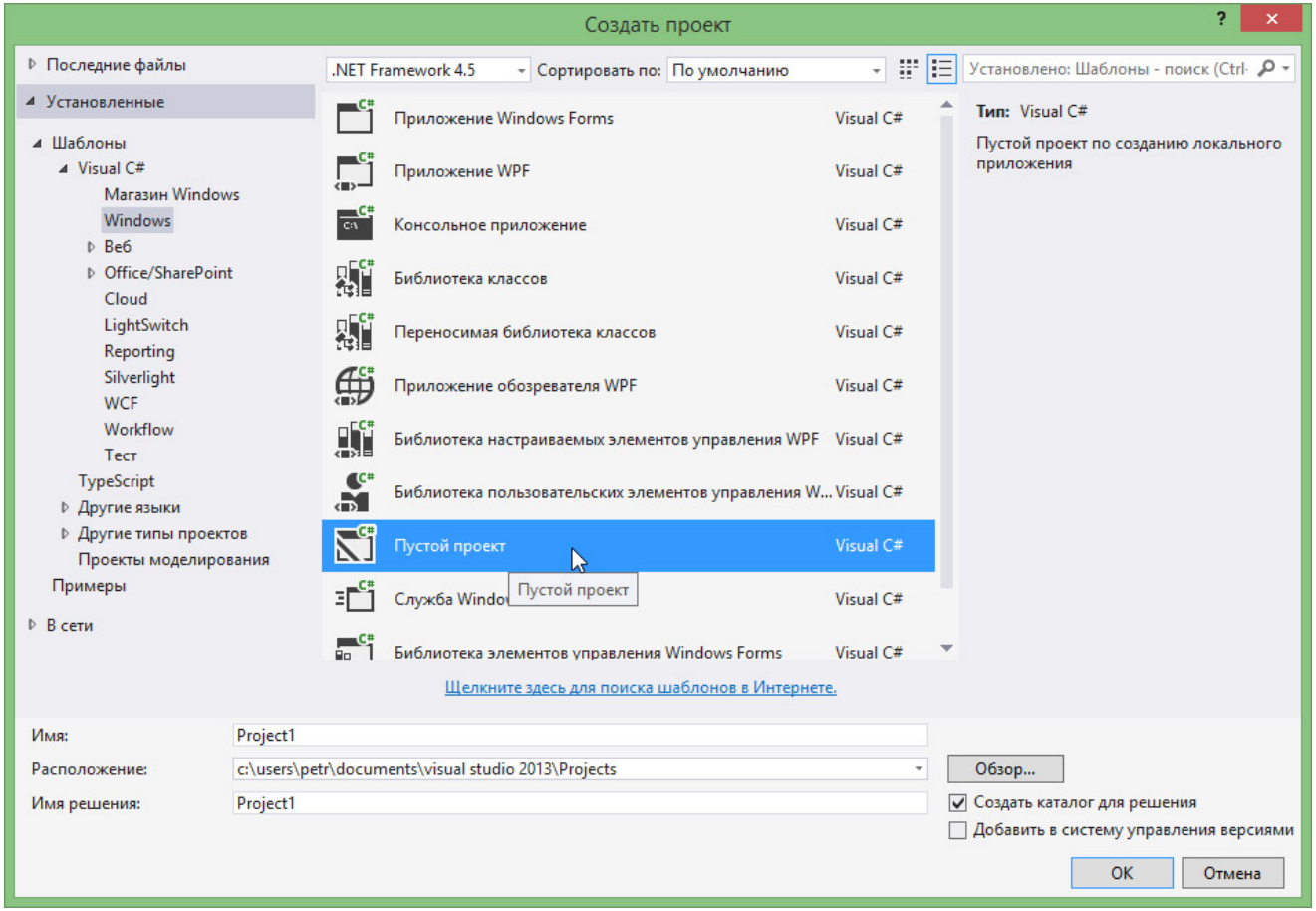
- Предыстория. Создание C#.
- Объектно-ориентированное программирование.
- Примеры.

Лекция 1. Содержание

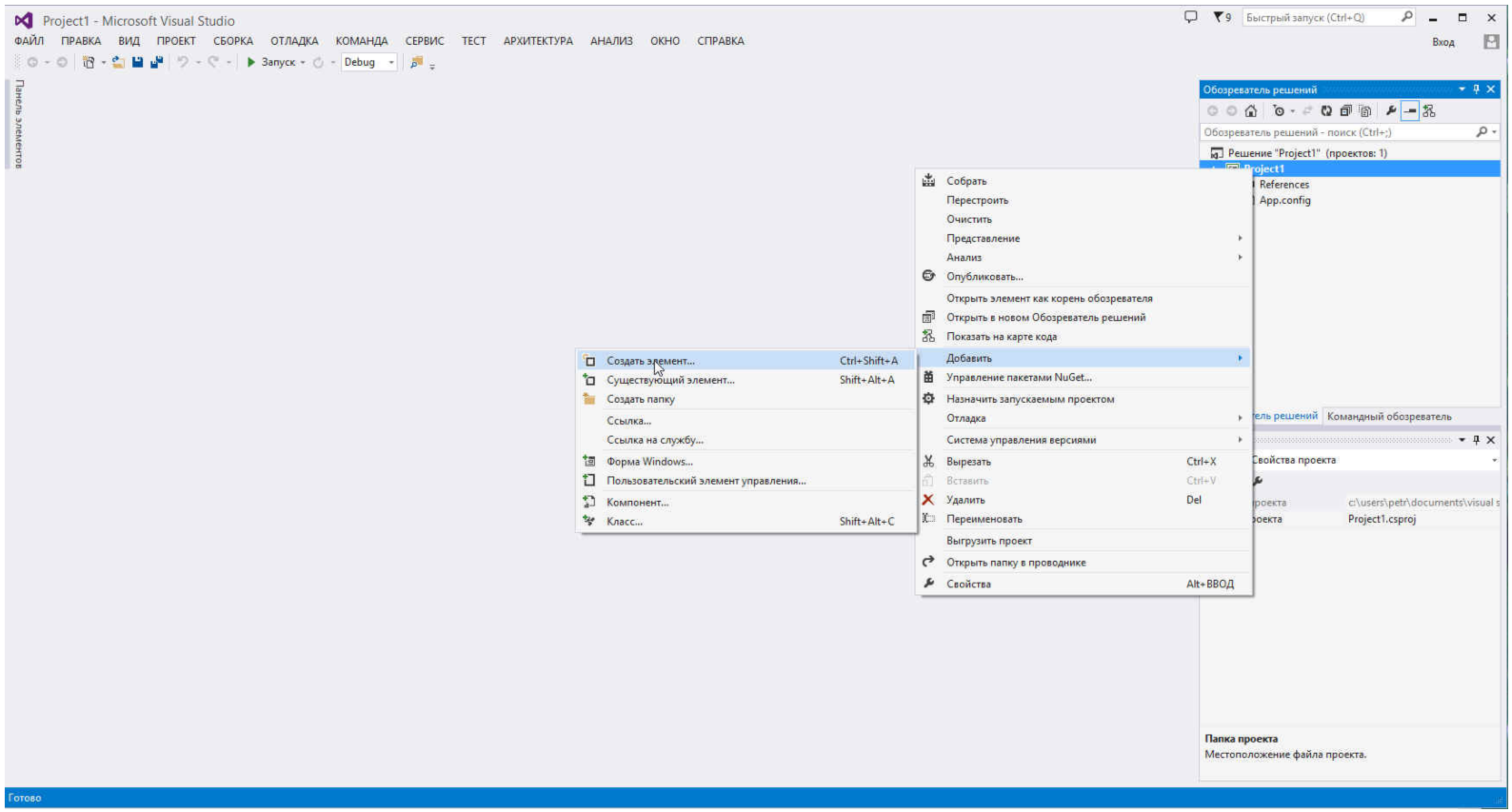
- Предыстория. Создание C#.
- Объектно-ориентированное программирование.
- Примеры.

```
/* Это простая программа на C#.
Назовем ее Example.cs. */
using System;
class Example
{
// Любая программа на C# начинается с вызова метода Main().
    static void Main()
    {
        Console.WriteLine("Простая программа на C#.");
    }
}
```

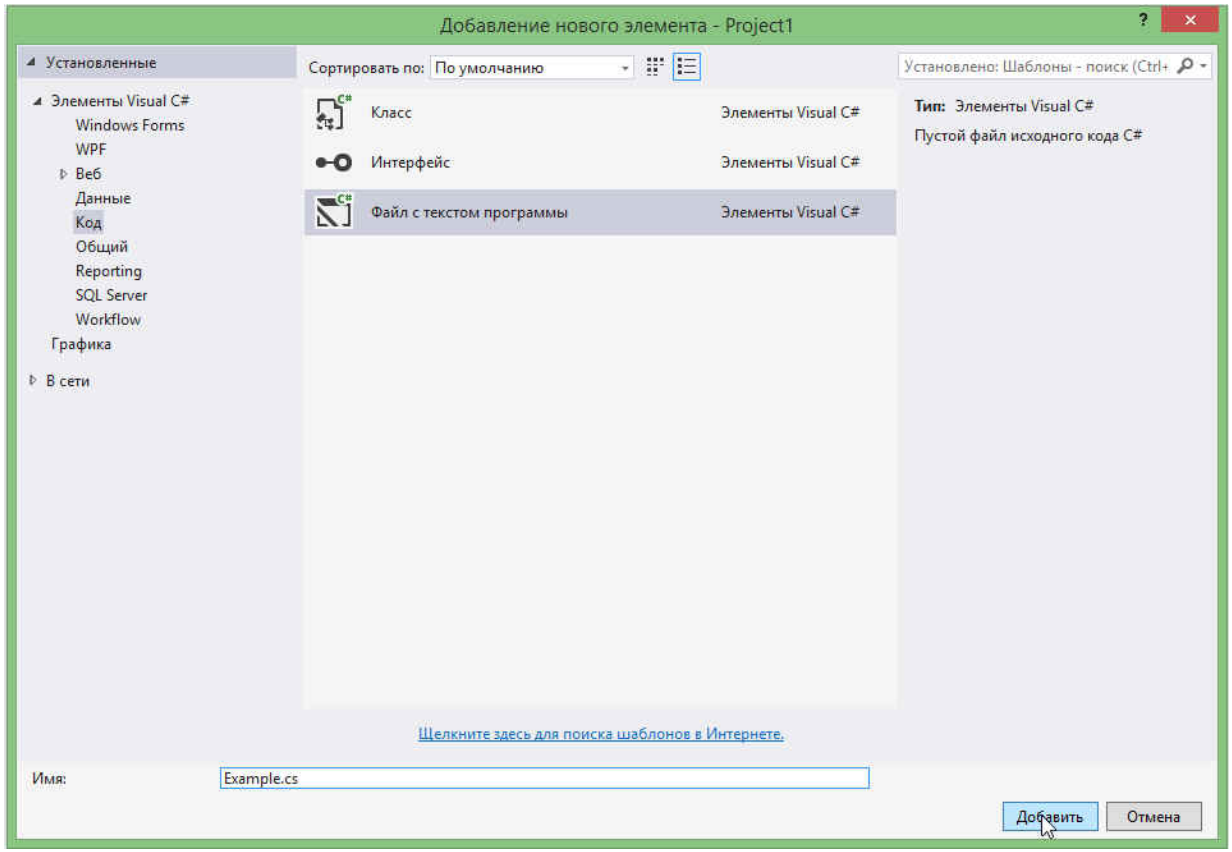
- Создайте новый (пустой) проект C#, выбрав команду Файл>Создать>Проект (File>New>Project). Затем выберите элемент Windows из списка Установленные шаблоны (Installed Templates) и далее — шаблон Пустой проект (Empty Project), как показано на рисунке. Щелкните на кнопке OK, чтобы создать проект.



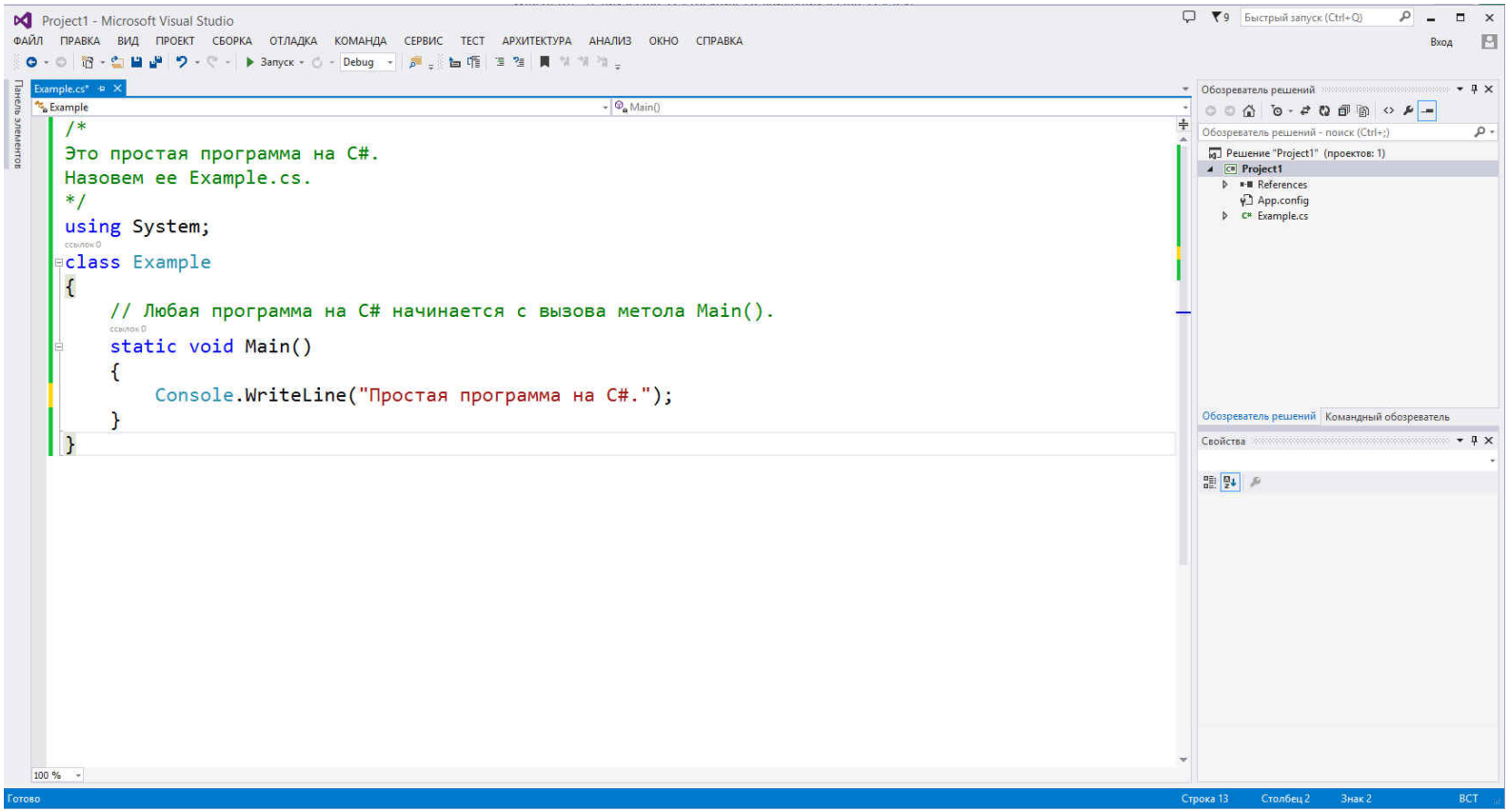
- На данном этапе проект пуст, и вам нужно ввести в него файл с исходным текстом программы на C#. Для этого щелкните правой кнопкой мыши на имени проекта (в данном случае — Project1) в окне Обозреватель решений, а затем выберите команду Добавить (Add) из контекстного меню. В итоге появится подменю, показанное на рисунке.



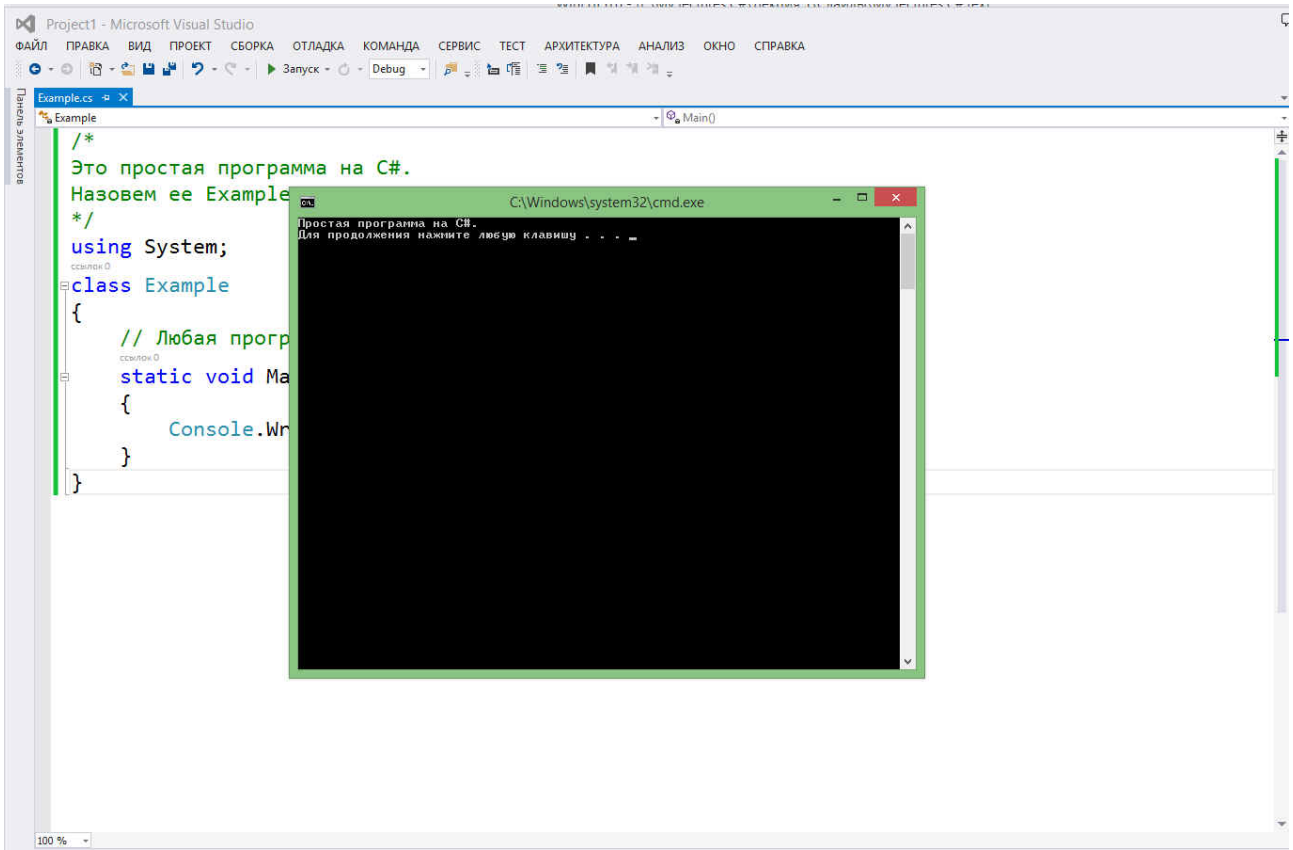
- Выберите команду Создать элемент (New Item), чтобы открыть диалоговое окно Добавление нового элемента (Add New Item). Выберите сначала элемент Код (Code) из списка Установленные шаблоны, а затем шаблон Файл с текстом программы (Code File) и измените имя файла на Example.cs, как показано на рисунке.



- Введите исходный текст программы в окне с меткой Example.cs, после чего сохраните этот текст в файле. По завершении ввода исходного текста программы экран будет выглядеть так, как показано на рисунке.



- Скомпилируйте программу, выбрав команду Построение>Построить решение (Build>Build Solution).
- Выполните программу, выбрав команду Отладка>Запуск без отладки (Debug>Start Without Debugging). В результате выполнения программы откроется окно, показанное на рисунке.



```
/* Это простая программа на C#.
Назовем ее Example.cs. */
using System;
class Example
// Любая программа на C# начинается с вызова метода Main().
static void Main()
Console.WriteLine("Простая программа на C#.");
```

```
/* Это простая программа на C#.
Назовем ее Example.cs. */
using System;
class Example
// Любая программа на C# начинается с вызова метода Main().
static void Main()
Console.WriteLine("Простая программа на C#.");
```

```
/* Это простая программа на C#.
Назовем ее Example.cs. */
using System;
class Example
// Любая программа на C# начинается с вызова метода Main().
static void Main()
Console.WriteLine("Простая программа на C#.");
```

```
/* Это простая программа на C#.
Назовем ее Example.cs. */
using System;
class Example
// Любая программа на C# начинается с вызова метода Main().
static void Main()
Console.WriteLine("Простая программа на C#.");
```

```
/* Это простая программа на C#.
Назовем ее Example.cs. */
using System;
class Example
// Любая программа на C# начинается с вызова метода Main().
static void Main()
Console.WriteLine("Простая программа на C#.");
```

```
/* Это простая программа на C#.
Назовем ее Example.cs. */
using System;
class Example
// Любая программа на C# начинается с вызова метода Main().
static void Main()
Console.WriteLine("Простая программа на C#.");
```

```
// Эта программа демонстрирует применение переменных.
using System;
class Example2
{
    static void Main()
    {
        int x; // здесь объявляется переменная
         int y; // здесь объявляется еще одна переменная
         x = 100; // здесь переменной x присваивается значение 100
         Console.WriteLine("x содержит" + x);
        y = x / 2;
        Console.Write("y содержит x / 2:");
        Console.WriteLine(y);
    }
}
```

Обычно для объявления переменной служит следующий оператор:

- **тип имя_переменной**;

где **тип** — это конкретный тип объявляемой переменной, а **имя_переменной** — имя самой переменной. Помимо типа int, в C# поддерживается ряд других типов данных.

Замечание

В C# внедрено средство, называемое неявно типизированной переменной. Неявно типизированными являются такие переменные, тип которых автоматически определяется компилятором.

```
/* Эта программа демонстрирует отличия между типами данных int и
double. */
using System;
class Example3
{
    static void Main()
    {
        int ivar; // объявить целочисленную переменную
        double dvar; // объявить переменную с плавающей точкой
        ivar = 100; // присвоить переменной ivar значение 100
        dvar = 100.0; // присвоить переменной dvar значение 100.0
         Console.WriteLine("Исходное значение ivar:" + ivar);
         Console.WriteLine("Исходное значение dvar:" + dvar);
         Console.WriteLine(); // вывести пустую строку
        // Разделить значения обеих переменных на 3.
        ivar = ivar / 3;
        dvar = dvar / 3.0;
         Console.WriteLine("Значение ivar после деления:" + ivar);
         Console.WriteLine("Значение dvar после деления:" + dvar);
    }
}
```

- Условный оператор: *if(условие) оператор*;
- Оператор цикла: *for(инициализация; условие; итерация) оператор*;
  Например: for(int i = 0; i < 3; i++) {...}
- Тернарный оператор "?": *Выражение 1 ? Выражение 2 : Выражение 3;*
  используется вместо определенных видов конструкций if-then-else
  Например: abs_val = val < 0 ? -val : val; // абсолютное значение переменной val

Замечание

В C# точка с запятой обозначает конец оператора. Составные элементы оператора можно располагать в отдельных строках. Например, следующий фрагмент кода считается в C# вполне допустимым:

```
Console.WriteLine("Это длинная строка вывода" +
x + y + z +
"дополнительный вывод");
```

- Условный оператор: *if(условие) оператор*;
- Оператор цикла: *for(инициализация; условие; итерация) оператор*;
  Например: for(int i = 0; i < 3; i++) {...}
- Тернарный оператор "?": *Выражение 1 ? Выражение 2 : Выражение 3;*
  используется вместо определенных видов конструкций if-then-else
  Например: abs_val = val < 0 ? -val : val; // абсолютное значение переменной val

Замечание

В C# точка с запятой обозначает конец оператора. Составные элементы оператора можно располагать в отдельных строках. Например, следующий фрагмент кода считается в C# вполне допустимым:

```
Console.WriteLine("Это длинная строка вывода" +
x + y + z +
"дополнительный вывод");
```

- Условный оператор: *if(условие) оператор*;
- Оператор цикла: *for(инициализация; условие; итерация) оператор*;
  Например: for(int i = 0; i < 3; i++) {...}
- Тернарный оператор "?": *Выражение 1 ? Выражение 2 : Выражение 3;*
  используется вместо определенных видов конструкций if-then-else
  Например: abs_val = val < 0 ? -val : val; // абсолютное значение переменной val

Замечание

В C# точка с запятой обозначает конец оператора. Составные элементы оператора можно располагать в отдельных строках. Например, следующий фрагмент кода считается в C# вполне допустимым:

```
Console.WriteLine("Это длинная строка вывода" +
x + y + z +
"дополнительный вывод");
```

Таблица 1: Ключевые слова, зарезервированные в языке C#

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	volatile
void	while			

Таблица 2: Контекстные ключевые слова C#

add	dynamic	from	get	global
group	into	join	let	orderby
partial	remove	select	set	value
var	where	yield		

Замечание

Несмотря на то что зарезервированные ключевые слова нельзя использовать в качестве идентификаторов, в C# разрешается применять ключевое слово с предшествующим знаком @ в качестве допустимого идентификатора. Например, @for — действительный идентификатор.

Лекция 1. Пример применения идентификатора со знаком @.

```
using System;
class IdTest
{
        static void Main()
        {
            int @if; // применение ключевого слова if
            // в качестве идентификатора
            for(@if = 0; @if < 10; @if++)
            Console.WriteLine("@if равно" + @if);
        }
}
```

Таблица 3: Типы значений в C#

Тип	Значение
bool	Логический, предоставляет два значения: “истина” или “ложь”
byte	8-разрядный целочисленный без знака
char	Символьный
decimal	Десятичный (для финансовых расчетов)
double	С плавающей точкой двойной точности
float	С плавающей точкой одинарной точности
int	Целочисленный
long	Длинный целочисленный
sbyte	8-разрядный целочисленный со знаком
short	Короткий целочисленный
uint	Целочисленный без знака
ulong	Длинный целочисленный без знака
ushort	Короткий целочисленный без знака

Замечание

Помимо простых типов, в C# определены еще три категории типов значений: перечисления, структуры и обнуляемые типы.

Таблица 4: Целочисленные типы

Тип	Разрядность в битах	Диапазон представления чисел
byte	8	0-255
sbyte	8	-128-127
short	16	-32 768-32 767
ushort	16	0-65 535
int	32	-2 147 483 648-2 147 483 647
uint	32	0-4 294 967 295
long	64	-9 223 372 036 854 775 808-9 223 372 036 854 775 807
ulong	64	0-18 446 744 073 709 551 615

Замечание

Тип char также является целочисленным.

- Тип decimal предназначен для применения в финансовых расчетах. Этот тип имеет разрядность 128 бит для представления числовых значений в пределах от 1E-28 до 7,9E+28.

```
// Использовать тип decimal для расчета скидки.
using System;
class UseDecimal
{
    static void Main()
    {
        decimal price;
        decimal discount;
        decimal discounted_price;
        // Рассчитать цену со скидкой.
        price = 19.95m;
        discount = 0.15m; // норма скидки составляет 15%
        discounted_price = price - ( price * discount);
         Console.WriteLine("Цена со скидкой:" + discounted_price);
    }
}
```

Упражнение 1.1

Представьте себе, к примеру, что некий богатей желает облагодетельствовать деньгами кафедру статистического моделирования и намерен внести первоначальный взнос в конце года 1. Если процентная ставка равна 10% и если меценат намерен обеспечивать кафедре по 100 тыс. дол. (с учетом 5-% роста заработной платы) ежегодно на бессрочную перспективу, напишите программу для расчета суммы, которую ему нужно отложить для этой цели сегодня.

Упражнение 1.1

Представьте себе, к примеру, что некий богатей желает облагодетельствовать деньгами кафедру статистического моделирования и намерен внести первоначальный взнос в конце года 1. Если процентная ставка равна 10% и если меценат намерен обеспечивать кафедре по 100 тыс. дол. (с учетом 5-% роста заработной платы) ежегодно на бессрочную перспективу, напишите программу для расчета суммы, которую ему нужно отложить для этой цели сегодня.

Упражнение 1.2

Дополните предыдущую программу возможностью расчета аннуитета (актива, который приносит фиксированный доход в течение конечного числа лет).

Упражнение 1.1

Представьте себе, к примеру, что некий богатей желает облагодетельствовать деньгами кафедру статистического моделирования и намерен внести первоначальный взнос в конце года 1. Если процентная ставка равна 10% и если меценат намерен обеспечивать кафедре по 100 тыс. дол. (с учетом 5-% роста заработной платы) ежегодно на бессрочную перспективу, напишите программу для расчета суммы, которую ему нужно отложить для этой цели сегодня.

Упражнение 1.2

Дополните предыдущую программу возможностью расчета аннуитета (актива, который приносит фиксированный доход в течение конечного числа лет).

Упражнение 1.3

Предположим, работники кафедры хотят получать зарплату от мецената не ежегодно, а ежемесячно (1/12 от 100 тыс с учетом 5% ежегодного роста) какова сумма в этом случае?

Упражнение 1.4

С помощью написанной программы выполните следующие расчеты:
а) Новый автомобиль стоит 10 тыс. дол. Если процентная ставка равна 5%, сколько вы должны отложить сейчас, чтобы накопить эту сумму за пять лет?
б) Вы должны платить за обучение по 12 тыс. дол. в конце каждого года в течение следующих шести лет. Если процентная ставка равна 8%, сколько вам нужно отложить сегодня, чтобы покрыть эту сумму?
в) Вы инвестировали 60 476 дол. под 8% годовых. Сколько у вас останется к концу года 6 после внесения платы за обучение, о которой говорилось выше?

Пример 1

```
Console.WriteLine("Вы заказали " + 2 +
"предмета по цене" + 3 + "$ каждый.");
```

Пример 2

```
Console.WriteLine("Деление 10/3 дает:" + 10.0/3.0);
```

Пример 1

```
Console.WriteLine("Вы заказали " + 2 +
"предмета по цене" + 3 + "$ каждый.");
```

Результат: Вы заказали 2 предмета по цене 3$ каждый.

Пример 2

```
Console.WriteLine("Деление 10/3 дает:" + 10.0/3.0);
```

Пример 1

```
Console.WriteLine("Вы заказали " + 2 +
"предмета по цене" + 3 + "$ каждый.");
```

Пример 2

```
Console.WriteLine("Деление 10/3 дает:" + 10.0/3.0);
```

Пример 1

```
Console.WriteLine("Вы заказали " + 2 +
"предмета по цене" + 3 + "$ каждый.");
```

Пример 2

```
Console.WriteLine("Деление 10/3 дает:" + 10.0/3.0);
```

Результат: Деление 10/3 дает: 3.333333333333.

Общий вид

WriteLine("форматирующая строка", arg0, arg1, ... , argN);

форматирующая строка

"форматирующая строка" = Текст {спецификатор формата} текст {спецификатор формата} ...

Общий вид спецификатора формата

{argnum, width: fmt}, где argnum — номер выводимого аргумента, начиная с нуля; width — минимальная ширина поля; fmt — формат.

Общий вид

WriteLine("форматирующая строка", arg0, arg1, ... , argN);

форматирующая строка

"форматирующая строка" = Текст {спецификатор формата} текст {спецификатор формата} ...

Общий вид спецификатора формата

{argnum, width: fmt}, где argnum — номер выводимого аргумента, начиная с нуля; width — минимальная ширина поля; fmt — формат.

Общий вид

WriteLine("форматирующая строка", arg0, arg1, ... , argN);

форматирующая строка

"форматирующая строка" = Текст {спецификатор формата} текст {спецификатор формата} ...

Общий вид спецификатора формата

{argnum, width: fmt}, где argnum — номер выводимого аргумента, начиная с нуля; width — минимальная ширина поля; fmt — формат.

Общий вид

WriteLine("форматирующая строка", arg0, arg1, ... , argN);

форматирующая строка

"форматирующая строка" = Текст {спецификатор формата} текст {спецификатор формата} ...

Общий вид спецификатора формата

{argnum, width: fmt}, где argnum — номер выводимого аргумента, начиная с нуля; width — минимальная ширина поля; fmt — формат.

Пример 1: Console.WriteLine("В феврале {0} или {1} дней.", 28, 29);

Общий вид

WriteLine("форматирующая строка", arg0, arg1, ... , argN);

форматирующая строка

"форматирующая строка" = Текст {спецификатор формата} текст {спецификатор формата} ...

Общий вид спецификатора формата

{argnum, width: fmt}, где argnum — номер выводимого аргумента, начиная с нуля; width — минимальная ширина поля; fmt — формат.

Пример 1: Console.WriteLine("В феврале {0} или {1} дней.", 28, 29);

Результат: В феврале 28 или 29 дней

Общий вид

WriteLine("форматирующая строка", arg0, arg1, ... , argN);

форматирующая строка

"форматирующая строка" = Текст {спецификатор формата} текст {спецификатор формата} ...

Общий вид спецификатора формата

{argnum, width: fmt}, где argnum — номер выводимого аргумента, начиная с нуля; width — минимальная ширина поля; fmt — формат.

Пример 1: Console.WriteLine("В феврале {0} или {1} дней.", 28, 29);

Результат: В феврале 28 или 29 дней

Пример 2: Console.WriteLine("Деление 10/3 дает: { 0:#.##}", 10.0/3.0);

Общий вид

WriteLine("форматирующая строка", arg0, arg1, ... , argN);

форматирующая строка

"форматирующая строка" = Текст {спецификатор формата} текст {спецификатор формата} ...

Общий вид спецификатора формата

{argnum, width: fmt}, где argnum — номер выводимого аргумента, начиная с нуля; width — минимальная ширина поля; fmt — формат.

Пример 1: Console.WriteLine("В феврале {0} или {1} дней.", 28, 29);

Результат: В феврале 28 или 29 дней

Пример 2: Console.WriteLine("Деление 10/3 дает: { 0:#.##}", 10.0/3.0);

Результат: Деление 10/3 дает: 3,33

Таблица 1: Управляющие последовательности символов

Управляющая последовательность	Описание
\a	Звуковой сигнал (звонок)
\b	Возврат на одну позицию
\f	Перевод страницы (переход на новую страницу)
\n	Новая строка (перевод строки)
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\0	Пустой символ
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта

Таблица 1: Управляющие последовательности символов

Управляющая последовательность	Описание
\a	Звуковой сигнал (звонок)
\b	Возврат на одну позицию
\f	Перевод страницы (переход на новую страницу)
\n	Новая строка (перевод строки)
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\0	Пустой символ
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта

Пример: `Console.WriteLine("Первая строка \n Вторая строка");`

Таблица 1: Управляющие последовательности символов

Управляющая последовательность	Описание
\a	Звуковой сигнал (звонок)
\b	Возврат на одну позицию
\f	Перевод страницы (переход на новую страницу)
\n	Новая строка (перевод строки)
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\0	Пустой символ
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта

Пример: Console.WriteLine("Первая строка \n Вторая строка");

Результат: Первая строка
Вторая строка

Литералы

Литералы это постоянные значения, представленные в удобной для восприятия форме.
Например, 12 — это литерал типа int, 12L — литерал типа long, 10.19F — это литерал типа float.

Шестнадцатеричные литералы

Шестнадцатеричные литералы должны начинаться с символов 0x, т.е. нуля и последующей латинской буквы "икс".
Например, number = 0xFF; // 255 в десятичной системе
age = 0x1a; // 26 в десятичной системе.

Строковые литералы

Строковый литерал представляет собой набор символов, заключенных в двойные кавычки.

Буквальный строковый литерал

Буквальный строковый литерал начинается с символа @, после которого следует строка в кавычках.

Литералы

Литералы это постоянные значения, представленные в удобной для восприятия форме.
Например, 12 — это литерал типа int, 12L — литерал типа long, 10.19F — это литерал типа float.

Шестнадцатеричные литералы

Шестнадцатеричные литералы должны начинаться с символов 0x, т.е. нуля и последующей латинской буквы "икс".
Например, number = 0xFF; // 255 в десятичной системе
age = 0x1a; // 26 в десятичной системе.

Строковые литералы

Строковый литерал представляет собой набор символов, заключенных в двойные кавычки.

Буквальный строковый литерал

Буквальный строковый литерал начинается с символа @, после которого следует строка в кавычках.

Литералы

Литералы это постоянные значения, представленные в удобной для восприятия форме.
Например, 12 — это литерал типа int, 12L — литерал типа long, 10.19F — это литерал типа float.

Шестнадцатеричные литералы

Шестнадцатеричные литералы должны начинаться с символов 0x, т.е. нуля и последующей латинской буквы "икс".
Например, number = 0xFF; // 255 в десятичной системе
age = 0x1a; // 26 в десятичной системе.

Строковые литералы

Строковый литерал представляет собой набор символов, заключенных в двойные кавычки.

Буквальный строковый литерал

Буквальный строковый литерал начинается с символа @, после которого следует строка в кавычках.

Литералы

Литералы это постоянные значения, представленные в удобной для восприятия форме.
Например, 12 — это литерал типа int, 12L — литерал типа long, 10.19F — это литерал типа float.

Шестнадцатеричные литералы

Шестнадцатеричные литералы должны начинаться с символов 0x, т.е. нуля и последующей латинской буквы "икс".
Например, number = 0xFF; // 255 в десятичной системе
age = 0x1a; // 26 в десятичной системе.

Строковые литералы

Строковый литерал представляет собой набор символов, заключенных в двойные кавычки.

Буквальный строковый литерал

Буквальный строковый литерал начинается с символа @, после которого следует строка в кавычках.

```
// Продемонстрировать применение буквальных строковых литералов.
using System;
class Verbatim
{
    static void Main()
    {
        Console.WriteLine(@"Это буквальный
         строковый литерал,
         занимающий несколько строк.
         ");
        Console.WriteLine(@"А это вывод с табуляцией:
        1 2 3 4
        5 6 7 8
        ");
         Console.WriteLine(@"Отзыв программиста: ""Мне нравится
C#.""");
    }
}
```

Замечание

Все переменные в C# должны быть объявлены до их применения.

Общая форма задания переменной

тип имя_переменной;

Замечание

Все переменные в C# должны быть объявлены до их применения.

Общая форма задания переменной

тип имя_переменной;

Замечание

Все переменные в C# должны быть объявлены до их применения.

Общая форма задания переменной

тип имя_переменной;

Инициализация переменной:

тип имя_переменной = значение (или некоторое выражение);

Замечание

Все переменные в C# должны быть объявлены до их применения.

Общая форма задания переменной

тип имя_переменной;

Инициализация переменной:

тип имя_переменной = значение (или некоторое выражение);

Неявно типизированная переменная:

Например, var e = 2.7183; // Тип - double

var e = 2.7183F; // Тип - float

```
// Продемонстрировать область действия кодового блока.
using System;
class ScopeDemo
{
  static void Main()
  {
   int x; // Эта переменная доступна для всего кода внутри метода Main().
   x = 10;
   if(x == 10)
   {// начать новую область действия
     int y = 20;// Эта переменная доступна только в данном кодовом
блоке.
    // Здесь доступны обе переменные, x и y.
    Console.WriteLine("x и y:" + x + " " + y);
    x = y * 2;
   }
   // y = 100; // Ошибка! Переменная y здесь недоступна.
   // А переменная x здесь по-прежнему доступна.
   Console.WriteLine("x равно" + x);
  }
}
```

```
/**** Эта программа не может быть скомпилирована. ****/
using System;
class NestVar
{
    static void Main()
    {
        int i= 5;
        for (int count = 0; count < 10; count++)
        {
            Console.WriteLine("Это подсчет:" + count);
            for(int i = 0; i < 2; i++) // Недопустимо!!!
            Console.WriteLine("В этой программе есть ошибка!");
        }
    }
}
```

Автоматическое преобразование типов

Неявное преобразование типов происходит автоматически при следующих условиях:

- оба типа совместимы;
- диапазон представления чисел целевого типа шире, чем у исходного типа.

Автоматическое преобразование типов

Неявное преобразование типов происходит автоматически при следующих условиях:

- оба типа совместимы;
- диапазон представления чисел целевого типа шире, чем у исходного типа.

Пример:

```
int i;
double d;
i = 10;
d = i; // присвоить целое значение переменной типа double
```

Автоматическое преобразование типов

Неявное преобразование типов происходит автоматически при следующих условиях:

- оба типа совместимы;
- диапазон представления чисел целевого типа шире, чем у исходного типа.

Приведение типов

Приведение — это команда компилятору преобразовать результат вычисления выражения в указанный тип.

Автоматическое преобразование типов

Неявное преобразование типов происходит автоматически при следующих условиях:

- оба типа совместимы;
- диапазон представления чисел целевого типа шире, чем у исходного типа.

Приведение типов

Приведение — это команда компилятору преобразовать результат вычисления выражения в указанный тип.

Общая форма приведения типов:

(целевой_тип) выражение

Автоматическое преобразование типов

Неявное преобразование типов происходит автоматически при следующих условиях:

- оба типа совместимы;
- диапазон представления чисел целевого типа шире, чем у исходного типа.

Приведение типов

Приведение — это команда компилятору преобразовать результат вычисления выражения в указанный тип.

Общая форма приведения типов:

(целевой_тип) выражение

Пример:

```
double x=5, y=3;
int i;
i = (int) (x / y);
```

- ЕСЛИ один операнд имеет тип decimal, ТО и второй операнд продвигается к типу decimal (но если второй операнд имеет тип float или double, результат будет ошибочным).
- ЕСЛИ один операнд имеет тип double, ТО и второй операнд продвигается к типу double.
- ЕСЛИ один операнд имеет тип float, ТО и второй операнд продвигается к типу float.
- ЕСЛИ один операнд имеет тип ulong, ТО и второй операнд продвигается к типу ulong (но если второй операнд имеет тип sbyte, short, int или long,результат будет ошибочным).
- ЕСЛИ один операнд имеет тип long, ТО и второй операнд продвигается к типу long.
- ЕСЛИ один операнд имеет тип uint, а второй — тип sbyte, short или int, ТО оба операнда продвигаются к типу long.
- ЕСЛИ один операнд имеет тип uint, ТО и второй операнд продвигается к типу uint.
- ИНАЧЕ оба операнда продвигаются к типу int.

```
// Пример неожиданного результата продвижения типов!
using System;
class PromDemo
{
    static void Main()
    {
        byte b;
        b = 10;
        b = (byte) (b * b); // Необходимо приведение типов!!
        Console.WriteLine("b: "+ b);
    }
}
```

```
// Пример неожиданного результата продвижения типов!
using System;
class PromDemo
{
    static void Main()
    {
        char ch1 = 'a', ch2 = 'b';
        ch1 = (char) (ch1 + ch2);
        Console.WriteLine("ch1: " + ch1);
    }
}
```

```
// Пример неожиданного результата продвижения типов!
using System;
class PromDemo
{
    static void Main()
```

Упражнение 2.1

Написать программу для вычисления целой и дробной части заданного числа.

```
        Console.WriteLine("ch1: " + ch1);

    }
}
```

Таблица 2: Арифметические операторы в C#

Оператор	Действие
+	Сложение
−	Вычитание, унарный минус
*	Умножение
/	Деление
%	Деление по модулю
−−	Декремент
++	Инкремент

```
// Продемонстрировать применение оператора %.
using System;
class ModDemo
{
    static void Main()
    {
        int iresult, irem;
        double dresult, drem;
        iresult = 10 / 3;
        irem = 10 % 3;
        dresult = 10.0 / 3.0;
        drem = 10.0 % 3.0;
        Console.WriteLine("Результат и остаток от деления 10/3:" +
        iresult + " " + irem);
        Console.WriteLine("Результат и остаток от деления 10.0 / 3.0:" +
        dresult + " " + drem);
    }
}
```

Операторы инкремента(++) и декремента(−−)

Оба оператора инкремента и декремента можно указывать до операнда (в префиксной форме) или же после операнда (в постфиксной форме)

Операторы инкремента(++) и декремента(−−)

Оба оператора инкремента и декремента можно указывать до операнда (в префиксной форме) или же после операнда (в постфиксной форме)

Пример:

```
++x; // префиксная форма
x++; // постфиксная форма
```

```
// Продемонстрировать отличие между префиксной
// и постфиксной формами оператора инкремента (++).
using System;
class PrePostDemo
{
    static void Main()
    {
        int x, y;
        int i;
        x = 1;
        y = 0;
        Console.WriteLine("Ряд чисел, полученных" +
        "с помощью оператора y = y + x++;");
        for(i = 0; i < 10; i++)
        {
          y = y + x++; // постфиксная форма оператора ++
          Console.WriteLine(y + " ");
        }
        Console.WriteLine();
```

```
        x = 1;
        y = 0;
        Console.WriteLine( "Ряд чисел, полученных" +
        "с помощью оператора y = y + ++x;");
        for(i = 0; i < 10; i++)
        {
          y = y + ++x; // префиксная форма оператора ++
          Console.WriteLine(y + " ");
        }
        Console.WriteLine();
    }
}
```

Таблица 3: Операторы отношения в C#

Оператор	Значение
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Таблица 4: Логические операторы в C#

Оператор	Значение
&	И
|	ИЛИ
^	Исключающее ИЛИ
&&	Укороченное И
||	Укороченное ИЛИ
!	НЕ

Таблица 5: Логические операторы в C#

p	q	p & q	p | q	p ^ q	!p
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

Таблица 5: Логические операторы в C#

p	q	p & q	p | q	p ^ q	!p
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

Упражнение 2.2

Написать программу для реализация операции импликации.

Таблица 5: Логические операторы в C#

p	q	p & q	p | q	p ^ q	!p
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

Упражнение 2.2

Написать программу для реализация операции импликации.

Упражнение 2.3

Написать программу укороченного варианта логического оператора "И"(&&) в которой обычный оператор(&) работать не будет.

Таблица 5: Логические операторы в C#

p	q	p & q	p | q	p ^ q	!p
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

Упражнение 2.2

Написать программу для реализация операции импликации.

Упражнение 2.3

Написать программу укороченного варианта логического оператора "И"(&&) в которой обычный оператор(&) работать не будет.

Упражнение 2.4

Написать программу с обратным примером.

Общий вид оператора присваивания:

имя_переменной = выражение

Общий вид оператора присваивания:

имя_переменной = выражение

Пример:

```
int x, y, z;
x = y = z = 100; // присвоить значение 100 переменным x, y и z
```

Общий вид оператора присваивания:

имя_переменной = выражение

Общий вид составного оператора присваивания:

имя_переменной оп= выражение,
где оп — арифметический или логический оператор, применяемый вместе с оператором присваивания

Общий вид оператора присваивания:

имя_переменной = выражение

Общий вид составного оператора присваивания:

имя_переменной оп= выражение,
где оп — арифметический или логический оператор, применяемый вместе с оператором присваивания

Таблица 6: Составные операторы присваивания

+=	−=	*=	/=
%=	&=	|=	^=

Общий вид оператора присваивания:

имя_переменной = выражение

Общий вид составного оператора присваивания:

имя_переменной оп= выражение,
где оп — арифметический или логический оператор, применяемый вместе с оператором присваивания

Таблица 6: Составные операторы присваивания

+=	−=	*=	/=
%=	&=	|=	^=

Пример:
Оператор x = x - 100;
эквивалентен оператору x −= 100;

Таблица 7: Поразрядные операторы

Оператор	Значение
&	Поразрядное И
|	Поразрядное ИЛИ
^	Поразрядное исключающее ИЛИ
>>	Сдвиг вправо
<<	Сдвиг влево
~	Дополнение до 1 (унарный оператор НЕ)

Таблица 8: Поразрядные операторы

p	q	p & q	p | q	p ^ q	!p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

Таблица 7: Поразрядные операторы

Оператор	Значение
&	Поразрядное И
|	Поразрядное ИЛИ
^	Поразрядное исключающее ИЛИ
>>	Сдвиг вправо
<<	Сдвиг влево
~	Дополнение до 1 (унарный оператор НЕ)

Таблица 8: Поразрядные операторы

p	q	p & q	p | q	p ^ q	!p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

Пример:

```
  1101 0011
& 1010 1010
-----------
  1000 0010
```

Упражнение 2.5

Применить поразрядный оператор "И",чтобы сделать число четным.

Упражнение 2.5

Применить поразрядный оператор "И",чтобы сделать число четным.

Упражнение 2.6

Применить поразрядный оператор "И",чтобы определить, является ли число нечетным.

Упражнение 2.5

Применить поразрядный оператор "И",чтобы сделать число четным.

Упражнение 2.6

Применить поразрядный оператор "И",чтобы определить, является ли число нечетным.

Упражнение 2.7

Применить поразрядный оператор "И",чтобы показать биты, составляющие байт.

Упражнение 2.5

Применить поразрядный оператор "И",чтобы сделать число четным.

Упражнение 2.6

Применить поразрядный оператор "И",чтобы определить, является ли число нечетным.

Упражнение 2.7

Применить поразрядный оператор "И",чтобы показать биты, составляющие байт.

Упражнение 2.8

Применить поразрядный оператор "ИЛИ",чтобы сделать число нечетным.

Свойство поразрядного оператора "исключающее ИЛИ"

$x = (x \hat{} y) \hat{} y;$

Свойство поразрядного оператора "исключающее ИЛИ"

```
x = (x ^ y) ^ y;
```

Пример:

```
int x= 33, y =71;
Console.WriteLine("Исходное значение:" + x);
x = (x ^ y) ^ y;
Console.WriteLine("Конечное значение:" + x);
```

Свойство поразрядного оператора "исключающее ИЛИ"

```
x = (x ^ y) ^ y;
```

Пример:

```
int x= 33, y =71;
Console.WriteLine("Исходное значение:" + x);
x = (x ^ y) ^ y;
Console.WriteLine("Конечное значение:" + x);
```

Результат:

Исходное значение: 33
Конечное значение: 33

Свойство поразрядного оператора "исключающее ИЛИ"

```
x = (x ^ y) ^ y;
```

Пример:

```
int x= 33, y =71;
Console.WriteLine("Исходное значение:" + x);
x = (x ^ y) ^ y;
Console.WriteLine("Конечное значение:" + x);
```

Результат:

Исходное значение: 33
Конечное значение: 33

Упражнение 2.9

Написать программу для шифрования текстовых сообщений. В качества ключа использовать число типа int.

Свойство поразрядного оператора "исключающее ИЛИ"

```
x = (x ^ y) ^ y;
```

Пример:

```
int x= 33, y =71;
Console.WriteLine("Исходное значение:" + x);
x = (x ^ y) ^ y;
Console.WriteLine("Конечное значение:" + x);
```

Результат:

Исходное значение: 33
Конечное значение: 33

Упражнение 2.9

Написать программу для шифрования текстовых сообщений. В качества ключа использовать число типа int.

Упражнение 2.10

Написать программу для расшифровки текстовых сообщений. Предполагается, что в качества ключа шифрования использовалось число типа int.

Общий вид оператора if:

```
if(условие)
{
    последовательность операторов
}
else
{
    последовательность операторов
}
```

Общий вид оператора if:

```
if(условие)
{
    последовательность операторов
}
else
{
    последовательность операторов
}
```

Компактная форма:

```
if(условие) оператор;
else оператор;
```

Общий вид оператора if:

```
if(условие)
{
    последовательность операторов
}
else
{
    последовательность операторов
}
```

Компактная форма:

```
if(условие) оператор;
else оператор;
```

Вложенный оператор if:

Вложенным называется такой оператор if, который является адресатом другого оператора if или же оператора else.

Пример:

```
if (i == 10)
{
    if(j < 20) a = b;
    if(k > 100) c = d;
    else a = c; // этот оператор else связан с оператором if(k > 100)
}
else a = d; // этот оператор else связан с оператором if(i == 10)
```

Вложенный оператор if:

Вложенным называется такой оператор if, который является адресатом другого оператора if или же оператора else.

Многоступенчатая конструкция if-else-if:

```
if(условие) оператор;
else if (условие) оператор;
else if (условие) оператор;
else оператор;
```

Общий вид оператора switch:

```
switch(выражение)
{
    case константа1:
    последовательность операторов
    break;
    case константа2:
    последовательность операторов
    break;
    case константа3:
    последовательность операторов
    break;
    default:
    последовательность операторов
    break;
}
```

Общий вид оператора switch:

```
switch(выражение)
{
    case константа1:
    последовательность операторов
    break;
    case константа2:
    последовательность операторов
    break;
    case константа3:
    последовательность операторов
    break;
    default:
    последовательность операторов
    break;
}
```

Замечание

Для управления оператором switch может быть использовано выражение любого целочисленного типа, включая и char

Пример 1:

```
for(int i=1; i < 5; i++)
switch(i)
{
    case 1:
    case 2:
    case 3: Console.WriteLine("i равно 1, 2 или 3");
    break;
    case 4: Console.WriteLine("i равно 4");
    break;
}
```

Пример 2:

```
switch(ch1)
{
    case 'A': Console.WriteLine("Эта ветвь A — часть" +
                        "внешнего оператора switch.");
    switch(ch2)
    {
        case 'A':
        Console.WriteLine("Эта ветвь A — часть" +
                    "внутреннего оператора switch");
        break;
        case 'B': // ...
    } // конец внутреннего оператора switch
    break;
    case 'B': // ...
}
```

Общий вид оператора for:

```
for(инициализация; условие; итерация)
{
    последовательность операторов;
}
```

Общий вид оператора for:

```
for(инициализация; условие; итерация)
{
    последовательность операторов;
}
```

Компактная форма:

```
for(инициализация; условие; итерация) оператор;
```

Общий вид оператора for:

```
for(инициализация; условие; итерация)
{
    последовательность операторов;
}
```

Компактная форма:

```
for(инициализация; условие; итерация) оператор;
```

Упражнение 2.11

Использовать операторы цикла for для выявления простых чисел в пределах от 2 до 20. Если число оказывается непростым, вывести его наибольший множитель.

Замечание

В операторе цикла for разрешается использовать две или более переменных для управления циклом. В этом случае операторы инициализации и инкремента каждой переменной разделяются запятой.

Замечание

В операторе цикла for разрешается использовать две или более переменных для управления циклом. В этом случае операторы инициализации и инкремента каждой переменной разделяются запятой.

Пример: // Использовать запятые в операторе цикла for.

```
using System;
class Comma
{
    static void Main()
    {
        int i, j;
        for(i=0, j=10; i < j; i++, j--)
        Console.WriteLine("i и j:" + i + " " + j);
    }
}
```

Замечание

В операторе цикла for разрешается использовать две или более переменных для управления циклом. В этом случае операторы инициализации и инкремента каждой переменной разделяются запятой.

Пример: // Использовать запятые в операторе цикла for.

```
using System;
class Comma
{
    static void Main()
    {
        int i, j;
        for(i=0, j=10; i < j; i++, j--)
        Console.WriteLine("i и j:" + i + " " + j);
    }
}
```

Упражнение 2.12

Использовать две переменные управления одним циклом for для выявления наибольшего и наименьшего множителя заданного целого числа.

Замечание

Условным выражением, управляющим циклом for, может быть любое действительное выражение, дающее результат типа bool.

Замечание

Условным выражением, управляющим циклом for, может быть любое действительное выражение, дающее результат типа bool.

Пример: // Условием выполнения цикла может служить любое выражение типа bool.

```
using System;
class forDemo {
    static void Main()
    {
        int i, j;
        bool done = false;
        for(i=0, j=100; !done; i++, j--)
        {
            if(i*i >= j) done = true;
            Console.WriteLine("i, j:" + i + " " + j);
        }
    }
}
```

Пример 1: // Отдельные части цикла for могут оставаться пустыми.

```
int i;
for(i = 0; i < 10; )
{
    Console.WriteLine("Проход №" + i);
    i++; // инкрементировать переменную управления циклом
}
```

Пример 1: // Отдельные части цикла for могут оставаться пустыми.

```
int i;
for(i = 0; i < 10; )
{
    Console.WriteLine("Проход №" + i);
    i++; // инкрементировать переменную управления циклом
}
```

Пример 2:

```
int i= 0; // исключить инициализацию из определения цикла
for(; i < 10; )
{
    Console.WriteLine("Проход №" + i);
    i++; // инкрементировать переменную управления циклом
}
```

Пример 1: // Отдельные части цикла for могут оставаться пустыми.

```
int i;
for(i = 0; i < 10; )
{
    Console.WriteLine("Проход №" + i);
    i++; // инкрементировать переменную управления циклом
}
```

Пример 2:

```
int i= 0; // исключить инициализацию из определения цикла
for(; i < 10; )
{
    Console.WriteLine("Проход №" + i);
    i++; // инкрементировать переменную управления циклом
}
```

Пример 3: // Отдельные части цикла for могут оставаться пустыми.

```
int i, sum = 0;
// получить сумму чисел от 1 до 5
for(i = 1; i <= 5; sum += i++);
Console.WriteLine("Сумма равна" + sum);
```

Общий вид оператора while

```
while (условие)
{
    последовательность операторов;
}
```

Общий вид оператора while

```
while (условие)
{
    последовательность операторов;
}
```

Общий вид оператора do-while

```
do
{
    последовательность операторов;
}
while (условие);
```

Общий вид оператора while

```
while (условие)
{
    последовательность операторов;
}
```

Общий вид оператора do-while

```
do
{
    последовательность операторов;
}
while (условие);
```

Упражнение 2.13

С помощью оператора do-while отобразить цифры целого числа в обратном порядке

Замечание

С помощью оператора break можно специально организовать немедленный выход из цикла в обход любого кода, оставшегося в теле цикла, а также минуя проверку условия цикла.

Замечание

С помощью оператора break можно специально организовать немедленный выход из цикла в обход любого кода, оставшегося в теле цикла, а также минуя проверку условия цикла.

Замечание

С помощью оператора continue можно организовать преждевременное завершение шага итерации цикла в обход обычной структуры управления циклом.

Замечание

С помощью оператора break можно специально организовать немедленный выход из цикла в обход любого кода, оставшегося в теле цикла, а также минуя проверку условия цикла.

Замечание

С помощью оператора continue можно организовать преждевременное завершение шага итерации цикла в обход обычной структуры управления циклом.

Пример:

```
// вывести четные числа от 0 до 100.
for (int i = 0; i <= 100; i++)
{
    if((i%2) != 0) continue; // перейти к следующему шагу итерации
    Console.WriteLine(i);
}
```

Оператор goto

Оператор goto представляет собой оператор безусловного перехода. Когда в программе встречается данный оператор, ее выполнение переходит непосредственно к тому месту, на которое указывает этот оператор.

Оператор goto

Оператор goto представляет собой оператор безусловного перехода. Когда в программе встречается данный оператор, ее выполнение переходит непосредственно к тому месту, на которое указывает этот оператор.

Пример:

```
// Продемонстрировать практическое применение оператора goto.
int i=0, j=0, k=0;
for(i=0; i < 10; i++)
{
    for(j=0; j < 10; j++ )
    {
        for(k=0; k < 10; k++)
        {
            Console.WriteLine("i, j, k:" + i + " " + j + " " + k);
            if(k == 3) goto stop;
        }
    }
}
stop:
Console.WriteLine("Остановлено! i, j, k:" + i + "," + j + " " + k);
```

Общий вид:

```
class имя_класса
{
    // Объявление переменных экземпляра.
    доступ тип переменная1;
    доступ тип переменная2;
    //...
    доступ тип переменнаяN;
    // Объявление методов.
    доступ возращаемый_тип метод1(параметры)
    {
        // тело метода
    }
    доступ возращаемый_тип метод2(параметры)
    {
        // тело метода
    }
    // ...
    доступ возращаемый_тип методN(параметры)
    {
        // тело метода
    }
}
```

```
доступ возвращаемый_тип имя_метода(параметры)
{
    // тело метода
    //...
    if(done) return значение; // значение имеет возвращаемый_тип
    // ...
    if(error) return значение; // значение  имеет возвращаемый_тип
}
```

```
доступ возвращаемый_тип имя_метода(параметры)
{
    // тело метода
    //...
    if(done) return значение; // значение имеет возвращаемый_тип
    // ...
    if(error) return значение; // значение  имеет возвращаемый_тип
}
```

Пример:

```
int Method1(int a, double b, float c)
{
     // ...
     return a;
    Console.WriteLine("это недоступный код");
}
```

Общий вид:

```
доступ имя_класса(список_параметров )
{
    // тело конструктора
}
```

Пример 1:

```
// Простой конструктор.
using System;
class MyClass
{
    public int x;
    public MyClass()
    {
        x = 10;
    }
}
class ConsDemo
{
    static void Main()
    {
        MyClass t1 = new MyClass();
        MyClass t2 = new MyClass();
        Console.WriteLine(t1.x +" "+ t2.x);
    }
}
```

Пример 2:

```
// Параметризированный конструктор.
using System;
class MyClass
{
    public int x;
    public MyClass(int i)
    {
        x = i;
    }
}
class ParmConsDemo
{
    static void Main()
    {
        MyClass t1 = new MyClass(10);
        MyClass t2 = new MyClass(88);
        Console.WriteLine(t1.x +" "+ t2.x);
    }
}
```

Форма оператора new

```
new имя_класса(список_аргументов)
```

Форма оператора new

new имя_класса(список_аргументов)

Пример:

```
// Использовать оператор new вместе с типом значения.
using System;
class newValue
{
    static void Main()
    {
        int i = new int(); // инициализировать переменную i нулевым
значением
        Console.WriteLine("Значение переменной i равно:" + i);
    }
}
```

Общий вид деструктора:

```
~ имя_класса()
{
    // код деструктора
}
```

Пример:

```
// Продемонстрировать применение деструктора.
using System;
class Destruct
{
    public int x;
    public Destruct(int i)
    {
        x = i;
    }
    // Вызывается при утилизации объекта.
    ~Destruct()
    {
        Console.WriteLine("Уничтожить "+ x);
    }
    // Создает объект и тут же уничтожает его.
    public void Generator(int i)
    {
        Destruct o = new Destruct(i);
    }
}
```

```
class DestructDemo
{
    static void Main()
    {
        int count;
        Destruct ob = new Destruct(0);
        /* А теперь создадим большое число объектов.
        В какой-то момент произойдет "сборка мусора".
        Примечание: для того чтобы активизировать
        "сборку мусора возможно, придется увеличить
        число создаваемых объектов. */
        for (count = 1; count < 10; count++)
        ob.Generator(count);
        Console.WriteLine("Готово!");
    }
}
```

Пример 1:
```
using System;
class Rect
{
    public int Width;
    public int Height;
    public Rect(int w, int h)
    {
        Width = w;
        Height = h;
     }
    public int Area()
    {
        return Width * Height;
    }
}
```

Пример 1:
```
using System;
class Rect
{
    public int Width;
    public int Height;
    public Rect(int w, int h)
    {
        this.Width = w;
        this.Height = h;
     }
    public int Area()
    {
        return this.Width * this.Height;
    }
}
```

Пример 2:
```
using System;
class Rect
{
    public int Width;
    public int Height;
    public Rect(int Width, int Height)
    {
        this.Width = Width;
        this.Height = Height;
     }
    public int Area()
    {
        return this.Width * this.Height;
    }
}
```

Общая форма:

тип[ ] имя_массива = new тип[размер];

где тип объявляет конкретный тип элемента массива.

Общая форма:

тип[ ] имя_массива = new тип[размер];
где тип объявляет конкретный тип элемента массива.

Пример:
int[ ] sample = new int[10];

Общая форма:

тип[ ] имя_массива = new тип[размер];
где тип объявляет конкретный тип элемента массива.

Пример:

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

Общая форма:

тип[ ] имя_массива = new тип[размер];
где тип объявляет конкретный тип элемента массива.

Общая форма инициализации одномерного массива:

тип[ ] имя_массива = {val1, val2, val3, ..., valN};

Общая форма:

тип[ ] имя_массива = new тип[размер];
где тип объявляет конкретный тип элемента массива.

Общая форма инициализации одномерного массива:

тип[ ] имя_массива = {val1, val2, val3, ..., valN};

Пример:
int[ ] nums = { 99, 10, 100, 18, 78, 23, 63, 9, 87, 49 };

Общая форма:

тип[ ] имя_массива = new тип[размер];
где тип объявляет конкретный тип элемента массива.

Общая форма инициализации одномерного массива:

тип[ ] имя_массива = {val1, val2, val3, ..., valN};

Пример:

```
int[ ] nums;
nums = new int[ ] { 99, 10, 100, 18, 78, 23, 63, 9, 87, 49 };
```

Общая форма:

тип[ ] имя_массива = new тип[размер];
где тип объявляет конкретный тип элемента массива.

Общая форма инициализации одномерного массива:

тип[ ] имя_массива = {val1, val2, val3, ..., valN};

Общая форма объявления многомерного массива:

тип[,...,] имя_массива = new тип[размер1, размер2, ... размерN];

Общая форма:

тип[ ] имя_массива = new тип[размер];
где тип объявляет конкретный тип элемента массива.

Общая форма инициализации одномерного массива:

тип[ ] имя_массива = {val1, val2, val3, ..., valN};

Общая форма объявления многомерного массива:

тип[,...,] имя_массива = new тип[размер1, размер2, ... размерN];

Упражнение 3.1

Написать программу для создания двумерного массива инициализированного целыми числами и посчитать его определитель.

Общая форма:

тип[ ] имя_массива = new тип[размер];
где тип объявляет конкретный тип элемента массива.

Общая форма инициализации одномерного массива:

тип[ ] имя_массива = {val1, val2, val3, ..., valN};

Общая форма объявления многомерного массива:

тип[,...,] имя_массива = new тип[размер1, размер2, ... размерN];

Общая форма инициализации двумерного массива:

тип[,] имя_массива = { {val, val, val, ..., val}, {val, val, val, ..., val}, {val, val, val, ..., val} };

Общая форма:

тип[ ] имя_массива = new тип[размер];
где тип объявляет конкретный тип элемента массива.

Общая форма инициализации одномерного массива:

тип[ ] имя_массива = {val1, val2, val3, ..., valN};

Общая форма объявления многомерного массива:

тип[,...,] имя_массива = new тип[размер1, размер2, ... размерN];

Общая форма инициализации двумерного массива:

тип[,] имя_массива = { {val, val, val, ..., val}, {val, val, val, ..., val}, {val, val, val, ..., val} };

Общая форма объявления двумерного ступенчатого массива:

тип[ ][ ] имя_массива = new тип[размер][ ];

Общая форма:

тип[ ] имя_массива = new тип[размер];
где тип объявляет конкретный тип элемента массива.

Общая форма инициализации одномерного массива:

тип[ ] имя_массива = {val1, val2, val3, ..., valN};

Пример:

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

Общая форма объявления двумерного ступенчатого массива:

тип[ ][ ] имя_массива = new тип[размер][ ];

Общая форма:

тип[ ] имя_массива = new тип[размер];
где тип объявляет конкретный тип элемента массива.

Общая форма инициализации одномерного массива:

тип[ ] имя_массива = {val1, val2, val3, ..., valN};

Пример:

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

Пример (массив массивов):

```
int[ ][ , ] jagged = new int[3][ , ];
```

Общая форма объявления двумерного ступенчатого массива:

тип[ ][ ] имя_массива = new тип[размер][ ];

Общая форма:

тип[ ] имя_массива = new тип[размер];
где тип объявляет конкретный тип элемента массива.

Общая форма инициализации одномерного массива:

тип[ ] имя_массива = {val1, val2, val3, ..., valN};

Упражнение 3.2

Используя свойство Length, поменять местами содержимое элементов массива

Общая форма объявления двумерного ступенчатого массива:

тип[ ][ ] имя_массива = new тип[размер][ ];

```
// Продемонстрировать неявно типизированный ступенчатый массив.
using System;
class Jagged
{
    static void Main()
    {
        var jagged = new [ ] {
        new [ ] { 1, 2, 3, 4 },
        new [ ] { 9, 8, 7 },
        new [ ] { 11, 12, 13, 14, 15 } };
        for(int j = 0; j < jagged.Length; j++)
        {
            for(int i=0; i < jagged[j].Length; i++)
            Console.Write(jagged[j][i] + );
            Console.WriteLine();
        }
    }
}
```

Общая форма оператора цикла foreach

foreach (тип имя_переменной_цикла in коллекция) оператор;

Общая форма оператора цикла foreach

foreach (тип имя_переменной_цикла in коллекция) оператор;

Замечание

Тип переменной цикла должен соответствовать типу элемента массива

Пример:

```
using System;
class foreachDemo
{
    static void Main()
    {
        int sum = 0;
        int[ ] nums = new int[10];
        // Задать первоначальные значения элементов массива nums.
        for (int i = 0; i < 10; i++)
        nums[i] = i;
        // Использовать цикл foreach для вывода значений
        // элементов массива и подсчета их суммы.
        foreach (int x in nums)
        {
            Console.WriteLine("Значение элемента равно: " + x);
            sum += x;
        }
        Console.WriteLine("Сумма равна: " + sum);
    }
}
```

Результат:

```
Значение элемента равно: 0
Значение элемента равно: 1
Значение элемента равно: 2
Значение элемента равно: 3
Значение элемента равно: 4
Значение элемента равно: 5
Значение элемента равно: 6
Значение элемента равно: 7
Значение элемента равно: 8
Значение элемента равно: 9
Сумма равна: 45
```

Результат:

```
Значение элемента равно: 0
Значение элемента равно: 1
Значение элемента равно: 2
Значение элемента равно: 3
Значение элемента равно: 4
Значение элемента равно: 5
Значение элемента равно: 6
Значение элемента равно: 7
Значение элемента равно: 8
Значение элемента равно: 9
Сумма равна: 45
```

Упражнение 3.3

Использовать оператор foreach для вычисления суммы элементов двумерного массива

Инициализация с помощью строкового литерала:

```
string str = "Строки в C# весьма эффективны.";
```

Инициализация с помощью строкового литерала:

```
string str = "Строки в C# весьма эффективны.";
```

Инициализация с помощью массива типа char:

```
char[ ] charray = {'t', 'e', 's', 't'};
string  str = new string(charray);
```

Таблица 1.Некоторые общеупотребительные методы обращения со строками

Метод	Описание
`static int Compare(string strA, string strB, StringComparison comparisonType)`	Возвращает отрицательное значение, если строка `strA` меньше строки `strB`; положительное значение, если строка `strA` больше строки `strB`; и нуль, если сравниваемые строки равны. Способ сравнения определяется аргументом `comparisonType`
`bool Equals(string value, StringComparison comparisonType)`	Возвращает логическое значение `true`, если вызывающая строка имеет такое же значение, как и у аргумента `value`. Способ сравнения определяется аргументом `comparisonType`
`int IndexOf(char value)`	Осуществляет поиск в вызывающей строке первого вхождения символа, определяемого аргументом `value`. Применяется порядковый способ поиска. Возвращает индекс первого совпадения с искомым символом или -1, если он не обнаружен
`int IndexOf(string value, StringComparison comparisonType)`	Осуществляет поиск в вызывающей строке первого вхождения подстроки, определяемой аргументом `value`. Возвращает индекс первого совпадения с искомой подстрокой или -1, если она не обнаружена. Способ поиска определяется аргументом `comparisonType`

Таблица 1.Некоторые общеупотребительные методы обращения со строками

Метод	**Описание**
`int LastIndexOf(char value)`	Осуществляет поиск в вызывающей строке последнего вхождения символа, определяемого аргументом *value*. Применяется порядковый способ поиска. Возвращает индекс последнего совпадения с искомым символом или -1, если он не обнаружен
`int LastIndexOf(string value, StringComparison comparisonType)`	Осуществляет поиск в вызывающей строке последнего вхождения подстроки, определяемой аргументом *value*. Возвращает индекс последнего совпадения с искомой подстрокой или -1, если она не обнаружена. Способ поиска определяется аргументом *comparisonType*
`string ToLower(CultureInfo. CurrentCulture culture)`	Возвращает вариант вызывающей строки в нижнем регистре. Способ преобразования определяется аргументом *culture*
`string ToUpper(CultureInfo. CurrentCulture culture)`	Возвращает вариант вызывающей строки в верхнем регистре. Способ преобразования определяется аргументом *culture*

Упражнение 3.4

Инициализировать несколько строк. Используя описанные выше методы:

а) Создать варианты строки набранные прописными и строчными буквами.

б) Вывести строку посимвольно.

в) Сравнить строки с учетом культурной среды.

г) Выделить из одной из строк подстроку и найти индекс ее первого вхождения в исходную строку.

Упражнение 3.4

Инициализировать несколько строк. Используя описанные выше методы:

а) Создать варианты строки набранные прописными и строчными буквами.

б) Вывести строку посимвольно.

в) Сравнить строки с учетом культурной среды.

г) Выделить из одной из строк подстроку и найти индекс ее первого вхождения в исходную строку.

Упражнение 3.5

Вывести отдельные цифры целого числа словами.

Пример:

```
// Продемонстрировать управление оператором switch посредством строк.
using System;
class StringSwitch {
static void Main() {
        string[ ] strs = { "один", "два", "три", "два", "один" };
        foreach (string s in strs)
         {   switch (s)
            {
                case "один":
                    Console.Write(1);
                    break;
                case "два":
                    Console.Write(2);
                    break;
                case "три":
                    Console.Write(3);
                    break;
            }
        }
        Console.WriteLine();      } }
```

Пример:

```
// Отличия между видами доступа public и private к членам класса.
using System;
class Myclass
{
    private int alpha; // закрытый доступ, указываемый явно
    int beta; // закрытый доступ по умолчанию
    public int gamma; // открытый доступ
    // Методы, которым доступны члены alpha и beta данного класса.
   //Член класса может иметь доступ к закрытому члену этого же класса.
    public void SetAlpha(int a){ alpha = a;}
    public int GetAlpha(){ return alpha;}
    public void SetBeta(int a){ beta = a;}
    public int GetBeta(){ return beta;}
}
```

Пример:

```
class AccessDemo
{
    static void Main()
    {
        Myclass ob = new Myclass();
        // Доступ к членам alpha и beta данного класса
        // разрешен только посредством его методов.
        ob.SetAlpha(-99);
        ob.SetBeta(19);
        Console.WriteLine("ob.alpha равно " + ob.GetAlpha());
        Console.WriteLine("ob.beta равно " + ob.GetBeta ());
        // Следующие виды доступа к членам alpha и beta
        // данного класса не разрешаются.
        // ob.alpha = 10; // Ошибка! alpha - закрытый член!
        // ob.beta =9; // Ошибка! beta - закрытый член!
        // Член gamma данного класса доступен непосредственно,
        // поскольку он является открытым.
        ob.gamma = 99;
    }
}
```

Общие принципы:

- Члены, используемые только в классе, должны быть закрытыми.
- Данные экземпляра, не выходящие за определенные пределы значений, должны быть закрытыми, а при организации доступа к ним с помощью открытых методов следует выполнять проверку диапазона представления чисел.
- Если изменение члена приводит к последствиям, распространяющимся за пределы области действия самого члена, т.е. оказывает влияние на другие аспекты объекта, то этот член должен быть закрытым, а доступ к нему — контролируемым.
- Члены, способные нанести вред объекту, если они используются неправильно, должны быть закрытыми. Доступ к этим членам следует организовать с помощью открытых методов, исключающих неправильное их использование.
- Методы, получающие и устанавливающие значения закрытых данных, должны быть открытыми.
- Переменные экземпляра допускается делать открытыми лишь в том случае, если нет никаких оснований для того, чтобы они были закрытыми.

Общие принципы:

- Члены, используемые только в классе, должны быть закрытыми.
- Данные экземпляра, не выходящие за определенные пределы значений, должны быть закрытыми, а при организации доступа к ним с помощью открытых методов следует выполнять проверку диапазона представления чисел.
- Если изменение члена приводит к последствиям, распространяющимся за пределы области действия самого члена, т.е. оказывает влияние на другие аспекты объекта, то этот член должен быть закрытым, а доступ к нему — контролируемым.
- Члены, способные нанести вред объекту, если они используются неправильно, должны быть закрытыми. Доступ к этим членам следует организовать с помощью открытых методов, исключающих неправильное их использование.
- Методы, получающие и устанавливающие значения закрытых данных, должны быть открытыми.
- Переменные экземпляра допускается делать открытыми лишь в том случае, если нет никаких оснований для того, чтобы они были закрытыми.

Общие принципы:

- Члены, используемые только в классе, должны быть закрытыми.
- Данные экземпляра, не выходящие за определенные пределы значений, должны быть закрытыми, а при организации доступа к ним с помощью открытых методов следует выполнять проверку диапазона представления чисел.
- Если изменение члена приводит к последствиям, распространяющимся за пределы области действия самого члена, т.е. оказывает влияние на другие аспекты объекта, то этот член должен быть закрытым, а доступ к нему — контролируемым.
- Члены, способные нанести вред объекту, если они используются неправильно, должны быть закрытыми. Доступ к этим членам следует организовать с помощью открытых методов, исключающих неправильное их использование.
- Методы, получающие и устанавливающие значения закрытых данных, должны быть открытыми.
- Переменные экземпляра допускается делать открытыми лишь в том случае, если нет никаких оснований для того, чтобы они были закрытыми.

Общие принципы:

- Члены, используемые только в классе, должны быть закрытыми.
- Данные экземпляра, не выходящие за определенные пределы значений, должны быть закрытыми, а при организации доступа к ним с помощью открытых методов следует выполнять проверку диапазона представления чисел.
- Если изменение члена приводит к последствиям, распространяющимся за пределы области действия самого члена, т.е. оказывает влияние на другие аспекты объекта, то этот член должен быть закрытым, а доступ к нему — контролируемым.
- Члены, способные нанести вред объекту, если они используются неправильно, должны быть закрытыми. Доступ к этим членам следует организовать с помощью открытых методов, исключающих неправильное их использование.
- Методы, получающие и устанавливающие значения закрытых данных, должны быть открытыми.
- Переменные экземпляра допускается делать открытыми лишь в том случае, если нет никаких оснований для того, чтобы они были закрытыми.

Общие принципы:

- Члены, используемые только в классе, должны быть закрытыми.
- Данные экземпляра, не выходящие за определенные пределы значений, должны быть закрытыми, а при организации доступа к ним с помощью открытых методов следует выполнять проверку диапазона представления чисел.
- Если изменение члена приводит к последствиям, распространяющимся за пределы области действия самого члена, т.е. оказывает влияние на другие аспекты объекта, то этот член должен быть закрытым, а доступ к нему — контролируемым.
- Члены, способные нанести вред объекту, если они используются неправильно, должны быть закрытыми. Доступ к этим членам следует организовать с помощью открытых методов, исключающих неправильное их использование.
- Методы, получающие и устанавливающие значения закрытых данных, должны быть открытыми.
- Переменные экземпляра допускается делать открытыми лишь в том случае, если нет никаких оснований для того, чтобы они были закрытыми.

Общие принципы:

- Члены, используемые только в классе, должны быть закрытыми.
- Данные экземпляра, не выходящие за определенные пределы значений, должны быть закрытыми, а при организации доступа к ним с помощью открытых методов следует выполнять проверку диапазона представления чисел.
- Если изменение члена приводит к последствиям, распространяющимся за пределы области действия самого члена, т.е. оказывает влияние на другие аспекты объекта, то этот член должен быть закрытым, а доступ к нему — контролируемым.
- Члены, способные нанести вред объекту, если они используются неправильно, должны быть закрытыми. Доступ к этим членам следует организовать с помощью открытых методов, исключающих неправильное их использование.
- Методы, получающие и устанавливающие значения закрытых данных, должны быть открытыми.
- Переменные экземпляра допускается делать открытыми лишь в том случае, если нет никаких оснований для того, чтобы они были закрытыми.

Общие принципы:

- Члены, используемые только в классе, должны быть закрытыми.
- Данные экземпляра, не выходящие за определенные пределы значений, должны быть закрытыми, а при организации доступа к ним с помощью открытых методов следует выполнять проверку диапазона

Упражнение 3.7

Используя модификаторы доступа, реализовать стек из элементов char. Добавить методы с помощью которых можно:

а) поместить символ в стек

б) извлечь символ из стека

в) проверить заполнен стек или нет

г) возвратить общую емкость стека

д) возвратить количество объектов, находящихся в данный момент в стеке.

- Переменные экземпляра допускается делать открытыми лишь в том случае, если нет никаких оснований для того, чтобы они были закрытыми.

Две причины по которым значение простого типа иногда требуется передавать по ссылке:

- разрешить методу изменить содержимое его аргументов
- возвратить несколько значений

Две причины по которым значение простого типа иногда требуется передавать по ссылке:

- разрешить методу изменить содержимое его аргументов
- возвратить несколько значений

Пример:

```
// Использовать модификатор ref для
// передачи значения обычного типа по ссылке.
using System;
class RefTest
{
     // Этот метод изменяет свой аргумент. Обратите
     // внимание на применение модификатора ref.
     public void Sqr(ref int i)
    {
      i = i * i;
    }
}
class RefDemo
{
    static void Main()
    {
      RefTest ob = new RefTest();
      int a = 10;
      Console.WriteLine("a до вызова: " + a);
      ob.Sqr(ref a); // обратите внимание на применение модификатора ref
      Console.WriteLine("a после вызова: " + a);
    }
}
```

Пример: // Использовать модификатор ref для
// передачи значения обычного типа по ссылке.

```
using System;
```

Результат:

```
a до вызова: 10
a после вызова: 100
```

```
class RefDemo
{
    static void Main()
    {
      RefTest ob = new RefTest();
      int a = 10;
      Console.WriteLine("a до вызова: " + a);
      ob.Sqr(ref a); // обратите внимание на применение модификатора ref
      Console.WriteLine("a после вызова: " + a);
    }
}
```

```
using System;
class Decompose
{ /* Разделить числовое значение с плавающей точкой на целую и
дробную части. */
    public int GetParts(double n, out double frac)
    {
        frac = n - (int)n;//передать дробную часть числа через параметр frac
        return (int)n; // возвратить целую часть числа
     }
}
class UseOut
{
static void Main()
    {
        Decompose ob = new Decompose();
        int i;
        double f;
        i = ob.GetParts(10.125, out f);
        Console.WriteLine("Целая часть числа равна " + i);
        Console.WriteLine("Дробная часть числа равна " + f);
    }
}
```

Упражнение 3.8

Написать метод который, во-первых, определяет общий множитель (кроме 1) для двух целых чисел, возвращая логическое значение true, если у них имеется общий множитель, а иначе — логическое значение false. И, во-вторых, возвращает посредством параметров типа out наименьший и наибольший общий множитель двух чисел, если таковые обнаруживаются

Метод с переменным числом параметров:

```
доступ тип имя_метода(params тип[ ] имя_параметра)
```

Метод с переменным числом параметров:

доступ тип имя_метода(params тип[ ] имя_параметра)

Упражнение 3.9

Реализовать метод, обнаруживающий наименьшее среди ряда значений.

Метод с переменным числом параметров:

доступ тип имя_метода(params тип[ ] имя_параметра)

Упражнение 3.9

Реализовать метод, обнаруживающий наименьшее среди ряда значений.

Общий вид:

доступ тип имя_метода(тип имя_параметра1, тип имя_параметра2,...,
тип имя_параметра N-1, params тип[ ] имя_параметраN)

Пример:

```
using System;
class Myclass
{
    public void ShowArgs(string msg, params int[] nums)
    {
        Console.Write(msg + " ");
        foreach(int i in nums)
        Console.Write(i + " ");
        Console.WriteLine();
    }
}
class ParamsDemo2
{
    static void Main()
    {
        Myclass ob = new Myclass();
        ob.ShowArgs("Это ряд целых чисел",1, 2, 3, 4, 5);
        ob.ShowArgs("А это еще два целых числа", 17, 20);
    }
}
```

Пример:

Результат:

Это ряд целых чисел: 1, 2, 3, 4, 5
А это еще два целых числа: 17, 20

```
class ParamsDemo2
{
    static void Main()
    {
        Myclass ob = new Myclass();
        ob.ShowArgs("Это ряд целых чисел",1, 2, 3, 4, 5);
        ob.ShowArgs("А это еще два целых числа", 17, 20);
    }
}
```

Общий вид:

доступ возвращаемый_тип метод(параметры)

Общий вид:

доступ возвращаемый_тип метод(параметры)

Упражнение 3.10

Создать фабрику класса (т.е. метод, предназначенный для построения объектов его же класса).

Общий вид:

доступ возвращаемый_тип метод(параметры)

Упражнение 3.10

Создать фабрику класса (т.е. метод, предназначенный для построения объектов его же класса).

Возврат массива из метода:

доступ возвращаемый_тип[ ] метод(параметры)

Общий вид:

доступ возвращаемый_тип метод(параметры)

Упражнение 3.10

Создать фабрику класса (т.е. метод, предназначенный для построения объектов его же класса).

Возврат массива из метода:

доступ возвращаемый_тип[ ] метод(параметры)

Упражнение 3.11

Реализовать метод, который возвращает массив, содержащий множители переданного ему аргумента

Перегрузка методов:

```
доступ возвращаемый_тип имя_метода(параметры /*(Набор 1)*/)
доступ возвращаемый_тип имя_метода(параметры /*(Набор 2)*/)
```

Пример:

```
using System;
class Overload
{
    public void OvlDemo()
    {
        Console.WriteLine("Без параметров") ;
    }
    // Перегрузка метода OvlDemo с одним целочисленным параметром.
    public void OvlDemo(int a)
    {
        Console.WriteLine("Один параметр: " + a);
    }
    // Перегрузка метода OvlDemo с двумя параметрами типа double.
    public double OvlDemo(double a, double b)
    {
        Console.WriteLine("Два параметра типа double: " + a + " " + b);
        return a + b;
    }
}
```

Пример 1:

```
public void MyMeth(int x)
{
    Console.WriteLine("В методе MyMeth(int): " + x);
}
public void MyMeth( ref int x)
{
    Console.WriteLine("В методе MyMeth(ref int): " + x);
}
```

Пример 1:

```
public void MyMeth(int x)
{
    Console.WriteLine("В методе MyMeth(int): " + x);
}
public void MyMeth( ref int x)
{
    Console.WriteLine("В методе MyMeth(ref int): " + x);
}
```

Пример 2:

```
// Неверно!
public void MyMeth(out int x) { // ...
public void MyMeth(ref int x) { // ...
```

Пример:

```
// Продемонстрировать перегрузку конструктора.
using System;
class MyClass
{
    public int x;
    public MyClass()
    {
        Console.WriteLine("В конструкторе MyClass().") ;
        x = 0;
    }
    public MyClass(double d)
    {
        Console.WriteLine("В конструкторе MyClass(double).") ;
        x = (int)d;
    }
    public MyClass(int i, int j)
    {
        Console.WriteLine("В конструкторе MyClass(int, int).") ;
        x = i * j;
    }
}
```

Общая форма:

```
имя_конструктора(список_параметров1) : this(список_параметров2)
{ ... }
```

Пример:

```
// Продемонстрировать вызов конструктора с помощью ключевого слова this.
using System;
class XYCoord
{
    public int x, y;
    public XYCoord(): this(0, 0)
    {
        Console.WriteLine("В конструкторе XYCoord()") ;
    }
    public XYCoord(XYCoord obj) : this(obj.x, obj.y)
    {
        Console.WriteLine("В конструкторе XYCoord(obj)") ;
    }
    public XYCoord(int i, int j)
    {
        Console.WriteLine("В конструкторе XYCoord(int, int)") ;
        x = i;
        y = j;
    }
}
```

Пример:

```
// Простой пример, демонстрирующий применение инициализаторов
объектов.
using System;
class MyClass
{
    public int Count;
    public string Str;
}
class ObjInitDemo
{
    static void Main()
    {
        // Сконструировать объект типа MyClass, используя
инициализаторы объектов.
        MyClass obj = new MyClass { Count = 100, Str = "Тестирование" };
        Console.WriteLine(obj.Count + " " + obj.Str);
    }
}
```

Пример:

```
// Простой пример, демонстрирующий применение инициализаторов
объектов.
using System;
class MyClass
{
    public int Count;
    public string Str;
}
class ObjInitDemo
{
    static void Main()
    {
        // Сконструировать объект типа MyClass, используя
инициализаторы объектов.
        MyClass obj = new MyClass { Count = 100, Str = "Тестирование" };
        Console.WriteLine(obj.Count + " " + obj.Str);
    }
}
```

Общая форма инициализации:

```
new имя_класса {имя = выражение, имя = выражение, ...}
```

Пример: // Использовать необязательный аргумент, чтобы упростить вызов метода.

```
using System;
class UseOptArgs
{
    // Вывести на экран символьную строку полностью или частично.
    static void Display(string str, int start = 0, int stop = -1)
    {
        if (stop < 0) stop = str.Length;
        // Проверить условие выхода за заданные пределы.
        if (stop > str.Length | start > stop | start < 0) return;
        for (int i = start; i < stop; i++)
        Console.Write(str[i]);
        Console.WriteLine();
    }
    static void Main()
    {
        Display("это простой тест") ;
        Display("это простой тест", 12);
        Display("это простой тест", 4, 14);
    }
}
```

Пример: // Применить именованные аргументы.

```
using System;
class NamedArgsDemo
{
    static bool IsFactor(int val, int divisor)
    {
        if ((val % divisor) == 0) return true;
        return false;
    }
    static void Main()
    {
      // Ниже демонстрируются разные способы вызова метода IsFactor().
        if (IsFactor(10, 2))
        Console.WriteLine("2 - множитель 10.") ;
        if (IsFactor(val: 10, divisor: 2))
        Console.WriteLine("2 - множитель 10.") ;
        if (IsFactor(divisor: 2, val: 10))
        Console.WriteLine("2 - множитель 10.") ;
        if (IsFactor(10, divisor: 2))
        Console.WriteLine("2 - множитель 10.") ;
    }
}
```

Пример: static int Main()

Пример: static int Main()

Аргументы командной строки:
static void Main(string[] args) static int Main(string[] args)

Пример: static int Main()

Аргументы командной строки:
static void Main(string[] args) static int Main(string[] args)

Упражнение 4.1

Добавить в любую из предыдущих программ передачу аргументов методу Main(), так чтобы программа выполнялась по слову "Пароль" и выдавала бы сообщение "Пароль не верен" в противном случае.

Упражнение 4.2

С помощью рекурсивного метода вывести аргументы командной строки в обратном порядке.

Пример: // Использовать модификатор static.

```
using System;
class StaticDemo
{
    public static int Val = 100;// Переменная типа static.
    public static int ValDiv2() // Метод типа static.
    {
        return Val/2;
    }
}
class SDemo
{
    static void Main()
    {
        Console.WriteLine("Исходное значение переменной " +
        "StaticDemo.Val равно" + StaticDemo.Val);
        StaticDemo.Val = 8;
        Console.WriteLine("Текущее значение переменной" +
        "StaticDemo.Val равно " + StaticDemo.Val);
        Console.WriteLine("StaticDemo.ValDiv2(): " + StaticDemo.ValDiv2());
    }
}
```

Ограничения:

Ограничения:

- В методе типа static должна отсутствовать ссылка this, поскольку такой метод не выполняется относительно какого-либо объекта.

Ограничения:

- В методе типа static должна отсутствовать ссылка this, поскольку такой метод не выполняется относительно какого-либо объекта.
- В методе типа static допускается непосредственный вызов только других методов типа static, но не метода экземпляра из того самого же класса. Дело в том, что методы экземпляра оперируют конкретными объектами, а метод типа static не вызывается для объекта. Следовательно, у такого метода отсутствуют объекты, которыми он мог бы оперировать.

Ограничения:

- В методе типа static должна отсутствовать ссылка this, поскольку такой метод не выполняется относительно какого-либо объекта.
- В методе типа static допускается непосредственный вызов только других методов типа static, но не метода экземпляра из того самого же класса. Дело в том, что методы экземпляра оперируют конкретными объектами, а метод типа static не вызывается для объекта. Следовательно, у такого метода отсутствуют объекты, которыми он мог бы оперировать.
- Аналогичные ограничения накладываются на данные типа static. Для метода типа static непосредственно доступными оказываются только другие данные типа static, определенные в его классе. Он, в частности, не может оперировать переменной экземпляра своего класса, поскольку у него отсутствуют объекты, которыми он мог бы оперировать.

Пример 1:

```
class StaticError
{
    public int Denom = 3; // обычная переменная экземпляра
    public static int Val = 1024; // статическая переменная
    /* Ошибка! Непосредственный доступ к нестатической переменной из
    статического метода недопустим. */
    static int ValDivDenom() return Val/Denom; // не подлежит
компиляции!
}
```

Пример 1:

```
class StaticError
{
    public int Denom = 3; // обычная переменная экземпляра
    public static int Val = 1024; // статическая переменная
    /* Ошибка! Непосредственный доступ к нестатической переменной из
    статического метода недопустим. */
    static int ValDivDenom() return Val/Denom; // не подлежит
компиляции!
}
```

Пример 2:

```
class AnotherStaticError
{
    void NonStaticMeth() // Нестатический метод.
    {
        Console.WriteLine("В методе NonStaticMeth().") ;
    }
    /* Ошибка! Непосредственный вызов нестатического     метода из
статического метода недопустим. */
    static void staticMeth() NonStaticMeth(); // не подлежит компиляции!
}
```

Пример 1:

```
class StaticError
{
    public int Denom = 3; // обычная переменная экземпляра
    public static int Val = 1024; // статическая переменная
    /* Ошибка! Непосредственный доступ к нестатической переменной из
    статического метода недопустим. */
    static int ValDivDenom() return Val/Denom; // не подлежит
компиляции!
}
```

Упражнение 4.3

Реализовать класс в котором используется поле типа static для подсчета экземпляров существующих объектов данного класса.

Пример 1:

```
class StaticError
{
    public int Denom = 3; // обычная переменная экземпляра
    public static int Val = 1024; // статическая переменная
    /* Ошибка! Непосредственный доступ к нестатической переменной из
    статического метода недопустим. */
    static int ValDivDenom() return Val/Denom; // не подлежит
компиляции!
}
```

Упражнение 4.3

Реализовать класс в котором используется поле типа static для подсчета экземпляров существующих объектов данного класса.

Упражнение 4.4

Реализовать статическую фабрику класса.

Общая форма:

```
static class имя_класса { // ...
```

Общая форма:

```
static class имя_класса { // ...
```

Замечание (1)

Объекты статического класса создавать нельзя. Все члены класса должны быть объявлены как static.

Общая форма:

```
static class имя_класса { // ...
```

Замечание (1)

Объекты статического класса создавать нельзя. Все члены класса должны быть объявлены как static.

Замечание (2)

Несмотря на то, что объекты статического класса создавать нельзя у статического класса может быть статический конструктор.

Общая форма:

```
static class имя_класса { // ...
```

Замечание (1)

Объекты статического класса создавать нельзя. Все члены класса должны быть объявлены как static.

Замечание (2)

Несмотря на то, что объекты статического класса создавать нельзя у статического класса может быть статический конструктор.

Упражнение 4.5

Реализовать статический класс MyMath с методами отсутствующими в классе Math. Например, добавить возможность вычисления определенного интеграла, округления вверх и т.п. (3-4 метода)

Общая форма перегрузки унарного оператора

```
public static возвращаемый_тип operator op(тип_параметра операнд)
{ // операции }
```

Общая форма перегрузки унарного оператора

```
public static возвращаемый_тип operator op(тип_параметра операнд)
{ // операции }
```

Общая форма перегрузки бинарного оператора

```
public static возвращаемый_тип operator op(тип_параметра1 операнд1,
тип_параметра2 операнд2)
{ // операции }
```

Общая форма перегрузки унарного оператора

```
public static возвращаемый_тип operator op(тип_параметра операнд)
{ // операции }
```

Общая форма перегрузки бинарного оператора

```
public static возвращаемый_тип operator op(тип_параметра1 операнд1,
тип_параметра2 операнд2)
{ // операции }
```

Замечание

Вместо "op" подставляется перегружаемый оператор, например "+" или "/"

Общая форма перегрузки унарного оператора

```
public static возвращаемый_тип operator op(тип_параметра операнд)
{ // операции }
```

Общая форма перегрузки бинарного оператора

```
public static возвращаемый_тип operator op(тип_параметра1 операнд1,
тип_параметра2 операнд2)
{ // операции }
```

Замечание

Вместо "op" подставляется перегружаемый оператор, например "+" или "/"

Замечание

Тип операнда унарных операторов должен быть таким же, как и у класса, для которого перегружается оператор. А в бинарных операторах хотя бы один из операндов должен быть такого же типа, как и у его класса.

Упражнение 4.6

Реализовать перегруженный бинарный оператор для сложения и вычитания заданных векторов из пространства $\Re^3$

Упражнение 4.6

Реализовать перегруженный бинарный оператор для сложения и вычитания заданных векторов из пространства $\Re^3$

Упражнение 4.7

Реализовать перегруженные операторы инкремента $(++)$ и декремента $(--)$ для заданных векторов из пространства $\Re^3$

Упражнение 4.6

Реализовать перегруженный бинарный оператор для сложения и вычитания заданных векторов из пространства $\Re^3$

Упражнение 4.7

Реализовать перегруженные операторы инкремента $(++)$ и декремента $(--)$ для заданных векторов из пространства $\Re^3$

Упражнение 4.8

Реализовать перегруженный бинарный оператор "+" для сдвига заданного вектора из пространства $\Re^3$ покоординатно на целое число.

Упражнение 4.6

Реализовать перегруженный бинарный оператор для сложения и вычитания заданных векторов из пространства $\Re^3$

Упражнение 4.7

Реализовать перегруженные операторы инкремента $(++)$ и декремента $(--)$ для заданных векторов из пространства $\Re^3$

Упражнение 4.8

Реализовать перегруженный бинарный оператор "+" для сдвига заданного вектора из пространства $\Re^3$ покоординатно на целое число.

Упражнение 4.9

Реализовать перегруженные бинарные операторы отношений ($<$ и $>$) для сравнения двух симметричных, положительно определенных квадратных матриц A и B $\in \Re^{3\times3}$.

Упражнение 4.6

Реализовать перегруженный бинарный оператор для сложения и вычитания заданных векторов из пространства $\Re^3$

Упражнение 4.7

Реализовать перегруженные операторы инкремента $(++)$ и декремента $(--)$ для заданных векторов из пространства $\Re^3$

Упражнение 4.8

Реализовать перегруженный бинарный оператор "+" для сдвига заданного вектора из пространства $\Re^3$ покоординатно на целое число.

Упражнение 4.9

Реализовать перегруженные бинарные операторы отношений ($<$ и $>$) для сравнения двух симметричных, положительно определенных квадратных матриц A и B $\in \Re^{3\times3}$.

Замечание

Операторы отношения должны перегружаться попарно.

Общая форма:

```
public static bool operator op(тип_параметра операнд)
{
// Возврат логического значения true или false.
}
```

Упражнение 4.10

Реализовать перегрузку операторов true и false для заданной матрицы A $\in \Re^{3\times 3}$ (считать (A) имеет значение false если $\det A = 0$).

Общая форма:

```
public static bool operator op(тип_параметра операнд)
{
// Возврат логического значения true или false.
}
```

Упражнение 4.10

Реализовать перегрузку операторов true и false для заданной матрицы A $\in \Re^{3\times 3}$ (считать (A) имеет значение false если $\det A = 0$).

Упражнение 4.11

Реализовать перегрузку операторов &, | и ! для заданных матриц A и B $\in \Re^{3\times 3}$

Четыре правила необходимых для использования && и ||:

- в классе должна быть произведена перегрузка логических операторов & и |.
- перегружаемые методы операторов & и | должны возвращать значение того же типа, что и у класса, для которого эти операторы перегружаются
- каждый параметр должен содержать ссылку на объект того класса, для которого перегружается логический оператор
- для класса должны быть перегружены операторы true и false

Четыре правила необходимых для использования && и ||:

- в классе должна быть произведена перегрузка логических операторов & и |.
- перегружаемые методы операторов & и | должны возвращать значение того же типа, что и у класса, для которого эти операторы перегружаются
- каждый параметр должен содержать ссылку на объект того класса, для которого перегружается логический оператор
- для класса должны быть перегружены операторы true и false

Четыре правила необходимых для использования && и ||:

- в классе должна быть произведена перегрузка логических операторов & и |.
- перегружаемые методы операторов & и | должны возвращать значение того же типа, что и у класса, для которого эти операторы перегружаются
- каждый параметр должен содержать ссылку на объект того класса, для которого перегружается логический оператор
- для класса должны быть перегружены операторы true и false

Четыре правила необходимых для использования && и ||:

- в классе должна быть произведена перегрузка логических операторов & и |.
- перегружаемые методы операторов & и | должны возвращать значение того же типа, что и у класса, для которого эти операторы перегружаются
- каждый параметр должен содержать ссылку на объект того класса, для которого перегружается логический оператор
- для класса должны быть перегружены операторы true и false

Четыре правила необходимых для использования && и ||:

- в классе должна быть произведена перегрузка логических операторов & и |.
- перегружаемые методы операторов & и | должны возвращать значение того же типа, что и у класса, для которого эти операторы перегружаются
- каждый параметр должен содержать ссылку на объект того класса, для которого перегружается логический оператор
- для класса должны быть перегружены операторы true и false

Упражнение 4.12

Переделать упражнения 4.10 и 4.11 с учетом сформулированных выше правил и добиться реализации операторов && и ||

Две формы операторов преобразования:

```
public static explicit operator целевой_тип(исходный_тип v) {return значение;}
public static implicit operator целевой_тип(исходный_тип v) {return значение;}
```

Две формы операторов преобразования:

public static explicit operator целевой_тип(исходный_тип v) {return значение;}
public static implicit operator целевой_тип(исходный_тип v) {return значение;}

Упражнение 4.13

Реализовать явный оператор преобразования матрицы A($\in \Re^{3\times 3}$) из класса матриц в число типа double ($= \det A$).

Две формы операторов преобразования:

public static explicit operator целевой_тип(исходный_тип v) {return значение;}
public static implicit operator целевой_тип(исходный_тип v) {return значение;}

Упражнение 4.13

Реализовать явный оператор преобразования матрицы A($\in \Re^{3\times 3}$) из класса матриц в число типа double ($= \det A$).

Упражнение 4.14

Реализовать кольцо на множестве целых чисел по модулю 7

Общая форма одномерного индексатора:

```
тип_элемента this[int индекс]
{
    // Аксессор для получения данных.
    get
    {
        // Возврат значения, которое определяет индекс.
    }
    // Аксессор для установки данных.
    set
    {
        // Установка значения, которое определяет индекс.
    }
}
```

Общая форма одномерного индексатора:

```
тип_элемента this[int индекс]
{
    // Аксессор для получения данных.
    get
    {
        // Возврат значения, которое определяет индекс.
    }
    // Аксессор для установки данных.
    set
    {
        // Установка значения, которое определяет индекс.
    }
}
```

Упражнение 5.1

Использовать индексатор для создания отказоустойчивого массива.

Общая форма свойства

```
тип имя
{
    get
    {
        // код аксессора для чтения из поля
    }
    set
    {
        // код аксессора для записи в поле
    }
```

Пример:

```
// Простой пример применения свойства.
using System;
class SimpProp
{
    int prop; // поле, управляемое свойством MyProp
    public SimpProp(){ prop = 0; }
    /* Это свойство обеспечивает доступ к закрытой переменной
    экземпляра prop.
    Оно допускает присваивание только положительных значений. */
    public int MyProp
    {
        get
        {
            return prop;
        }
        set
        {
            if(value >= 0) prop = value;
        }
    }
```

Общая форма свойства

```
тип имя
{
    доступ get
    {
        // код аксессора для чтения из поля
    }
    доступ set
    {
        // код аксессора для записи в поле
    }
```

Общая форма свойства

```
тип имя
{
    доступ get
    {
        // код аксессора для чтения из поля
    }
    доступ set
    {
        // код аксессора для записи в поле
    }
```

Автоматически реализуемые свойства

```
тип имя { доступ get;доступ set; }
```

Общая форма свойства

```
тип имя
{
    доступ get
    {
        // код аксессора для чтения из поля
    }
    доступ set
    {
        // код аксессора для записи в поле
    }
```

Автоматически реализуемые свойства

```
тип имя { доступ get;доступ set; }
```

Пример:

```
public int UserCount { get; private set; }
```

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие модификаторам доступа в свойствах:

- действию модификатора доступа подлежит только один аксессор: set или get, но не оба сразу.
- модификатор должен обеспечивать более ограниченный доступ к аксессору, чем доступ на уровне свойства или индексатора
- модификатор доступа нельзя использовать при объявлении аксессора в интерфейсе или же при реализации аксессора, указываемого в интерфейсе

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие модификаторам доступа в свойствах:

- действию модификатора доступа подлежит только один аксессор: set или get, но не оба сразу.
- модификатор должен обеспечивать более ограниченный доступ к аксессору, чем доступ на уровне свойства или индексатора
- модификатор доступа нельзя использовать при объявлении аксессора в интерфейсе или же при реализации аксессора, указываемого в интерфейсе

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие модификаторам доступа в свойствах:

- действию модификатора доступа подлежит только один аксессор: set или get, но не оба сразу.
- модификатор должен обеспечивать более ограниченный доступ к аксессору, чем доступ на уровне свойства или индексатора
- модификатор доступа нельзя использовать при объявлении аксессора в интерфейсе или же при реализации аксессора, указываемого в интерфейсе

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие модификаторам доступа в свойствах:

- действию модификатора доступа подлежит только один аксессор: set или get, но не оба сразу.
- модификатор должен обеспечивать более ограниченный доступ к аксессору, чем доступ на уровне свойства или индексатора
- модификатор доступа нельзя использовать при объявлении аксессора в интерфейсе или же при реализации аксессора, указываемого в интерфейсе

Упражнение 5.2

Заменить в одной из программ из предыдущих упражнений поля(поле) свойствами(свойством). Обосновать целесообразность данной замены.

Упражнение 5.2

Заменить в одной из программ из предыдущих упражнений поля(поле) свойствами(свойством). Обосновать целесообразность данной замены.

Упражнение 5.3

С помощью свойств и индексаторов реализовать класс для вычисления миноров матриц

Общая форма объявления класса наследования

```
class имя_производного_класса : имя_базового_класса
{
// тело класса
}
```

Общая форма объявления класса наследования

```
class имя_производного_класса : имя_базового_класса
{
// тело класса
}
```

Пример:

```
// Класс для двумерных объектов.
class TwoDShape
{
    public double Width;
    public double Height;
    public void ShowDim()
    {
        Console.WriteLine("Ширина и высота равны " + Width + " и "
        + Height);
    }
}
```

Пример:

```
// Класс для прямоугольников, производный от класса TwoDShape.
class Rectangle : TwoDShape
{
     // Возвратить логическое значение true, если
     // прямоугольник является квадратом.
    public bool IsSquare()
    {
        if(Width == Height) return true;
        return false;
    }
     // Возвратить площадь прямоугольника.
    public double Area()
    {
        return Width * Height;
    }
}
```

Пример:

```
// Класс для прямоугольников, производный от класса TwoDShape.
class Rectangle : TwoDShape
{
     // Возвратить логическое значение true, если
     // прямоугольник является квадратом.
    public bool IsSquare()
    {
        if(Width == Height) return true;
        return false;
    }
     // Возвратить площадь прямоугольника.
    public double Area()
    {
        return Width * Height;
    }
}
```

Замечание

Закрытый член класса остается закрытым в своем классе. Он не доступен из кода за пределами своего класса, включая и производные классы.

Пример:

```
//Продемонстрировать применение модификатора доступа protected.
using System;
class B
{
    //члены, закрытые для класса B, но доступные для класса D
    protected int i, j;
}
class D : B
{
    int k; //закрытый член
    public void Setk()
    {
        k = i * j; //члены i и j класса B доступны для класса D
    }
    public void Showk()
    {
        Console.WriteLine(k);
    }
}
```

Общая форма объявления конструктора производного класса

```
конструктор_производного_класса(список_параметров) :
base(список_аргументов)
{
// тело конструктора
}
```

Общая форма объявления конструктора производного класса

```
конструктор_производного_класса(список_параметров) :
base(список_аргументов)
{
// тело конструктора
}
```

Упражнение 5.4

Для любого из ранее реализованных классов создать производный класс по своему усмотрению и сделать для него конструктор общего вида.

Пример:
```
// Пример сокрытия имени с наследственной связью.
using System;
class A { public int i = 0;} // Создать производный класс.
class B : A
{
    new int i; // этот член скрывает член i из класса A
    public B(int b)
    {
        i = b; // член i в классе B
    }
    public void Show()
    {Console.WriteLine("Член i в производном классе: " + i);}
}
class NameHiding
{
    static void Main()
    {
        B ob = new B(2);
        ob.Show();
    }
}
```

Пример: // Пример сокрытия имени с наследственной связью.

```
using System;
class A { public int i = 0;} // Создать производный класс.
class B : A
```

Замечание

Доступ к скрытому имени базового класса осуществляется с помощью ключевого слова base:

base.имя_члена

```
    }
    public void Show()
    {Console.WriteLine("Член i в производном классе: " + i);}
}
class NameHiding
{
    static void Main()
    {
        B ob = new B(2);
        ob.Show();
    }
}
```

Пример: // Пример сокрытия имени с наследственной связью.

```
using System;
class A { public int i = 0;} // Создать производный класс.
class B : A
```

Замечание

Доступ к скрытому имени базового класса осуществляется с помощью ключевого слова base:

base.имя_члена

```
    }
```

Упражнение 5.5

Продемонстрировать порядок вызова конструкторов на примере многоуровневой иерархии производных классов (не менее двух от базового).

```
    static void Main()
    {
        B ob = new B(2);
        ob.Show();
    }
}
```

Пример: //По ссылке на объект базового класса можно обращаться
//к объекту производного класса.

```
using System;
class X { public int a=1; }
class Y : X{ public int b=2; }
class BaseRef
{
  static void Main()
  {
    X x = new X();
    X x2;
    Y y = new Y();
    x2 = x; //верно, т.к. оба объекта относятся к одному и тому же типу
    Console.WriteLine("x2.a: " + x2.a);
    x2 = y; //верно, поскольку класс Y является производным от класса X
    Console.WriteLine("x2.a: " + x2.a);
     //ссылкам на объекты класса X известно только о членах класса X
    x2.a = 19; //верно
    // x2.b = 27;//неверно, поскольку член b отсутствует у класса X
  }
}
```

Пример: //По ссылке на объект базового класса можно обращаться
//к объекту производного класса.

```
using System;
class X { public int a=1; }
```

Упражнение 5.6

Реализовать конструктор производного класса, позволяющий создавать копию объекта того же класса.

```
  {
    X x = new X();
    X x2;
    Y y = new Y();
    x2 = x; //верно, т.к. оба объекта относятся к одному и тому же типу
    Console.WriteLine("x2.a: " + x2.a);
    x2 = y; //верно, поскольку класс Y является производным от класса X
    Console.WriteLine("x2.a: " + x2.a);
     //ссылкам на объекты класса X известно только о членах класса X
    x2.a = 19; //верно
    // x2.b = 27;//неверно, поскольку член b отсутствует у класса X
  }
}
```

Пример:

```
// Продемонстрировать виртуальный метод.
using System;
class Base
{
    // Создать виртуальный метод в базовом классе.
    public virtual void Who()
    {
        Console.WriteLine("Метод Who() в классе Base");
    }
}
class Derivedl : Base
{
    // Переопределить метод Who() в производном классе.
    public override void Who()
    {
        Console.WriteLine("Метод Who() в классе Derivedl");
    }
}
```

Замечание

Выбор виртуального метода который следует вызывать, осуществляется исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения.

Замечание

Выбор виртуального метода который следует вызывать, осуществляется исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения.

Замечание

Свойства и индексаторы также подлежат модификации ключевым словом virtual и переопределению ключевым словом override.

Замечание

Выбор виртуального метода который следует вызывать, осуществляется исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения.

Замечание

Свойства и индексаторы также подлежат модификации ключевым словом virtual и переопределению ключевым словом override.

Упражнение 5.7

Продемонстрировать вызовы виртуальных методов для различных производных классов от общего базового класса.

Замечание

Выбор виртуального метода который следует вызывать, осуществляется исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения.

Замечание

Свойства и индексаторы также подлежат модификации ключевым словом virtual и переопределению ключевым словом override.

Упражнение 5.7

Продемонстрировать вызовы виртуальных методов для различных производных классов от общего базового класса.

Упражнение 5.8

Реализовать виртуальные методы для случая многоуровневой иерархии производных классов.

Замечание

Выбор виртуального метода который следует вызывать, осуществляется исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения.

Замечание

Свойства и индексаторы также подлежат модификации ключевым словом virtual и переопределению ключевым словом override.

Упражнение 5.7

Продемонстрировать вызовы виртуальных методов для различных производных классов от общего базового класса.

Упражнение 5.8

Реализовать виртуальные методы для случая многоуровневой иерархии производных классов.

Упражнение 5.9

Продемонстрировать удобство использования виртуальных методов и производных классов на примере базового класса для 2D фигур и различных производных от него (круги, треугольники, прямоугольники и т.п.).

Общая форма абстрактного класса:

```
abstract class имя_класса
{
    //в классе должен быть хотя бы один абстрактный метод
    abstract тип имя(список_параметров);
    ...
}
```

Замечание

Можно создавать ссылки на объекты абстрактного класса, но объявлять(инициализировать) эти объекты уже нельзя.

Пример 1:

```
sealed class A
{ // ...}
class  B : A // ОШИБКА! Наследовать класс A нельзя
{ // ... }
```

Пример 1:

```
sealed class A
{ // ...}
class  B : A // ОШИБКА! Наследовать класс A нельзя
{ // ... }
```

Пример 2:

```
class B
{
    public virtual void MyMethod()
    { /* ... */ }
}
class D : B
{
    // Здесь герметизируется метод MyMethod() и
    // предотвращается его дальнейшее переопределение.
    sealed public override void MyMethod() { /* ... */ }
}
class X : D
{
    // Ошибка! Метод MyMethod() герметизирован!
    public override void MyMethod() { /* ... */ }
}
```

Таблица 1. Методы класса object

Метод	Назначение
`public virtual bool Equals(object ob)`	Определяет, является ли вызывающий объект таким же, как и объект, доступный по ссылке *ob*
`public static bool Equals(object objA, object objB)`	Определяет, является ли объект, доступный по ссылке *objA*, таким же, как и объект, доступный по ссылке *objB*
`protected Finalize()`	Выполняет завершающие действия перед "сборкой мусора". В C# метод `Finalize()` доступен посредством деструктора
`public virtual int GetHashCode()`	Возвращает хеш-код, связанный с вызывающим объектом
`public Type GetType()`	Получает тип объекта во время выполнения программы
`protected object MemberwiseClone()`	Выполняет неполное копирование объекта, т.е. копируются только члены, но не объекты, на которые ссылаются эти члены
`public static bool ReferenceEquals(obj objA, object objB)`	Определяет, делаются ли ссылки *objA* и *objB* на один и тот же объект
`public virtual string ToString()`	Возвращает строку, которая описывает объект

Таблица 1. Методы класса object

Метод	Назначение
`public virtual bool Equals(object ob)`	Определяет, является ли вызывающий объект таким же, как и объект, доступный по ссылке *ob*
`public static bool Equals(object objA, object objB)`	Определяет, является ли объект, доступный по ссылке *objA*, таким же, как и объект, доступный по ссылке *objB*
p	...у- ...м
p	...м
`GetHashCode()`	объектом
`public Type GetType()`	Получает тип объекта во время выполнения программы
`protected object MemberwiseClone()`	Выполняет неполное копирование объекта, т.е. копируются только члены, но не объекты, на которые ссылаются эти члены
`public static bool ReferenceEquals(obj objA, object objB)`	Определяет, делаются ли ссылки *objA* и *objB* на один и тот же объект
`public virtual string ToString()`	Возвращает строку, которая описывает объект

Упражнение 5.10

Продемонстрировать применение метода ToString() для любого своего класса

Упаковка

Присваивание ссылки на объект класса object переменной типа значения называется упаковкой. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект.

Упаковка

Присваивание ссылки на объект класса object переменной типа значения называется упаковкой. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект.

Распаковка

Распаковка представляет собой процесс извлечения упакованного значения из объекта. Это делается с помощью явного приведения типа ссылки на объект класса object к соответствующему типу значения.

Упаковка

Присваивание ссылки на объект класса object переменной типа значения называется упаковкой. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект.

Пример 1:

```
// Простой пример упаковки и распаковки.
using System;
class BoxingDemo
{
    static void Main()
    {
        int x;
        object obj;
        x = 10;
        obj = x; // упаковать значение переменной x в объект
        int y = (int)obj; // распаковать значение из объекта, доступного по
        // ссылке obj, в переменную типа int
        Console.WriteLine(y);
    }
}
```

Упаковка

Присваивание ссылки на объект класса object переменной типа значения называется упаковкой. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект.

Пример 2:

```
// Благодаря упаковке становится возможным вызов методов по значению!
using System;
class MethOnValue
{
    static void Main()
    {
        Console.WriteLine(10.ToString());
    }
}
```

Упаковка

Присваивание ссылки на объект класса object переменной типа значения называется упаковкой. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект.

Пример 2:

```
// Благодаря упаковке становится возможным вызов методов по значению!
using System;
class MethOnValue
{
    static void Main()
    {
        Console.WriteLine(10.ToString());
    }
}
```

Упражнение 5.11

Использовать класс object для создания массива "обобщенного" типа

Упрощенная форма объявления интерфейса

```
interface имя
{
    возвращаемый_тип имя_метода1(список_параметров);
    возвращаемый_тип имя_метода2(список_параметров);
    //...
    возвращаемый_тип имя_методаN(список_параметров);
}
```

Упрощенная форма объявления интерфейса

```
interface имя
{
    возвращаемый_тип имя_метода1(список_параметров);
    возвращаемый_тип имя_метода2(список_параметров);
    //...
    возвращаемый_тип имя_методаN(список_параметров);
}
```

Пример:

```
public interface ISeries
{
    int GetNext(); // возвратить следующее по порядку число
    void Reset(); // перезапустить
    void SetStart(int x); // задать начальное значение
}
```

Упрощенная форма объявления интерфейса

```
interface имя
{
    возвращаемый_тип имя_метода1(список_параметров);
    возвращаемый_тип имя_метода2(список_параметров);
    //...
    возвращаемый_тип имя_методаN(список_параметров);
}
```

Пример:

```
public interface ISeries
{
    int GetNext(); // возвратить следующее по порядку число
    void Reset(); // перезапустить
    void SetStart(int x); // задать начальное значение
}
```

Замечание

Помимо методов, в интерфейсах можно также указывать свойства, индексаторы и события, но не члены данных. В них нельзя также определить конструкторы, деструкторы или операторные методы.

Общая форма реализации интерфейса в классе

```
class имя_класса : имя_интерфейса
{
    // тело класса
}
```

Общая форма реализации интерфейса в классе

```
class имя_класса : имя_интерфейса
{
    // тело класса
}
```

Замечание

В классе можно наследовать базовый класс и в тоже время реализовать один или более интерфейс. В таком случае имя базового класса должно быть указано перед списком интерфейсов, разделяемых запятой.

Общая форма реализации интерфейса в классе

```
class имя_класса : имя_интерфейса
{
    // тело класса
}
```

Упражнение 6.1

Используя интерфейс ISeries, создать класс, генерирующий последовательный ряд чисел, в котором каждое последующее число на два больше предыдущего. Добавить в этот класс метод GetPrevious().

Общая форма реализации интерфейса в классе

```
class имя_класса : имя_интерфейса
{
    // тело класса
}
```

Упражнение 6.1

Используя интерфейс ISeries, создать класс, генерирующий последовательный ряд чисел, в котором каждое последующее число на два больше предыдущего. Добавить в этот класс метод GetPrevious().

Упражнение 6.2

Используя интерфейс ISeries, реализовать класс, генерирующий ряд простых чисел

Общая форма реализации интерфейса в классе

```
class имя_класса : имя_интерфейса
{
    // тело класса
}
```

Упражнение 6.1

Используя интерфейс ISeries, создать класс, генерирующий последовательный ряд чисел, в котором каждое последующее число на два больше предыдущего. Добавить в этот класс метод GetPrevious().

Упражнение 6.2

Используя интерфейс ISeries, реализовать класс, генерирующий ряд простых чисел

Упражнение 6.3

Продемонстрировать интерфейсные ссылки.

Общий вид интерфейсного свойства

```
тип имя_свойства
{
    get; // аксессор без реализации и модификатора доступа
    set; // аксессор без реализации и модификатора доступа
}
```

Общий вид интерфейсного свойства

```
тип имя_свойства
{
    get; // аксессор без реализации и модификатора доступа
    set; // аксессор без реализации и модификатора доступа
}
```

Общая форма объявления интерфейсного индексатора

```
тип_элемента this[int индекс]
{
    get; // аксессор без реализации и модификатора доступа
    set; // аксессор без реализации и модификатора доступа
}
```

Общий вид интерфейсного свойства

```
тип имя_свойства
{
    get; // аксессор без реализации и модификатора доступа
    set; // аксессор без реализации и модификатора доступа
}
```

Общая форма объявления интерфейсного индексатора

```
тип_элемента this[int индекс]
{
    get; // аксессор без реализации и модификатора доступа
    set; // аксессор без реализации и модификатора доступа
}
```

Упражнение 6.4

Добавить в интерфейс ISeries индексатор только для чтения, возвращающий i-й элемент числового ряда.

Пример:

```
// Пример наследования интерфейсов.
using System;
public interface IA
{
    void Meth1();
} //В базовый интерфейс включен метод Meth1().
//а в производный интерфейс добавлен еще один метод — Meth2().
public interface IB : IA
{
    void Meth2();
}
//В этом классе должны быть реализованы методы интерфейсов IA и IB.
class MyClass : IB
{
    public void Meth1()
    {
        Console.WriteLine("Реализовать метод Meth1().");
    }
    public void Meth2()
    {
        Console.WriteLine("Реализовать метод Meth2().");
    }
}
```

Пример: interface IMyIF

```
{
    int MyMeth(int x);
}
class MyClass : IMyIF
{
     int IMyIF.MyMeth(int x) //явная реализация члена интерфейса
    {
         return x / 3;
    }
    public int MyMethI(int x)
    {
         IMyIF a_ob;
        a_ob = this;
        return a_ob.MyMeth(x); // вызов интерфейсного метода IMyIF
    }
}
```

Пример: interface IMyIF

```
{
    int MyMeth(int x);
}
class MyClass : IMyIF
{
     int IMyIF.MyMeth(int x) //явная реализация члена интерфейса
    {
         return x / 3;
    }
    public int MyMethI(int x)
    {
         IMyIF a_ob;
        a_ob = this;
        return a_ob.MyMeth(x); // вызов интерфейсного метода IMyIF
    }
}
```

Упражнение 6.5

Реализовать интерфейс IEven, в котором объявляются два метода. Один будет служить для определения четности числа, второй — для нечетности. Реализовать этим методы в классе MyClass и продемонстрировать их работу. Один из методов нужно реализовать явно.

Общая форма объявления структуры:

```
struct имя : интерфейсы
{
// объявления членов
}
```

Общая форма объявления структуры:

```
struct имя : интерфейсы
{
// объявления членов
}
```

Замечание (1)

Структура, подобна классу, но относится к типу значения, а не к ссылочному типу данных. Структуры не могут наследовать другие структуры и классы или служить в качестве базовых для других структур и классов. (Разумеется, структуры, как и все остальные типы данных в C#, наследуют класс object.)

Общая форма объявления структуры:

```
struct имя : интерфейсы
{
// объявления членов
}
```

Замечание (1)

Структура, подобна классу, но относится к типу значения, а не к ссылочному типу данных. Структуры не могут наследовать другие структуры и классы или служить в качестве базовых для других структур и классов. (Разумеется, структуры, как и все остальные типы данных в C#, наследуют класс object.)

Замечание (2)

Объект структуры может быть создан с помощью оператора new таким же образом,как и объект класса, но в этом нет особой необходимости. Когда этот оператор не используется, объект по-прежнему создается, хотя и не инициализируется.

Общая форма объявления структуры:

```
struct имя : интерфейсы
{
// объявления членов
}
```

Упражнение 6.6

Придумать и реализовать пример обоснованного использования структуры (вместо класса).

Общая форма объявления перечисления

```
enum имя список_перечисления;
```

Пример:

```
using System;
class EnumDemo
{
    enum Apple{Jonathan, GoldenDel, RedDel, Winesap, Cortland, McIntosh};
    static void Main()
    {
        string[] color = {"красный","желтый","красный","красный",
        "красный","красновато-зеленый"};
        Apple i; // объявить переменную перечислимого типа
        // Использовать переменную i для циклического
        // обращения к членам перечисления.
        for(i = Apple.Jonathan; i <= Apple.McIntosh; i++)
        Console.WriteLine(i + " имеет значение " + (int)i);
        Console.WriteLine();
        // Использовать перечисление для индексирования массива.
        for(i = Apple.Jonathan; i <= Apple.McIntosh; i++)
        Console.WriteLine("Цвет сорта " + i + " — " +
        color[(int)i]);
    }
}
```

Общая форма объявления перечисления

```
enum имя список_перечисления;
```

Общая форма объявления перечисления

```
enum имя список_перечисления;
```

Инициализация перечисления

```
enum Apple Jonathan, GoldenDel, RedDel = 10, Winesap, Cortland,
McIntosh ;
```

Общая форма объявления перечисления

```
enum имя список_перечисления;
```

Инициализация перечисления

```
enum Apple Jonathan, GoldenDel, RedDel = 10, Winesap, Cortland,
McIntosh ;
```

Указание базового типа перечисления

```
enum имя : целочисленный_тип список_перечисления ;
```

Общая форма объявления перечисления

enum имя список_перечисления;

Инициализация перечисления

enum Apple Jonathan, GoldenDel, RedDel = 10, Winesap, Cortland, McIntosh ;

Указание базового типа перечисления

enum имя : целочисленный_тип список_перечисления ;

Упражнение 6.7

Сымитировать управление лентой конвейера. Для этой цели можно создать метод Conveyor(), принимающий в качестве параметров следующие команды: "старт","стоп","вперед"и "назад".

Обработка исключений

Обработка исключительных ситуаций в C# организуется с помощью четырех ключевых слов: try, catch, throw и finally.

Обработка исключений

Обработка исключительных ситуаций в C# организуется с помощью четырех ключевых слов: try, catch, throw и finally.

Общая форма определения блоков try/catch

```
try
{
    // Блок кода, проверяемый на наличие ошибок.
}
catch (ExcepType1 exOb)
{
    // Обработчик исключения типа ExcepType1.
}
catch (ExcepType2 exOb)
{
    // Обработчик исключения типа ExcepType2.
}
```

Обработка исключений

Обработка исключительных ситуаций в C# организуется с помощью четырех ключевых слов: try, catch, throw и finally.

Общая форма определения блоков try/catch

```
try
{
    // Блок кода, проверяемый на наличие ошибок.
}
catch (ExcepType1 exOb)
{
    // Обработчик исключения типа ExcepType1.
}
catch (ExcepType2 exOb)
{
    // Обработчик исключения типа ExcepType2.
}
```

Упражнение 6.8

Продемонстрировать обработку исключения типа IndexOutOfRangeException (выход за границы массива)

Обработка исключений

Обработка исключительных ситуаций в C# организуется с помощью четырех ключевых слов: try, catch, throw и finally.

Перехват всех исключений

```
try
{
    // Блок кода, проверяемый на наличие ошибок.
}
catch
{
    // Обработчик всех исключений.
}
```

Упражнение 6.8

Продемонстрировать обработку исключения типа IndexOutOfRangeException (выход за границы массива)

Обработка исключений

Обработка исключительных ситуаций в C# организуется с помощью четырех ключевых слов: try, catch, throw и finally.

Перехват всех исключений

```
try
{
    // Блок кода, проверяемый на наличие ошибок.
}
catch
{
    // Обработчик всех исключений.
}
```

Упражнение 6.9

Использовать вложенный блок try для контроля неучтенных исключений.

Упражнение 6.8

Продемонстрировать обработку исключения типа IndexOutOfRangeException (выход за границы массива)

Общая форма генерирования исключений

```
throw exceptOb;
```

Общая форма генерирования исключений

```
throw exceptOb;
```

Пример:

```
// Сгенерировать исключение вручную.
using System;
class ThrowDemo
{
    static void Main()
    {
        try
        {
            Console.WriteLine("До генерирования исключения.");
            throw new DivideByZeroException();
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Исключение перехвачено.");
        }
        Console.WriteLine("После пары операторов try/catch.");
    }
}
```

Общая форма совместного использования блоков try/ catch и finally

```
try
{
    // Блок кода, предназначенный для обработки ошибок.
}
catch (ExcepType1 exOb) {
    // Обработчик исключения типа ExcepType1.
}
catch (ExcepType2 exOb)
{
    // Обработчик исключения типа ExcepType2.
}
finally
{
    // Код завершения обработки исключений.
}
```

Класс Exception

В классе Exception определяется ряд свойств. К числу самых интересных относятся три свойства, доступные только для чтения: Message, StackTrace и TargetSite:

а) свойство Message содержит символьную строку, описывающую характер ошибки;

б) свойство StackTrace — строку с вызовами стека, приведшими к исключительной ситуации

в) свойство TargetSite получает объект, обозначающий метод, сгенерировавший исключение

Класс Exception

В классе Exception определяется ряд свойств. К числу самых интересных относятся три свойства, доступные только для чтения: Message, StackTrace и TargetSite:

а) свойство Message содержит символьную строку, описывающую характер ошибки;

б) свойство StackTrace — строку с вызовами стека, приведшими к исключительной ситуации

в) свойство TargetSite получает объект, обозначающий метод, сгенерировавший исключение

Класс Exception

В классе Exception определяется ряд свойств. К числу самых интересных относятся три свойства, доступные только для чтения: Message, StackTrace и TargetSite:

а) свойство Message содержит символьную строку, описывающую характер ошибки;

б) свойство StackTrace — строку с вызовами стека, приведшими к исключительной ситуации

в) свойство TargetSite получает объект, обозначающий метод, сгенерировавший исключение

Класс Exception

В классе Exception определяется ряд свойств. К числу самых интересных относятся три свойства, доступные только для чтения: Message, StackTrace и TargetSite:

а) свойство Message содержит символьную строку, описывающую характер ошибки;

б) свойство StackTrace — строку с вызовами стека, приведшими к исключительной ситуации

в) свойство TargetSite получает объект, обозначающий метод, сгенерировавший исключение

Класс Exception

В классе Exception определяется ряд свойств. К числу самых интересных относятся три свойства, доступные только для чтения: Message, StackTrace и TargetSite:

а) свойство Message содержит символьную строку, описывающую характер ошибки;

б) свойство StackTrace — строку с вызовами стека, приведшими к исключительной ситуации

в) свойство TargetSite получает объект, обозначающий метод, сгенерировавший исключение

Упражнение 6.10

Получить в процессе выполнения программы какое-либо исключение и продемонстрировать для него значения вышеперечисленных свойств (например, со помощью метода WriteLine()).

Конструкторы класса Exception

а) public Exception()

б) public Exception(string сообщение)

в) public Exception(string сообщение, Exception внутреннее_исключение)

г) protected Exception(System.Runtime.Serialization.SerializationInfo информация, System.Runtime.Serialization.StreamingContext контекст)

Конструкторы класса Exception

а) public Exception()

б) public Exception(string сообщение)

в) public Exception(string сообщение, Exception внутреннее_исключение)

г) protected Exception(System.Runtime.Serialization.SerializationInfo информация, System.Runtime.Serialization.StreamingContext контекст)

Конструкторы класса Exception

а) public Exception()

б) public Exception(string сообщение)

в) public Exception(string сообщение, Exception внутреннее_исключение)

г) protected Exception(System.Runtime.Serialization.SerializationInfo информация, System.Runtime.Serialization.StreamingContext контекст)

Конструкторы класса Exception

а) public Exception()

б) public Exception(string сообщение)

в) public Exception(string сообщение, Exception внутреннее_исключение)

г) protected Exception(System.Runtime.Serialization.SerializationInfo информация, System.Runtime.Serialization.StreamingContext контекст)

Конструкторы класса Exception

а) public Exception()

б) public Exception(string сообщение)

в) public Exception(string сообщение, Exception внутреннее_исключение)

г) protected Exception(System.Runtime.Serialization.SerializationInfo информация, System.Runtime.Serialization.StreamingContext контекст)

Таблица 1. Наиболее часто используемые исключения, определенные в пространстве имен System.

Исключение	**Значение**
`ArrayTypeMismatchException`	Тип сохраняемого значения несовместим с типом массива
`DivideByZeroException`	Попытка деления на нуль
`IndexOutOfRangeException`	Индекс оказался за границами массива
`InvalidCastException`	Неверно выполнено динамическое приведение типов
`OutOfMemoryException`	Недостаточно свободной памяти для дальнейшего выполнения программы. Это исключение может быть, например, сгенерировано, если для создания объекта с помощью оператора new не хватает памяти
`OverflowException`	Произошло арифметическое переполнение
`NullReferenceException`	Попытка использовать пустую ссылку, т.е. ссылку, которая не указывает ни на один из объектов

Таблица 1. Наиболее часто используемые исключения, определенные в пространстве имен System.

Исключение	**Значение**
`ArrayTypeMismatchException`	Тип сохраняемого значения несовместим с типом массива
`DivideByZeroException`	Попытка деления на нуль
`IndexOutOfRangeException`	Индекс оказался за границами массива
`InvalidCastException`	Неверно выполнено динамическое приведение типов
`OutOfMemoryException`	Недостаточно свободной памяти для дальнейшего выполнения программы. Это исключение может быть, например, сгенерировано, если для создания объекта с помощью оператора new не хватает памяти
`OverflowException`	Произошло арифметическое переполнение
`NullReferenceException`	Попытка использовать пустую ссылку, т.е. ссылку, которая не указывает ни на один из объектов

Упражнение 6.11

Продемонстрировать обработку исключения NullReferenceException.

Пример:

```
class MyException : Exception
{
  /* Реализовать все конструкторы класса Exception. Такие конструкторы
   просто реализуют конструктор базового класса. А поскольку класс
   исключения MyException ничего не добавляет к классу Exception, то
   никаких дополнительных действий не требуется. */
  public MyException() : base() { }
  public MyException(string str) : base(str) { }
  public MyException(string str, Exception inner) : base(str, inner) { }
  protected MyException(System.Runtime.Serialization.SerializationInfo
  si,System.Runtime.Serialization.StreamingContext sc) : base(si, sc) { }
  //Переопределить метод ToString() для класса исключения MyException.
  public override string ToString()
  {
    return Message;
  }
}
```

Пример:

```
class MyException : Exception
{
  /* Реализовать все конструкторы класса Exception. Такие конструкторы
   просто реализуют конструктор базового класса. А поскольку класс
   исключения MyException ничего не добавляет к классу Exception, то
   никаких дополнительных действий не требуется. */
  public MyException() : base() { }
  public MyException(string str) : base(str) { }
  public MyException(string str, Exception inner) : base(str, inner) { }
  protected MyException(System.Runtime.Serialization.SerializationInfo
  si,System.Runtime.Serialization.StreamingContext sc) : base(si, sc) { }
  //Переопределить метод ToString() для класса исключения MyException.
  public override string ToString()
  {
    return Message;
  }
}
```

Упражнение 6.12

Реализовать класс для создания массива, индексируемого в пределах от -10 до 10. Добавить свое исключение для обработки ошибок при обращении к массиву вне допустимого диапазона.

Замечание

Если требуется перехватывать исключения базового и производного классов, то первым по порядку должен следовать оператор catch, перехватывающий исключение производного класса.

Пример:

```
using System;
class ExceptA : Exception //Создать класс исключения.
{
    public ExceptA(string str) : base(str) { }
    public override string ToString(){ return Message; }
}
class OrderMatters
{
    static void Main()
    {
        for(int x = 0; x < 3; x++)
        {
            try
            {
                if(x==0) throw new ExceptA("Перехват ExceptA");
                else throw new Exception();
            }
            catch (ExceptA exc) {Console.WriteLine(exc);}
            catch (Exception exc) {Console.WriteLine(exc);}
        }
    }
}
```

Две общие формы ключевого слова checked

(1) checked (выражение)

(2) checked

```
{
    // проверяемые операторы
}
```

Две общие формы ключевого слова checked

(1) checked (выражение)

(2)
```
checked
{
    // проверяемые операторы
}
```

Две общие формы ключевого слова unchecked

(1) unchecked (выражение)

(2)
```
unchecked
{
    // проверяемые операторы
}
```

Две общие формы ключевого слова checked

(1) checked (выражение)

(2) checked
```
{
    // проверяемые операторы
}
```

Две общие формы ключевого слова unchecked

(1) unchecked (выражение)

(2) unchecked
```
{
    // проверяемые операторы
}
```

Упражнение 6.13

Написать программу в которой демонстрируется применение ключевых слов checked и unchecked

Таблица 1: Некоторые методы, определенные в классе Stream

Метод	Описание
void Close()	Закрывает поток
void Flush()	Выводит содержимое потока на физическое устройство
int ReadByte()	Возвращает целочисленное представление следующего байта, доступного для ввода из потока. При обнаружении конца файла возвращает значение -1
int Read(byte[] buffer, int offset, int count)	Делает попытку ввести count байтов в массив *buffer*, начиная с элемента *buffer[offset]*. Возвращает количество успешно введенных байтов
long Seek(long offset, SeekOrigin origin)	Устанавливает текущее положение в потоке по указанному смещению *offset* относительно заданного начала отсчета *origin*. Возвращает новое положение в потоке
void WriteByte(byte value)	Выводит один байт в поток вывода
void Write(byte[]buffer, int offset, int count)	Выводит подмножество count байтов из массива *buffer*, начиная с элемента *buffer[offset]*. Возвращает количество выведенных байтов

Таблица 2: Свойства, определенные в классе Stream

Свойство	Описание
bool CanRead	Принимает значение *true*, если из потока можно ввести данные. Доступно только для чтения
bool CanSeek	Принимает значение *true*, если поток поддерживает запрос текущего положения в потоке. Доступно только для чтения
bool CanWrite	Принимает значение *true*, если в поток можно вывести данные. Доступно только для чтения
long Length	Содержит длину потока. Доступно только для чтения
long Position	Представляет текущее положение в потоке. Доступно как для чтения, так и для записи
int ReadTimeout	Представляет продолжительность времени ожидания в операциях ввода. Доступно как для чтения, так и для записи
int WriteTimeout	Представляет продолжительность времени ожидания в операциях вывода. Доступно как для чтения, так и для записи

Таблица 3: Классы байтовых потоков

Класс потока	**Описание**
BufferedStream	Заключает в оболочку байтовый поток и добавляет буферизацию. Буферизация, как правило, повышает производительность
FileStream	Байтовый поток, предназначенный для файлового ввода-вывода
MemoryStream	Байтовый поток, использующий память для хранения данных
UnmanagedMemoryStream	Байтовый поток, использующий неуправляемую память для хранения данных

Таблица 4: Методы ввода, определенные в классе TextReader

Метод	Описание
int Peek()	Получает следующий символ из потока ввода, но не удаляет его. Возвращает значение -1, если ни один из символов не доступен
int Read()	Возвращает целочисленное представление следующего доступного символа из вызывающего потока ввода. При обнаружении конца потока возвращает значение -1
int Read(char[]buffer, int index, int count)	Делает попытку ввести количество count символов в массив buffer, начиная с элемента buffer[index], и возвращает количество успешно введенных символов
int ReadBlock(char[]buffer, int index, int count)	Делает попытку ввести количество count символов в массив buffer, начиная с элемента buffer[index], и возвращает количество успешно введенных символов
string ReadLine()	Вводит следующую текстовую строку и возвращает ее в виде объекта типа string. При попытке прочитать признак конца файла возвращает пустое значение
string ReadToEnd()	Вводит все символы, оставшиеся в потоке, и возвращает их в виде объекта типа string

Таблица 5: Методы класса TextWriter

Метод	Описание
void Write(int value)	Выводит значение типа int
void Write(double value)	Выводит значение типа double
void Write(bool value)	Выводит значение типа bool
void WriteLine(string value)	Выводит значение типа string с последующим символом новой строки
void WriteLine(uint value)	Выводит значение типа uint с последующим символом новой строки
void WriteLine(char value)	Выводит символ с последующим символом новой строки

Таблица 6: Классы символьных потоков, связанные с TextReader и TextWriter

Класс потока	Описание
StreamReader	Предназначен для ввода символов из байтового потока. Этот класс является оболочкой для байтового потока ввода
StreamWriter	Предназначен для вывода символов в байтовый поток. Этот класс является оболочкой для байтового потока вывода
StringReader	Предназначен для ввода символов из символьной строки
StringWriterм	Предназначен для вывода символов в символьную строку

метод Read()

Для чтения одного символа служит метод Read():

```
static int Read()
```

метод Read()

Для чтения одного символа служит метод Read():

```
static int Read()
```

метод ReadLine()

Для считывания строки символов служит метод ReadLine():

```
static string ReadLine()
```

метод Read()

Для чтения одного символа служит метод Read():
```
static int Read()
```

метод ReadLine()

Для считывания строки символов служит метод ReadLine():
```
static string ReadLine()
```

Две формы объявления метода ReadKey().

Для считывания отдельно введенного с клавиатуры символа без построчной буферизации служит метод ReadKey():
```
static ConsoleKeyInfo ReadKey()
static ConsoleKeyInfo ReadKey( bool intercept)
```

метод Read()

Для чтения одного символа служит метод Read():
static int Read()

метод ReadLine()

Для считывания строки символов служит метод ReadLine():
static string ReadLine()

Две формы объявления метода ReadKey().

Для считывания отдельно введенного с клавиатуры символа без построчной буферизации служит метод ReadKey():
static ConsoleKeyInfo ReadKey()
static ConsoleKeyInfo ReadKey(bool intercept)

Замечание

Структура ConsoleKeyInfo состоит из следующих свойств, доступных только для чтения
char KeyChar
ConsoleKey Key
ConsoleModifiers Modifiers

метод Read()

Для чтения одного символа служит метод Read():
```
static int Read()
```

метод ReadLine()

Для считывания строки символов служит метод ReadLine():
```
static string ReadLine()
```

Две формы объявления метода ReadKey().

Для считывания отдельно введенного с клавиатуры символа без построчной буферизации служит метод ReadKey():
```
static ConsoleKeyInfo ReadKey()
static ConsoleKeyInfo ReadKey( bool intercept)
```

Упражнение 7.1

Написать программу для считывания символов введенных с консоли, используя метод ReadKey() и свойства KeyChar, Key и Modifiers

Распространенный конструктор класса FileStream

FileStream(string путь, FileMode режим)
путь обозначает имя открываемого файла
режим — порядок открытия файла.

Распространенный конструктор класса FileStream

FileStream(string путь, FileMode режим)
путь обозначает имя открываемого файла
режим — порядок открытия файла.

Таблица 7: Значения из перечисления FileMode

Значение	Описание
FileMode.Append	Добавляет выводимые данные в конец файла
FileMode.Create	Создает новый выходной файл. Существующий файл с таким же именем будет разрушен
FileMode.CreateNew	Создает новый выходной файл. Файл с таким же именем не должен существовать
FileMode.Open	Открывает существующий файл
FileMode.OpenOrCreate	Открывает файл, если он существует. В противном случае создает новый файл
FileMode.Truncate	Открывает существующий файл, но сокращает его длину до нуля

Распространенный конструктор класса FileStream

FileStream(string путь, FileMode режим, FileAccess доступ)
путь обозначает имя открываемого файла
режим — порядок открытия файла.
доступ — указывается одно из значений FileAccess.Read, FileAccess.Write, FileAccess.ReadWrite

Распространенный конструктор класса FileStream

FileStream(string путь, FileMode режим, FileAccess доступ)
путь обозначает имя открываемого файла
режим – порядок открытия файла.
доступ – указывается одно из значений FileAccess.Read, FileAccess.Write, FileAccess.ReadWrite

Пример:

```
// файл открывается только для чтения
FileStream fin = new FileStream("test.dat" , FileMode.Open,
FileAccess.Read);
```

Распространенный конструктор класса FileStream

FileStream(string путь, FileMode режим, FileAccess доступ)
путь обозначает имя открываемого файла
режим — порядок открытия файла.
доступ — указывается одно из значений FileAccess.Read, FileAccess.Write, FileAccess.ReadWrite

Метод для чтения одного байта из файла

int ReadByte()
Всякий раз, когда этот метод вызывается, из файла считывается один байт, который затем возвращается в виде целого значения.

Распространенный конструктор класса FileStream

FileStream(string путь, FileMode режим, FileAccess доступ)
путь обозначает имя открываемого файла
режим — порядок открытия файла.
доступ — указывается одно из значений FileAccess.Read, FileAccess.Write, FileAccess.ReadWrite

Метод для чтения одного байта из файла

int ReadByte()
Всякий раз, когда этот метод вызывается, из файла считывается один байт, который затем возвращается в виде целого значения.

Метод для чтения блока байтов из файла

int Read(byte[ ] array, int offset, int count)
В данном методе предпринимается попытка считать количество count байтов в массив array, начиная с элемента array[offset]. В случае успеха, возвращается количество байтов считанных из файла.

Распространенный конструктор класса FileStream

FileStream(string путь, FileMode режим, FileAccess доступ)
путь обозначает имя открываемого файла
режим — порядок открытия файла.
доступ — указывается одно из значений FileAccess.Read, FileAccess.Write, FileAccess.ReadWrite

Упражнение 7.2

Написать программу для отображения текстового файла. В случае, если файл не удастся открыть, выдать сообщение об ошибке.

Распространенный конструктор класса FileStream

FileStream(string путь, FileMode режим, FileAccess доступ)
путь обозначает имя открываемого файла
режим — порядок открытия файла.
доступ — указывается одно из значений FileAccess.Read, FileAccess.Write, FileAccess.ReadWrite

Упражнение 7.2

Написать программу для отображения текстового файла. В случае, если файл не удастся открыть, выдать сообщение об ошибке.

Замечание

По завершении работы с файлом его следует закрыть, вызвав метод Close()

Распространенный конструктор класса FileStream

FileStream(string путь, FileMode режим, FileAccess доступ)
путь обозначает имя открываемого файла
режим — порядок открытия файла.
доступ — указывается одно из значений FileAccess.Read, FileAccess.Write, FileAccess.ReadWrite

Упражнение 7.2

Написать программу для отображения текстового файла. В случае, если файл не удастся открыть, выдать сообщение об ошибке.

Упражнение 7.3

Написать программу для записи данных в файл

Распространенный конструктор класса FileStream

FileStream(string путь, FileMode режим, FileAccess доступ)
путь обозначает имя открываемого файла
режим — порядок открытия файла.
доступ — указывается одно из значений FileAccess.Read, FileAccess.Write, FileAccess.ReadWrite

Упражнение 7.2

Написать программу для отображения текстового файла. В случае, если файл не удастся открыть, выдать сообщение об ошибке.

Упражнение 7.3

Написать программу для записи данных в файл

Упражнение 7.4

Написать программу для копирования данных из одного файла в другой

Конструкторы класса StreamWriter

StreamWriter(Stream поток)
StreamWriter(string путь)
StreamWriter(string путь, bool append)
(Если значение параметра append равно true, то выводимые данные присоединяются в конец существующего файла.)

Конструкторы класса StreamWriter

StreamWriter(Stream поток)
StreamWriter(string путь)
StreamWriter(string путь, bool append)
(Если значение параметра append равно true, то выводимые данные присоединяются в конец существующего файла.)

Пример:

```
FileStream file_out;
file_out = new FileStream("test.txt" , FileMode.Create);
StreamWriter file_str_out = new StreamWriter( file_out);
```

Конструкторы класса StreamWriter

StreamWriter(Stream поток)
StreamWriter(string путь)
StreamWriter(string путь, bool append)
(Если значение параметра append равно true, то выводимые данные присоединяются в конец существующего файла.)

Пример:

```
FileStream file_out;
file_out = new FileStream("test.txt" , FileMode.Create);
StreamWriter file_str_out = new StreamWriter( file_out);
```

Упражнение 7.5

Сохранить текст, введенный с клавиатуры, в файл test.txt с помощью символьного потока. Набираемый тест вводится до тех пор, пока в нем не встретится строка "стоп". Использовать различные конструкторы класса StreamWriter

Конструкторы класса StreamReader

StreamReader(Stream поток)
StreamReader(string путь)

Конструкторы класса StreamReader

StreamReader(Stream поток)
StreamReader(string путь)

Замечание

Для обнаружения конца потока в классе StreamReader можно использовать свойство EndOfStream. Это доступное для чтения свойство имеет логическое значение true, когда достигается конец потока, в противном случае — логическое значение false.

Конструкторы класса StreamReader

StreamReader(Stream поток)
StreamReader(string путь)

Замечание

Для обнаружения конца потока в классе StreamReader можно использовать свойство EndOfStream. Это доступное для чтения свойство имеет логическое значение true, когда достигается конец потока, в противном случае — логическое значение false.

Упражнение 7.6

Вывести на экран содержимое текстового файла "Test.txt", созданного в предыдущем примере. Использовать различные конструкторы класса StreamReader

Переадресация стандартных потоков

Для переадресации стандартных потоков в классе Console служат методы SetIn(), SetOut() и SetError():

```
static void SetIn( TextReader новый_поток_ввода)
static void SetOut( TextWriter новый_поток_вывода)
static void SetError( TextWriter новый_поток_сообщений_об_ошибках)
```

Переадресация стандартных потоков

Для переадресации стандартных потоков в классе Console служат методы SetIn(), SetOut() и SetError():

```
static void SetIn( TextReader новый_поток_ввода)
static void SetOut( TextWriter новый_поток_вывода)
static void SetError( TextWriter новый_поток_сообщений_об_ошибках)
```

Упражнение 7.7

Переадресовать вывод потока Console.Out с экрана в файл "logfile.txt".

Переадресация стандартных потоков

Для переадресации стандартных потоков в классе Console служат методы SetIn(), SetOut() и SetError():

```
static void SetIn( TextReader новый_поток_ввода)
static void SetOut( TextWriter новый_поток_вывода)
static void SetError( TextWriter новый_поток_сообщений_об_ошибках)
```

Упражнение 7.7

Переадресовать вывод потока Console.Out с экрана в файл "logfile.txt".

Замечание

При выполнении упражнения 7.7 нужно учитывать, что класс StreamWriter (StreamReader) является производным от класса TextWriter (TextReader).

Стандартный конструктор класса BinaryWriter

BinaryWriter(Stream output)
output обозначает поток, в который выводятся записываемые данные. Для записи в выходной файл в качестве параметра output может быть указан объект, создаваемый средствами класса FileStream.

Таблица 8: Наиболее часто используемые методы класса BinaryWriter

Метод	Описание
void Write(sbyte value)	Записывает значение типа sbyte со знаком
void Write(byte value)	Записывает значение типа byte без знака
void Write(byte[] buffer)	Записывает массив значений типа byte
void Write(short value)	Записывает целочисленное значение типа short (короткое целое)
void Write(ushort value)	Записывает целочисленное значение типа ushort (короткое целое без знака)
void Write(int value)	Записывает целочисленное значение типа
void Write(uint value)	Записывает целочисленное значение типа uint(целое без знака)
void Write(long value)	Записывает целочисленное значение типа long (длинное целое)
void Write(ulong value)	Записывает целочисленное значение типа ulong (длинное целое без знака)
void Write(float value)	Записывает значение типа float (с плавающей точкой одинарной точности)
void Write(double value)	Записывает значение типа double (с плавающей точкой двойной точности)
void Write(decimal value)	Записывает значение типа decimal (с двумя десятичными разрядами после запятой)
void Write(char ch)	Записывает символ
void Write(char[] buffer)	Записывает массив символов
void Write(string value)	Записывает строковое значение типа string, представленное во внутреннем формате с указанием длины строки

Стандартный конструктор класса BinaryReader

BinaryReader(Stream input)
input обозначает поток, из которого вводятся считываемые данные. Для чтения из входного файла в качестве параметра input может быть указан объект, создаваемый средствами класса FileStream.

Таблица 9: Наиболее часто используемые методы класса BinaryReader

Метод	Описание
bool ReadBoolean()	Считывает значение логического типа bool
byte ReadByte()	Считывает значение типа byte
sbyte ReadSByte()	Считывает значение типа sbyte
byte[] ReadBytes(int count)	Считывает количество count байтов и возвращает их в виде массива
char ReadChar()	Считывает значение типа char
char[] ReadChars(int count)	Считывает количество count символов и возвращает их в виде массива
decimal ReadDecimal()	Считывает значение типа decimal
double ReadDouble()	Считывает значение типа double
float ReadSingle()	Считывает значение типа float
short ReadInt16()	Считывает значение типа short
int ReadInt32()	Считывает значение типа int
long ReadInt64()	Считывает значение типа long
ushort ReadUIntl6()	Считывает значение типа ushort
uint ReadUInt32()	Считывает значение типа uint
ulong ReadUInt64()	Считывает значение типа ulong
string ReadString()	Считывает значение типа string, представленное во внутреннем двоичном формате с указанием длины строки. Этот метод следует использовать для считывания строки, которая была записана средствами класса BinaryWriter

Замечание

В классе BinaryReader определены также три приведенных ниже варианта метода Read() и стандартный метод Close().

Замечание

В классе BinaryReader определены также три приведенных ниже варианта метода Read() и стандартный метод Close().

Таблица 10: Варианта метода Read() класса BinaryReader

Метод	Описание
int Read()	Возвращает целочисленное представление следующего доступного символа из вызывающего потока ввода. При обнаружении конца файла возвращает значение -1
int Read(byte[] buffer, int offset, int count)	Делает попытку прочитать количество count байтов в массив buffer, начиная с элемента buffer[offset], и возвращает количество успешно считанных байтов
int Read(char[]buffer, int offset, int count)	Делает попытку прочитать количество count символов в массив buffer, начиная с элемента buffer[offset], и возвращает количество успешно считанных символов

Замечание

В классе BinaryReader определены также три приведенных ниже варианта метода Read() и стандартный метод Close().

Упражнение 7.8

Используя классы BinaryReader и BinaryWriter записать в файл данные разных типов и считать их обратно.

Замечание

В классе BinaryReader определены также три приведенных ниже варианта метода Read() и стандартный метод Close().

Упражнение 7.8

Используя классы BinaryReader и BinaryWriter записать в файл данные разных типов и считать их обратно.

Упражнение 7.9

Использовать классы BinaryReader и BinaryWriter для реализации программы по учету товаров на складе. Вначале вводятся наименования товаров их количество и цена. Данные записываются в файл. Далее по ключевому слову осуществляется поиск в файле, после чего выводится информация о товаре(кол-во на складе, цена за штуку, общая цена всей партии)

Замечание

В классе BinaryReader определены также три приведенных ниже варианта метода Read() и стандартный метод Close().

Упражнение 7.8

Используя классы BinaryReader и BinaryWriter записать в файл данные разных типов и считать их обратно.

Упражнение 7.9

Использовать классы BinaryReader и BinaryWriter для реализации программы по учету товаров на складе. Вначале вводятся наименования товаров их количество и цена. Данные записываются в файл. Далее по ключевому слову осуществляется поиск в файле, после чего выводится информация о товаре(кол-во на складе, цена за штуку, общая цена всей партии)

Замечание

Для проверки достижения конца файла можно использовать исключение EndOfStreamException

Общая форма метода Seek() класса FileStream:

long Seek(long offset, SeekOrigin origin)
offset обозначает новое положение указателя файла в байтах относительно заданного начала отсчета (origin). В качестве origin может быть указано одно из приведенных ниже значений, определяемых в перечислении SeekOrigin

Таблица 11: Перечисление SeekOrigin

Значение	Описание
SeekOrigin.Begin	Поиск от начала файла
SeekOrigin.Current	Поиск от текущего положения
SeekOrigin.End	Поиск от конца файла

Общая форма метода Seek() класса FileStream:

long Seek(long offset, SeekOrigin origin)
offset обозначает новое положение указателя файла в байтах относительно заданного начала отсчета (origin). В качестве origin может быть указано одно из приведенных ниже значений, определяемых в перечислении SeekOrigin

Таблица 11: Перечисление SeekOrigin

Значение	Описание
SeekOrigin.Begin	Поиск от начала файла
SeekOrigin.Current	Поиск от текущего положения
SeekOrigin.End	Поиск от конца файла

Упражнение 7.9

Записать в файл буквы английского алфавита, а затем вывести их на экран в случайном порядке с помощью метода Seek().

Общая форма метода Seek() класса FileStream:

long Seek(long offset, SeekOrigin origin)
offset обозначает новое положение указателя файла в байтах относительно заданного начала отсчета (origin). В качестве origin может быть указано одно из приведенных ниже значений, определяемых в перечислении SeekOrigin

Таблица 11: Перечисление SeekOrigin

Значение	Описание
SeekOrigin.Begin	Поиск от начала файла
SeekOrigin.Current	Поиск от текущего положения
SeekOrigin.End	Поиск от конца файла

Упражнение 7.9

Записать в файл буквы английского алфавита, а затем вывести их на экран в случайном порядке с помощью метода Seek().

Упражнение 7.10

Реализовать программу из предыдущего примера используя свойство Position (см. таблицу 2).

Класс MemoryStream

Для ввода-вывода данных в(из) массив(а) служит класс MemoryStream. В данном классе определено несколько конструкторов. Вот один из них:
MemoryStream(byte[] buffer)

Класс MemoryStream

Для ввода-вывода данных в(из) массив(а) служит класс MemoryStream. В данном классе определено несколько конструкторов. Вот один из них:
MemoryStream(byte[] buffer)

```
Пример: byte[] storage = new byte[255];
// Создать запоминающий поток.
MemoryStream memstrm = new MemoryStream(storage);
```

Класс MemoryStream

Для ввода-вывода данных в(из) массив(а) служит класс MemoryStream. В данном классе определено несколько конструкторов. Вот один из них:
MemoryStream(byte[] buffer)

Пример:
```
byte[] storage = new byte[255];
// Создать запоминающий поток.
MemoryStream memstrm = new MemoryStream(storage);
```

Упражнение 7.11

Заполнить массив storage своими данными и продемонстрировать применение класса MemoryStream. Использовать два подхода: отобразить содержимое массива storage непосредственно и средствами ввода данных из потока (методы класса StreamReader)

Класс MemoryStream

Для ввода-вывода данных в(из) массив(а) служит класс MemoryStream. В данном классе определено несколько конструкторов. Вот один из них:
MemoryStream(byte[] buffer)

Пример:
```
byte[] storage = new byte[255];
// Создать запоминающий поток.
MemoryStream memstrm = new MemoryStream(storage);
```

Упражнение 7.11

Заполнить массив storage своими данными и продемонстрировать применение класса MemoryStream. Использовать два подхода: отобразить содержимое массива storage непосредственно и средствами ввода данных из потока (методы класса StreamReader)

Замечание

Отметим, что данные зачастую записываются в объект назначения не сразу. Вместо этого они буферизуются на уровне операционной системы до тех пор, пока не накопится достаточный объем данных, чтобы записать их сразу одним блоком. Если данные требуется записать без предварительного накопления в буфере, то для этой цели можно вызвать метод Flush().

Класс File

В классе File имеется ряд удобных методов для чтения из файлов и записи в них, открытия файла и получения ссылки типа FileStream на него. Рассмотрим три из них:

Класс File

В классе File имеется ряд удобных методов для чтения из файлов и записи в них, открытия файла и получения ссылки типа FileStream на него. Рассмотрим три из них:

Метод Copy()

```
static void Copy (string имя_исходн_файла, string имя_целевого_файла)
static void Copy (string имя_исходн_файла, string имя_целевого_файла,
boolean overwrite)
```

Класс File

В классе File имеется ряд удобных методов для чтения из файлов и записи в них, открытия файла и получения ссылки типа FileStream на него. Рассмотрим три из них:

Метод Copy()

static void Copy (string имя_исходн_файла, string имя_целевого_файла)
static void Copy (string имя_исходн_файла, string имя_целевого_файла, boolean overwrite)

Метод Exists()

Метод Exists() определяет, существует ли файл
static bool Exists(string путь)

Класс File

В классе File имеется ряд удобных методов для чтения из файлов и записи в них, открытия файла и получения ссылки типа FileStream на него. Рассмотрим три из них:

Метод Copy()

static void Copy (string имя_исходн_файла, string имя_целевого_файла)
static void Copy (string имя_исходн_файла, string имя_целевого_файла, boolean overwrite)

Метод Exists()

Метод Exists() определяет, существует ли файл
static bool Exists(string путь)

Метод GetLastAccessTime()

Метод GetLastAccessTime() возвращает дату и время последнего доступа к файлу
static DateTime GetLastAccessTime(string путь)

Класс File

В классе File имеется ряд удобных методов для чтения из файлов и записи в них, открытия файла и получения ссылки типа FileStream на него. Рассмотрим три из них:

Упражнение 7.12

Применить методы Copy(), Exists() и GetLastAccessTime()

Класс File

В классе File имеется ряд удобных методов для чтения из файлов и записи в них, открытия файла и получения ссылки типа FileStream на него. Рассмотрим три из них:

Упражнение 7.12

Применить методы Copy(), Exists() и GetLastAccessTime()

Замечание

Для выполнения нескольких операций над одним и тем же файлом лучше воспользоваться классом FileInfo т.к., в отличии от класса File, он предоставляет методы экземпляра и свойства, а не статические методы.

Метод Parse()

осуществляет преобразование числовых строк в их внутреннее представление.

Метод Parse()

осуществляет преобразование числовых строк в их внутреннее представление.

Таблица 12: Структуры и методы преобразования

Структура	**Метод преобразования**
Decimal	static decimal Parse(string s)
Double	static double Parse(string s)
Single	static float Parse(string s)
Int64	static long Parse(string s)
Int32	static int Parse(string s)
Intl6	static short Parse(string s)
UInt64	static ulong Parse(string s)
UInt32	static uint Parse(string s)
UInt16	static ushort Parse(string s)
Byte	static byte Parse(string s)
Sbyte	static sbyte Parse(string s)

Метод Parse()

осуществляет преобразование числовых строк в их внутреннее представление.

Замечание

Для того чтобы избежать генерирования исключений при преобразовании числовых строк, можно воспользоваться методом TryParse().

Метод Parse()

осуществляет преобразование числовых строк в их внутреннее представление.

Замечание

Для того чтобы избежать генерирования исключений при преобразовании числовых строк, можно воспользоваться методом TryParse().

Пример:

```
static bool TryParse(string s, out int результат)
```

Общая форма объявления делегата

```
delegate возвращаемый_тип имя(список_параметров);
```

Общая форма объявления делегата

```
delegate возвращаемый_тип имя(список_параметров);
```

Пример 1:

```
using System; // Объявить тип делегата.
delegate string StrMod(string str);
class DelegateTest
{
    static string Reverse(string s){ ... }
    static string RemoveSpaces(string s){ ... }
    static void Main()
    {
        // Сконструировать делегат.
        StrMod strOp = new StrMod(Reverse);
        Console.WriteLine("Результат:" + strOp("Это простой тест."));
        StrMod strOp = new StrMod(RemoveSpaces);
        Console.WriteLine("Результат:" + strOp("Это простой тест."));
    }
}
```

Общая форма объявления делегата

```
delegate возвращаемый_тип имя(список_параметров);
```

Пример 2:

```
using System; // Объявить тип делегата.
delegate string StrMod(string str);
class DelegateTest
{
    static string Reverse(string s){ ... }
    static string RemoveSpaces(string s){ ... }
    static void Main()
    {
        // Сконструировать делегат.
        StrMod strOp = Reverse;
        Console.WriteLine("Результат:" + strOp("Это простой тест."));
        StrMod strOp = RemoveSpaces;
        Console.WriteLine("Результат:" + strOp("Это простой тест."));
    }
}
```

Общая форма объявления делегата

```
delegate возвращаемый_тип имя(список_параметров);
```

Пример 3:

```
using System; // Объявить тип делегата.
delegate string StrMod(string str);
class StringOps
{
    string Reverse(string s){ ... }
    string RemoveSpaces(string s){ ... }
}
class DelegateTest
    static void Main()
    {
        StringOps so = new StringOps();
        // Инициализировать делегат.
        StrMod strOp = so.Reverse;
        Console.WriteLine("Результат:" + strOp("Это простой тест."));
        StrMod strOp = so.RemoveSpaces;
        Console.WriteLine("Результат:" + strOp("Это простой тест."));
    }
}
```

Групповая адресация

Групповая адресация — это возможность создать список, или цепочку вызовов, для методов, которые вызываются автоматически при обращении к делегату. Для добавления метода в цепочку служит оператор $+$ или $+=$. Для удаления метода из цепочки служит оператор $-$ или $-=$.

Пример 4:

```
using System; // Объявить тип делегата.
delegate string StrMod(ref string str);
class MultiCastDemo
{
    static string Reverse(ref string s){ ... }
    static string RemoveSpaces(ref string s){ ... }
    static void Main()
    {
        string str = "Это простой тест.";
        // Сконструировать делегаты.
        StrMod strOp;
        StrMod reverseStr = Reverse;
        StrMod removeSp = RemoveSpaces;
        // Организовать групповую адресацию.
        strOp = reverseStr;
        strOp += removeSp;
        Console.WriteLine("Результат:" + strOp(str));
    }
}
```

Пример 4:

```
using System; // Объявить тип делегата.
delegate string StrMod(ref string str);
class MultiCastDemo
{
    static string Reverse(ref string s){ ... }
    static string RemoveSpaces(ref string s){ ... }
    static void Main()
```

Упражнение 8.1

Реализовать методы Reverse() и RemoveSpaces() в примерах 1-4. Добавить 1-2 своих метода. Продемонстрировать применение делегатов для этих примеров.

```
        StrMod removeSp = RemoveSpaces;
        // Организовать групповую адресацию.
        strOp = reverseStr;
        strOp += removeSp;
        Console.WriteLine("Результат:" + strOp(str));
    }
}
```

Ковариантность

Ковариантность позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата.

Ковариантность

Ковариантность позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата.

Контравариантность

Контравариантность позволяет присвоить делегату метод, типом параметра которого служит класс, являющийся базовым для класса, указываемого в объявлении делегата

Пример:

```
using System;
class A {...}
//Класс B, производный от класса A.
class B : A {...}
//Этот делегат возвращает объект класса A и
//принимает объект класса B в качестве аргумента.
delegate A ChangeIt(B obj);
class CoContraVariance
{
    static A IncrA(A obj) {...}
    static B IncrB(B obj) {...}
    static void Main()
    {
        B Bob = new B();
        ChangeIt change = IncrA;
        A Aob = change(Bob);
        change = IncrB;
        Bob = (B) change (Bob);
    }
}
```

Пример:

```
using System;
class A {...}
//Класс B, производный от класса A.
class B : A {...}
//Этот делегат возвращает объект класса A и
//принимает объект класса B в качестве аргумента.
delegate A ChangeIt(B obj);
```

Упражнение 8.2

Продемонстрировать свойства ковариантности и контравариантности делегатов на примере своих классов и методов.

```
    static void Main()
    {
        B Bob = new B();
        ChangeIt change = IncrA;
        A Aob = change(Bob);
        change = IncrB;
        Bob = (B) change (Bob);
    }
}
```

Общий вид

```
delegate возвращаемый_тип имя_делегата(список_параметров);
class имя_класса {
static void Main()
{
имя_делегата имя_экземпляра = delegate(список_параметров)
{ ... }; // обратите внимание на точку с запятой после "}"
// вызов анонимного метода
имя_экземпляра(список_параметров);
}
```

Общий вид

```
delegate возвращаемый_тип имя_делегата(список_параметров);
class имя_класса {
static void Main()
{
имя_делегата имя_экземпляра = delegate(список_параметров)
{ ... }; // обратите внимание на точку с запятой после "}"
// вызов анонимного метода
имя_экземпляра(список_параметров);
}
```

Замечание

Локальная переменная, в область действия которой входит анонимный метод, называется внешней переменной. Такие переменные доступны для использования в анонимном методе. И в этом случае внешняя переменная считается захваченной. Захваченная переменная существует до тех пор, пока захвативший ее делегат не будет собран в "мусор".

Общий вид

```
delegate возвращаемый_тип имя_делегата(список_параметров);
class имя_класса {
static void Main()
{
имя_делегата имя_экземпляра = delegate(список_параметров)
{ ... }; // обратите внимание на точку с запятой после "}"
// вызов анонимного метода
имя_экземпляра(список_параметров);
}
```

Замечание

Локальная переменная, в область действия которой входит анонимный метод, называется внешней переменной. Такие переменные доступны для использования в анонимном методе. И в этом случае внешняя переменная считается захваченной. Захваченная переменная существует до тех пор, пока захвативший ее делегат не будет собран в "мусор".

Упражнение 8.3

Продемонстрировать применение захваченной переменной.

Общая форма одиночного лямбда-выражения

(список_параметров) => выражение

Общая форма одиночного лямбда-выражения

(список_параметров) => выражение

Пример I:

```
count => count + 2 //значение параметра count увеличивается на 2.
```

Общая форма одиночного лямбда-выражения

(список_параметров) => выражение

Пример I:

```
count => count + 2 //значение параметра count увеличивается на 2.
```

Пример II:

```
n => n % 2 == 0 //выражение возвращает логическое значение true, если n четное, а иначе — false.
```

Общая форма одиночного лямбда-выражения

(список_параметров) => выражение

Пример I:

```
count => count + 2 //значение параметра count увеличивается на 2.
```

Пример II:

```
n => n % 2 == 0 //выражение возвращает логическое значение true, если n четное, а иначе — false.
```

Пример III:

```
(low, high, val) => val >= low && val <= high;
//выражение возвращает логическое значение true, если val внутри границ диапазона.
```

Общая форма одиночного лямбда-выражения

(список_параметров) => выражение

Пример I:

```
count => count + 2 //значение параметра count увеличивается на 2.
```

Пример II:

```
n => n % 2 == 0 //выражение возвращает логическое значение true, если n четное, а иначе — false.
```

Пример III:

```
(low, high, val) => val >= low && val <= high;
//выражение возвращает логическое значение true, если val внутри границ диапазона.
```

Упражнение 8.4

Использовать делегаты для применения описанных выше лямбда-выражений.

Общая форма одиночного лямбда-выражения

(список_параметров) => выражение

Пример I:

```
count => count + 2 //значение параметра count увеличивается на 2.
```

Пример II:

```
n => n % 2 == 0 //выражение возвращает логическое значение true, если n четное, а иначе — false.
```

Пример III:

```
(low, high, val) => val >= low && val <= high;
//выражение возвращает логическое значение true, если val внутри границ диапазона.
```

Упражнение 8.4

Использовать делегаты для применения описанных выше лямбда-выражений.

Замечание

Внешние переменные могут использоваться и захватываться в лямбда-выражениях таким же образом, как и в анонимных методах.

Общая форма одиночного лямбда-выражения

(список_параметров) => выражение

Пример I:
count => count + 2 //значение параметра count увеличивается на 2.

Пример II:
n => n % 2 == 0 //выражение возвращает логическое значение true, если n четное, а иначе — false.

Пример III: (low, high, val) => val >= low && val <= high;
//выражение возвращает логическое значение true, если val внутри границ диапазона.

Упражнение 8.4

Использовать делегаты для применения описанных выше лямбда-выражений.

Упражнение 8.5

Продемонстрировать применение захваченной переменной в лямбда-выражении.

Общая форма блочного лямбда-выражения

(список_параметров) => {тело_выражения}

Общая форма блочного лямбда-выражения

(список_параметров) => {тело_выражения}

Упражнение 8.6

Продемонстрировать применение блочного лямбда-выражения для вычисления факториала.

Общая форма блочного лямбда-выражения

(список_параметров) => {тело_выражения}

Упражнение 8.6

Продемонстрировать применение блочного лямбда-выражения для вычисления факториала.

Упражнение 8.7

Переписать упражнение 8.1 используя блочные лямбда-выражения.

Пример:

```
using System;
delegate void MyEventHandler(); //Объявить тип делегата для события.
class MyEvent //Объявить класс, содержащий событие.
{
    public event MyEventHandler SomeEvent;
    public void OnSomeEvent() //Метод для запуска события.
    {
        if (SomeEvent != null) SomeEvent();
    }
}
class EventDemo
{
    static void Handler() //Обработчик события.
    {Console.WriteLine("Произошло событие");}
    static void Main()
    {
        MyEvent evt = new MyEvent();
        //Добавить метод Handler() в список событий.
        evt.SomeEvent += Handler;
        evt.OnSomeEvent(); //Запустить событие.
    }
}
```

Пример:

```
using System;
delegate void MyEventHandler(); //Объявить тип делегата для события.
class MyEvent //Объявить класс, содержащий событие.
{
    public event MyEventHandler SomeEvent;
    public void OnSomeEvent() //Метод для запуска события.
    {
```

Упражнение 8.8

Продемонстрировать групповую адресацию события. Использовать как методы экземпляра так и статические методы для обработки события.

```
{
    static void Handler() //Обработчик события.
    {Console.WriteLine("Произошло событие");}
    static void Main()
    {
        MyEvent evt = new MyEvent();
        //Добавить метод Handler() в список событий.
        evt.SomeEvent += Handler;
        evt.OnSomeEvent(); //Запустить событие.
    }
}
```

Расширенная форма оператора event

```
event делегат_события имя_события
{
    add
    {
        // Код добавления события в цепочку событий.
    }
    remove
    {
        // Код удаления события из цепочки событий.
    }
}
```

Расширенная форма оператора event

```
event делегат_события имя_события
{
    add
    {
        // Код добавления события в цепочку событий.
    }
    remove
    {
        // Код удаления события из цепочки событий.
    }
}
```

Замечание

Когда вызывается аксессор add или remove, он принимает в качестве параметра добавляемый или удаляемый обработчик. Как и в других разновидностях аксессоров, этот неявный параметр называется value.

Расширенная форма оператора event

```
event делегат_события имя_события
{
    add
    {
        // Код добавления события в цепочку событий.
    }
    remove
    {
        // Код удаления события из цепочки событий.
    }
}
```

Упражнение 8.9

Используя расширенную форму оператора event, реализовать обработку не более чем 3-х событий*.

**Подсказка: использовать массив событий.*

Упражнение 8.10

Применить анонимные методы вместе с событиями.

Упражнение 8.10

Применить анонимные методы вместе с событиями.

Упражнение 8.11

Применить лямбда-выражения вместе с событиями.

Общая форма обработчика события

```
void обработчик(object отправитель, EventArgs e)
{
    //...
}
```

Общая форма обработчика события

```
void обработчик(object отправитель, EventArgs e)
{
    //...
}
```

Замечание

Как правило, отправитель — это параметр, передаваемый вызывающим кодом с помощью ключевого слова this. А параметр e типа EventArgs содержит дополнительную информацию о событии и может быть проигнорирован, если он не нужен.

Пример:

```
using System;
using System.Windows.Forms;
using System.ComponentModel;
class MyForm : Form
{
    private Button button1;
    public MyForm()
    {
        button1 = new Button();
        button1.Text = "MyButton";
        button1.Click += button1_Click;
        Text = "Form1 ";
        StartPosition = FormStartPosition.CenterScreen;
        Controls.Add(button1);
    }
    private void button1_Click(object sender, EventArgs e)
    {
        Console.WriteLine("Объект: " + sender.ToString());
        Console.WriteLine("Тип события: " + e.GetType());
        Form.ActiveForm.Close();
    }
}
```

```
class MyFormDemo
{
    static void Main(string[] args)
    {
        Application.Run(new MyForm());
        Console.WriteLine("Форма закрыта ");
    }
}
```

```
class MyFormDemo
{
    static void Main(string[] args)
    {
        Application.Run(new MyForm());
        Console.WriteLine("Форма закрыта ");
    }
}
```

Замечание

Для запуска примера нужно добавить в проекте ссылки на сборки "System" и "System.Windows.Forms"(Project → References → Добавить ссылку).

```
class MyFormDemo
{
    static void Main(string[] args)
    {
        Application.Run(new MyForm());
        Console.WriteLine("Форма закрыта ");
    }
}
```

Упражнение 8.12

Написать свой пример формирования .NET-совместимого события. Использовать собственные классы с обработчиками события вида:

```
public void Handler(object source, EventArgs arg)
{
    //...
}
```

Встроенный обобщенный делегат EventHandler<TEventArgs>

В среде .NET Framework предоставляется встроенный обобщенный делегат EventHandler<TEventArgs>. Тип TEventArgs обозначает тип аргумента, передаваемого параметру EventArgs события.

Встроенный обобщенный делегат EventHandler<TEventArgs>

В среде .NET Framework предоставляется встроенный обобщенный делегат EventHandler<TEventArgs>. Тип TEventArgs обозначает тип аргумента, передаваемого параметру EventArgs события.

Пример:

```
//Объявить класс, производный от класса EventArgs.
class MyEventArgs : EventArgs{...}
class MyEvent//Объявить класс, содержащий событие.
{
     public event EventHandler<MyEventArgs> SomeEvent;
    //Метод запускающий событие SomeEvent.
     public void OnSomeEvent(){...}
}
```

Встроенный обобщенный делегат EventHandler<TEventArgs>

В среде .NET Framework предоставляется встроенный обобщенный делегат EventHandler<TEventArgs>. Тип TEventArgs обозначает тип аргумента, передаваемого параметру EventArgs события.

Встроенный необобщенный делегат EventHandler

В среде .NET Framework предоставляется встроенный необобщенный делегат EventHandler. Он может быть использован для объявления обработчиков событий, которым не требуется дополнительная информация о событиях. В этом случае в качестве параметра EventArgs события используется статический параметр EventArgs.Empty.

Встроенный обобщенный делегат EventHandler<TEventArgs>

В среде .NET Framework предоставляется встроенный обобщенный делегат EventHandler<TEventArgs>. Тип TEventArgs обозначает тип аргумента, передаваемого параметру EventArgs события.

Встроенный необобщенный делегат EventHandler

В среде .NET Framework предоставляется встроенный необобщенный делегат EventHandler. Он может быть использован для объявления обработчиков событий, которым не требуется дополнительная информация о событиях. В этом случае в качестве параметра EventArgs события используется статический параметр EventArgs.Empty.

Упражнение 8.13

Переделать упражнение 8.12 удалив свой делегат и добавив встроенный обобщенный или необобщенный делегат EventHandler.

Встроенный обобщенный делегат EventHandler<TEventArgs>

В среде .NET Framework предоставляется встроенный обобщенный делегат EventHandler<TEventArgs>. Тип TEventArgs обозначает тип аргумента, передаваемого параметру EventArgs события.

Встроенный необобщенный делегат EventHandler

В среде .NET Framework предоставляется встроенный необобщенный делегат EventHandler. Он может быть использован для объявления обработчиков событий, которым не требуется дополнительная информация о событиях. В этом случае в качестве параметра EventArgs события используется статический параметр EventArgs.Empty.

Упражнение 8.13

Переделать упражнение 8.12 удалив свой делегат и добавив встроенный обобщенный или необобщенный делегат EventHandler.

Упражнение 8.14

Написать программу для обработки событий, связанных с нажатием клавиш на клавиатуре.

Общая форма объявления пространства имен

```
namespace имя
{
  //члены
}
```

Общая форма объявления пространства имен

```
namespace имя
{
  //члены
}
```

Замечание

Доступ к членам пространства имен осуществляется следующим образом: имя_пространства_имен.имя_члена

Общая форма объявления пространства имен

```
namespace имя
{
  //члены
}
```

Пример:

```
namespace NStest
{
    class A {... }
    class B {... }
    class C {... }
}
class NSDemo
{
    static void Main()
    {
        NStest.A a1 = new NStest.A();
        NStest.B b1 =new NStest.B();
        NStest.C c1 = new NStest.C();
    }
}
```

Общая форма объявления пространства имен

```
namespace имя
{
  //члены
}
```

Упражнение 9.1

Добавить в одну из своих программы несколько пространств имен и реализовать обращения к их членам. В качестве членов разных пространств имен использовать классы с одинаковыми именами.

Директива using

С помощью директивы using можно сделать видимыми вновь создаваемые пространства имен. Существуют две формы этой директивы:
using имя;
using псевдоним = имя;

Директива using

С помощью директивы using можно сделать видимыми вновь создаваемые пространства имен. Существуют две формы этой директивы:
using имя;
using псевдоним = имя;

Пример:

```
using ClassAFromNStest = NStest.A;
namespace NStest
{
    class A {... }
}
class NSDemo
{
    ClassAFromNStest aNStest = new ClassAFromNStest();
    ...
}
```

Директива using

С помощью директивы using можно сделать видимыми вновь создаваемые пространства имен. Существуют две формы этой директивы:
using имя;
using псевдоним = имя;

Пример:

```
using ClassAFromNStest = NStest.A;
namespace NStest
{
    class A {... }
}
class NSDemo
{
    ClassAFromNStest aNStest = new ClassAFromNStest();
    ...
}
```

Упражнение 9.2

Для предыдущего упражнения продемонстрировать применение обеих форм директивы using.

Замечание

Под одним именем можно объявить несколько пространств имен. Это дает возможность распределить пространство имен по нескольким файлам или даже разделить его в пределах одного и того же файла исходного кода.

Замечание

Под одним именем можно объявить несколько пространств имен. Это дает возможность распределить пространство имен по нескольким файлам или даже разделить его в пределах одного и того же файла исходного кода.

Две формы задания вложенных пространств имен:

I:
```
namespace OuterNS
{
    namespace InnerNS
    {
        //...
    }
}
```

II:
```
namespace OuterNS.InnerNS
{
    //...
}
```

Замечание

Под одним именем можно объявить несколько пространств имен. Это дает возможность распределить пространство имен по нескольким файлам или даже разделить его в пределах одного и того же файла исходного кода.

Две формы задания вложенных пространств имен:

I:
```
namespace OuterNS
{
    namespace InnerNS
    {
        //...
    }
}
```

II:
```
namespace OuterNS.InnerNS
{
    //...
}
```

Общая форма оператора ::

псевдоним_пространства_имен::идентификатор

Общая форма оператора ::

псевдоним_пространства_имен::идентификатор

Пример:

```
using NSt = NStest;
...
NSt::A cd1 = new NSt::A();
```

Общая форма оператора ::

псевдоним_пространства_имен::идентификатор

Пример:

```
using NSt = NStest;
...
NSt::A cd1 = new NSt::A();
```

Замечание

Описатель :: можно также использовать вместе с предопределенным идентификатором global для ссылки на глобальное пространство имен.

Общая форма оператора ::

псевдоним_пространства_имен::идентификатор

Пример:

```
using NSt = NStest;
...
NSt::A cd1 = new NSt::A();
```

Замечание

Описатель :: можно также использовать вместе с предопределенным идентификатором global для ссылки на глобальное пространство имен.

Упражнение 9.3

Использовать описатель :: для идентификации классов с одинаковыми именами из разных пространств имен

Таблица 1: Директивы препроцессора, определенные в C#.

`#define`	`#elif`	`#else`	`#endif`
`#endregion`	`#error`	`#if`	`#line`
`#pragma`	`#region`	`#undef`	`#warning`

Таблица 1: Директивы препроцессора, определенные в C#.

`#define`	`#elif`	`#else`	`#endif`
`#endregion`	`#error`	`#if`	`#line`
`#pragma`	`#region`	`#undef`	`#warning`

Замечание

Термин директива препроцессора появился в связи с тем, что подобные инструкции по традиции обрабатывались на отдельной стадии компиляции, называемой препроцессором. Обрабатывать директивы на отдельной стадии препроцессора в современных компиляторах уже не нужно, но само ее название закрепилось.

Общая форма директивы #define.

#define идентификатор

Общая форма директивы #define.

```
#define идентификатор
```

Общая форма директив #if и #endif

```
#if идентификаторное_выражение
  последовательность операторов
#endif
```

Общая форма директивы #define.

#define идентификатор

Общая форма директив #if и #endif

#if идентификаторное_выражение
  последовательность операторов
#endif

Замечание

Идентификаторное выражение может быть простым, как наименование идентификатора. В то же время в нем разрешается применение следующих операторов: $!$, $==$, $!=$, $\&\&$ *и* $||$, *а также круглых скобок.*

Общая форма директивы #define.

```
#define идентификатор
```

Общая форма директив #if и #endif

```
#if идентификаторное_выражение
  последовательность операторов
#endif
```

Замечание

Идентификаторное выражение может быть простым, как наименование идентификатора. В то же время в нем разрешается применение следующих операторов: !, ==, !=, && *и* ||, *а также круглых скобок.*

Упражнение 9.4

Продемонстрировать применение директив #if, #endif и #define. Использовать идентификаторное выражение.

Общая форма директивы #define.

#define идентификатор

Общая форма директив #if и #endif

#if идентификаторное_выражение
  последовательность операторов
#endif

Замечание

Идентификаторное выражение может быть простым, как наименование идентификатора. В то же время в нем разрешается применение следующих операторов: !, ==, != , && *и* ||, *а также круглых скобок.*

Упражнение 9.4

Продемонстрировать применение директив #if, #endif и #define. Использовать идентификаторное выражение.

Упражнение 9.5

Продемонстрировать применение директив #else и #elif.

Общая форма директивы #undef.

#undef идентификатор

Общая форма директивы #undef.

#undef идентификатор

Общая форма директивы #error.

#error сообщение_об_ошибке

Общая форма директивы #undef.

#undef идентификатор

Общая форма директивы #error.

#error сообщение_об_ошибке

Общая форма директивы #warning.

#warning предупреждающее_сообщение

Общая форма директивы #undef.

#undef идентификатор

Общая форма директивы #error.

#error сообщение_об_ошибке

Общая форма директивы #warning.

#warning предупреждающее_сообщение

Общая форма директивы #line.

#line номер "имя_файла"

Общая форма директивы #undef.

#undef идентификатор

Общая форма директивы #error.

#error сообщение_об_ошибке

Общая форма директивы #warning.

#warning предупреждающее_сообщение

Общая форма директивы #line.

#line номер "имя_файла"

[1] #line default

[2] #line hidden

Общая форма директивы #undef.

```
#undef идентификатор
```

Общая форма директивы #error.

```
#error сообщение_об_ошибке
```

Общая форма директивы #warning.

```
#warning предупреждающее_сообщение
```

Общая форма директивы #line.

```
#line номер "имя_файла"
```

Общая форма директив #region и #endregion

```
#region текст
// последовательность кода
#endregion текст
```

Общая форма директивы #pragma

```
#pragma опция
```

Общая форма директивы #line.

```
#line номер "имя_файла"
```

Общая форма директив #region и #endregion

```
#region текст
// последовательность кода
#endregion текст
```

Общая форма директивы #pragma

```
#pragma опция
```

[1] #pragma warning disable номер_предупреждения

[2] #pragma warning restore номер_предупреждения

[3] #pragma checksum

Общая форма директивы #line.

```
#line номер "имя_файла"
```

Общая форма директив #region и #endregion

```
#region текст
// последовательность кода
#endregion текст
```

Общая форма оператора is

выражение is тип

где *выражение* обозначает отдельное выражение, описывающее объект, тип которого проверяется. Если выражение имеет совместимый или такой же тип, как и проверяемый тип, то результат этой операции получается истинным, в противном случае – ложным.

Общая форма оператора is

выражение is тип
где *выражение* обозначает отдельное выражение, описывающее объект, тип которого проверяется. Если выражение имеет совместимый или такой же тип, как и проверяемый тип, то результат этой операции получается истинным, в противном случае – ложным.

Общая форма оператора as

выражение as тип
где *выражение* обозначает отдельное выражение, преобразуемое в указанный тип. Если исход такого преобразования оказывается удачным, то возвращается ссылка на тип, а иначе — пустая ссылка.

Общая форма оператора is

выражение is тип
где *выражение* обозначает отдельное выражение, описывающее объект, тип которого проверяется. Если выражение имеет совместимый или такой же тип, как и проверяемый тип, то результат этой операции получается истинным, в противном случае – ложным.

Общая форма оператора as

выражение as тип
где *выражение* обозначает отдельное выражение, преобразуемое в указанный тип. Если исход такого преобразования оказывается удачным, то возвращается ссылка на тип, а иначе — пустая ссылка.

Упражнение 9.6

Продемонстрировать применение оператора is.

Общая форма оператора is

выражение is тип
где *выражение* обозначает отдельное выражение, описывающее объект, тип которого проверяется. Если выражение имеет совместимый или такой же тип, как и проверяемый тип, то результат этой операции получается истинным, в противном случае – ложным.

Общая форма оператора as

выражение as тип
где *выражение* обозначает отдельное выражение, преобразуемое в указанный тип. Если исход такого преобразования оказывается удачным, то возвращается ссылка на тип, а иначе — пустая ссылка.

Упражнение 9.6

Продемонстрировать применение оператора is.

Упражнение 9.7

Продемонстрировать применение оператора as.

Рефлексия

Рефлексия — это средство, позволяющее получать сведения о типе данных. Для применения рефлексии в код программы обычно вводится строка:

```
using System.Reflection;
```

Рефлексия

Рефлексия — это средство, позволяющее получать сведения о типе данных. Для применения рефлексии в код программы обычно вводится строка:
using System.Reflection;

Оператор typeof

Данный оператор извлекает объект класса System.Type для заданного типа. Информация, описывающая тип, инкапсулируется в возвращаемом объекте класса Type.

Рефлексия

Рефлексия — это средство, позволяющее получать сведения о типе данных. Для применения рефлексии в код программы обычно вводится строка:
using System.Reflection;

Оператор typeof

Данный оператор извлекает объект класса System.Type для заданного типа. Информация, описывающая тип, инкапсулируется в возвращаемом объекте класса Type.

Общая форма оператора typeof

typeof(тип);
где тип обозначает получаемый тип.

Рефлексия

Рефлексия — это средство, позволяющее получать сведения о типе данных. Для применения рефлексии в код программы обычно вводится строка:
using System.Reflection;

Оператор typeof

Данный оператор извлекает объект класса System.Type для заданного типа. Информация, описывающая тип, инкапсулируется в возвращаемом объекте класса Type.

Общая форма оператора typeof

typeof(тип);
где тип обозначает получаемый тип.

Пример: Type t = typeof(MyClass);

Рефлексия

Рефлексия — это средство, позволяющее получать сведения о типе данных. Для применения рефлексии в код программы обычно вводится строка:
using System.Reflection;

Оператор typeof

Данный оператор извлекает объект класса System.Type для заданного типа. Информация, описывающая тип, инкапсулируется в возвращаемом объекте класса Type.

Общая форма оператора typeof

typeof(тип);
где тип обозначает получаемый тип.

Пример: Type t = typeof(MyClass);

Упражнение 9.8

Продемонстрировать применение оператора typeof используя три свойства класса Type: FullName, IsClass и IsAbstract.

Таблица 2: Свойства абстрактного класса System.Reflection.MemberInfo

Свойство	Описание
Type DeclaringType	Тип класса или интерфейса, в котором объявляется отражаемый член
MemberTypes MemberType	Тип члена. Это значение обозначает, является ли член полем, методом, свойством, событием или конструктором
int MetadataToken	Значение, связанное к конкретными метаданными
Module Module	Объект типа Module, представляющий модуль (исполняемый файл), в котором находится отражаемый
string Name	Имя типа
Type ReflectedType	Тип отражаемого объекта

Таблица 2: Свойства абстрактного класса System.Reflection.MemberInfo

Свойство	Описание
Type DeclaringType	Тип класса или интерфейса, в котором объявляется отражаемый член
MemberTypes MemberType	Тип члена. Это значение обозначает, является ли член полем, методом, свойством, событием или конструктором
int MetadataToken	Значение, связанное к конкретными метаданными
Module Module	Объект типа Module, представляющий модуль (исполняемый файл), в котором находится отражаемый
string Name	Имя типа
Type ReflectedType	Тип отражаемого объекта

Таблица 3: Методы абстрактного класса System.Reflection.MemberInfo

Метод	Назначение
GetCustomAttributes()	Получает список специальных атрибутов, имеющих отношение к вызывающему объекту
IsDefined()	Устанавливает, определен ли атрибут для вызывающего метода

Таблица 4: Методы класса Type

Метод	Назначение
ConstructorInfo[] GetConstructors()	Получает список конструкторов для заданного типа
EventInfo[] GetEvents()	Получает список событий для заданного типа
FieldInfo[] GetFields()	Получает список полей для заданного типа
Type[] GetGenericArguments()	Получает список аргументов типа, связанных с закрыто сконструированным обобщенным типом, или же список параметров типа, если заданный тип определен как обобщенный. Для открыто сконструированного типа этот список может содержать как аргументы, так и параметры типа.
MemberInfo[] GetMembers()	Получает список членов для заданного типа
MethodInfo[] GetMethods()	Получает список методов для заданного типа
PropertyInfo[] GetProperties()	Получает список свойств для заданного типа

Таблица 5: Свойства класса Type

Свойство	Назначение
Assembly Assembly	Получает сборку для заданного типа
TypeAttributes Attributes	Получает атрибуты для заданного типа
Type BaseType	Получает непосредственный базовый тип для заданного типа
string FullName	Получает полное имя заданного типа
bool IsAbstract	Истинно, если заданный тип является абстрактным
bool IsArray	Истинно, если заданный тип является массивом
bool IsClass	Истинно, если заданный тип является классом
bool IsEnum	Истинно, если заданный тип является перечислением
bool IsGenericParameter	Истинно, если заданный тип является параметром обобщенного типа.
bool IsGenericType	Истинно, если заданный тип является обобщенным.
string Namespace	Получает пространство имен для заданного типа

Метод GetMethods()

MethodInfo[] GetMethods()
Этот метод возвращает массив объектов класса MethodInfo, которые описывают методы, поддерживаемые вызывающим типом.

Метод GetMethods()

MethodInfo[] GetMethods()
Этот метод возвращает массив объектов класса MethodInfo, которые описывают методы, поддерживаемые вызывающим типом.

Замечание

Для получения имени метода служит свойство Name. Возвращаемый тип метода находится в доступном только для чтения свойстве ReturnType, которое является объектом класса Type.

Метод GetMethods()

MethodInfo[] GetMethods()
Этот метод возвращает массив объектов класса MethodInfo, которые описывают методы, поддерживаемые вызывающим типом.

Замечание

Для получения имени метода служит свойство Name. Возвращаемый тип метода находится в доступном только для чтения свойстве ReturnType, которое является объектом класса Type.

Метод GetParameters()

ParameterInfo[] GetParameters()
Метод GetParameters() возвращает список параметров, связанных с анализируемым методом.

Метод GetMethods()

MethodInfo[] GetMethods()
Этот метод возвращает массив объектов класса MethodInfo, которые описывают методы, поддерживаемые вызывающим типом.

Замечание

Для получения имени метода служит свойство Name. Возвращаемый тип метода находится в доступном только для чтения свойстве ReturnType, которое является объектом класса Type.

Метод GetParameters()

ParameterInfo[] GetParameters()
Метод GetParameters() возвращает список параметров, связанных с анализируемым методом.

Замечание

Особое значение имеют два свойства класса ParameterInfo : Name — представляет собой строку, содержащую имя параметра, а ParameterType — описывает тип параметра, который инкапсулирован в объекте класса Type.

Метод GetMethods()

MethodInfo[] GetMethods()
Этот метод возвращает массив объектов класса MethodInfo, которые описывают методы, поддерживаемые вызывающим типом.

Замечание

Для получения имени метода служит свойство Name. Возвращаемый тип метода находится в доступном только для чтения свойстве ReturnType, которое является объектом класса Type.

Метод GetParameters()

ParameterInfo[] GetParameters()
Метод GetParameters() возвращает список параметров, связанных с анализируемым методом.

Упражнение 9.9

Провести анализ методов своего класса с помощью рефлексии (используя методы GetMethods() и GetParameters(), и свойства Name, ReturnType и ParameterType)

Метод Invoke()

```
object Invoke(object obj, object[] parameters)
```

Метод Invoke()

```
object Invoke(object obj, object[] parameters)
```

Упражнение 9.10

Вызвать методы своего класса с помощью рефлексии (используя метод Invoke()).

Пример:

```
class MyClass
{
    public int MyMethod(){...}
...
}
class InvokeMethDemo
{
    static void Main()
    {
        Type t = typeof(MyClass);
        MyClass reflectOb = new MyClass(10, 20);
        MethodInfo[] mi = t.GetMethods();
        foreach(MethodInfo m in mi) //Вызвать метод.
        {
            ParameterInfo[] pi = m.GetParameters(); //Получить параметры.
            if(m.Name.CompareTo("MyMethod")==0 )
            {
                m.Invoke(reflectOb, args);
            }
        }
    }
}
```

Метод GetConstructors()

Метод GetConstructors() возвращает массив объектов класса ConstructorInfo, описывающих конструкторы.

Метод GetConstructors()

Метод GetConstructors() возвращает массив объектов класса ConstructorInfo, описывающих конструкторы.

Форма метода Invoke() для создания объекта

```
object Invoke(object[] parameters)
```

Метод GetConstructors()

Метод GetConstructors() возвращает массив объектов класса ConstructorInfo, описывающих конструкторы.

Форма метода Invoke() для создания объекта

```
object Invoke(object[] parameters)
```

Пример:
```
Type t = typeof(MyClass);
ConstructorInfo[] ci = t.GetConstructors();
object reflectOb = ci[2].Invoke(1,1);
```

Метод GetConstructors()

Метод GetConstructors() возвращает массив объектов класса ConstructorInfo, описывающих конструкторы.

Форма метода Invoke() для создания объекта

```
object Invoke(object[] parameters)
```

Пример:
```
Type t = typeof(MyClass);
ConstructorInfo[] ci = t.GetConstructors();
object reflectOb = ci[2].Invoke(1,1);
```

Упражнение 9.11

Создать объект с помощью рефлексии.

Создание объекта класса Assembly

static Assembly LoadFrom(string файл_сборки)
где *файл_сборки* обозначает конкретное имя файла сборки.

Создание объекта класса Assembly

static Assembly LoadFrom(string файл_сборки)
где *файл_сборки* обозначает конкретное имя файла сборки.

Получение типов данных, определенных в объекте класса Assembly

Type[] Имя имя_объекта.GetTypes()

Создание объекта класса Assembly

static Assembly LoadFrom(string файл_сборки)
где *файл_сборки* обозначает конкретное имя файла сборки.

Получение типов данных, определенных в объекте класса Assembly

Type[] Имя имя_объекта.GetTypes()

Пример:

```
// Загрузить сборку MyClasses.exe.
Assembly asm = Assembly.LoadFrom("MyClasses.exe");
// Обнаружить типы, содержащиеся в сборке MyClasses.exe.
Type[] alltypes = asm.GetTypes();
```

Создание объекта класса Assembly

static Assembly LoadFrom(string файл_сборки)
где *файл_сборки* обозначает конкретное имя файла сборки.

Получение типов данных, определенных в объекте класса Assembly

Type[] Имя имя_объекта.GetTypes()

Пример:

```
// Загрузить сборку MyClasses.exe.
Assembly asm = Assembly.LoadFrom("MyClasses.exe");
// Обнаружить типы, содержащиеся в сборке MyClasses.exe.
Type[] alltypes = asm.GetTypes();
```

Упражнение 10.1

Обнаружить сборку (сделать для своего класса), определить типы и создать объект с помощью рефлексии.

Создание объекта класса Assembly

static Assembly LoadFrom(string файл_сборки)
где *файл_сборки* обозначает конкретное имя файла сборки.

Получение типов данных, определенных в объекте класса Assembly

Type[] Имя имя_объекта.GetTypes()

Пример:

```
// Загрузить сборку MyClasses.exe.
Assembly asm = Assembly.LoadFrom("MyClasses.exe");
// Обнаружить типы, содержащиеся в сборке MyClasses.exe.
Type[] alltypes = asm.GetTypes();
```

Упражнение 10.1

Обнаружить сборку (сделать для своего класса), определить типы и создать объект с помощью рефлексии.

Упражнение 10.2

Выполнить предыдущее упражнение для dll-библиотеки созданной одногруппником. Вызвать несколько методов определенных в классах библиотеки.

Атрибут

С помощью атрибута(т.е. специального класса) определяются дополнительные сведения, связанные с классом, структурой, методом и т.д.

Атрибут

С помощью атрибута(т.е. специального класса) определяются дополнительные сведения, связанные с классом, структурой, методом и т.д.

Пример: [AttributeUsage(AttributeTargets.All)]

```
public class RemarkAttribute : Attribute
{
    string pri_remark; // базовое поле свойства Remark
    public RemarkAttribute(string comment)
    {
        pri_remark = comment;
    }
    public string Remark
    {
        get
        {
            return pri_remark;
        }
    }
}
```

Замечание

Атрибут указывается перед тем элементом, к которому он присоединяется, и для этого его конструктор заключается в квадратные скобки.

Замечание

Атрибут указывается перед тем элементом, к которому он присоединяется, и для этого его конструктор заключается в квадратные скобки.

Пример:

```
[RemarkAttribute("В этом классе используется атрибут.")]
class UseAttrib
{
// ...
}
```

Замечание

Атрибут указывается перед тем элементом, к которому он присоединяется, и для этого его конструктор заключается в квадратные скобки.

Пример:

```
[Remark("В этом классе используется атрибут.")]
class UseAttrib
{
// ...
}
```

Замечание

Атрибут указывается перед тем элементом, к которому он присоединяется, и для этого его конструктор заключается в квадратные скобки.

Пример:

```
[Remark("В этом классе используется атрибут.")]
class UseAttrib
{
// ...
}
```

Два метода получения атрибутов объекта

[1] object[] GetCustomAttributes(bool наследование)
Если наследование имеет логическое значение true, то в список включаются атрибуты всех базовых классов, наследуемых по иерархической цепочке. В противном случае атрибуты извлекаются только из тех классов, которые определяются указанным типом.

[2] static Attribute.GetCustomAttribute(MemberInfo элемент, Type тип_а)
где элемент обозначает объект класса MemberInfo, описывающий тот элемент, для которого создаются атрибуты, тогда как тип_а — требуемый атрибут.

Замечание

Первый метод определяется в классе MemberInfo и наследуется классом Type. Второй метод определяется в классе Attribute.

Два метода получения атрибутов объекта

[1] object[] GetCustomAttributes(bool наследование)
Если наследование имеет логическое значение true, то в список включаются атрибуты всех базовых классов, наследуемых по иерархической цепочке. В противном случае атрибуты извлекаются только из тех классов, которые определяются указанным типом.

[2] static Attribute.GetCustomAttribute(MemberInfo элемент, Type тип_а)
где элемент обозначает объект класса MemberInfo, описывающий тот элемент, для которого создаются атрибуты, тогда как тип_а — требуемый атрибут.

Замечание

Первый метод определяется в классе MemberInfo и наследуется классом Type. Второй метод определяется в классе Attribute.

Пример:

```
//Получить экземпляр объекта класса MemberInfo, связанного
//с классом, содержащим атрибут RemarkAttribute.
Type t = typeof(UseAttrib);
// Извлечь атрибут RemarkAttribute.
Type RemAt = typeof(RemarkAttribute);
RemarkAttribute ra = (RemarkAttribute)Attribute.GetCustomAttribute(t, RemAt);
```

Замечание

Первый метод определяется в классе MemberInfo и наследуется классом Type. Второй метод определяется в классе Attribute.

Пример:

```
//Получить экземпляр объекта класса MemberInfo, связанного
//с классом, содержащим атрибут RemarkAttribute.
Type t = typeof(UseAttrib);
// Извлечь атрибут RemarkAttribute.
Type RemAt = typeof(RemarkAttribute);
RemarkAttribute ra = (RemarkAttribute)Attribute.GetCustomAttribute(t, RemAt);
```

Упражнение 10.3

Реализовать программу демонстрирующую применение атрибута.

Общая форма

[attrib(список_позиционных_параметров,
именованный_параметр_1 = значение,
именованный_параметр_2 = значение, ...)]

Общая форма

[attrib(список_позиционных_параметров,
именованный_параметр_1 = значение,
именованный_параметр_2 = значение, ...)]

Замечание

Первыми указываются позиционные параметры, если они существуют. Далее следуют именованные параметры с присваиваемыми значениями. Порядок следования именованных параметров особого значения не имеет. Именованным параметрам не обязательно присваивать значение, и в этом случае используется значение, устанавливаемое по умолчанию.

Пример:

```
[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute
{
    string pri_remark; // базовое поле свойства Remark
    // Это поле можно использовать в качестве именованного параметра.
    public string Supplement;
    public RemarkAttribute(string comment)
    {
        pri_remark = comment;
        Supplement = "Отсутствует";
    }
    public string Remark
    {
        get
        {
            return pri_remark;
        }
    }
}
```

Пример:

```
[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute
{
    string pri_remark; // базовое поле свойства Remark
    // Это поле можно использовать в качестве именованного параметра.
    public string Supplement;
    public RemarkAttribute(string comment)
    {
        pri_remark = comment;
        Supplement = "Отсутствует";
    }
[RemarkAttribute("В этом классе используется атрибут.",
Supplement = "Это дополнительная информация.")]
class UseAttrib
{
    //...
}
```

Пример:

```
[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute
{
    string pri_remark; // базовое поле свойства Remark
    // Это поле можно использовать в качестве именованного параметра.
    public string Supplement;
    public RemarkAttribute(string comment)
    {
        pri_remark = comment;
        Supplement = "Отсутствует";
    }
[RemarkAttribute("В этом классе используется атрибут.",
Supplement = "Это дополнительная информация.")]
class UseAttrib
{
    //...
}
```

Упражнение 10.4

Реализовать программу демонстрирующую применение атрибута с именованными параметрами (использовать поля и свойства).

Пример:

```
[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute
{
    string pri_remark; // базовое поле свойства Remark
    // Это поле можно использовать в качестве именованного параметра.
    public string Supplement;
    public RemarkAttribute(string comment)
    {
        pri_remark = comment;
        Supplement = "Отсутствует";
    }
```

Замечание

Тип параметра атрибута (как позиционного, так и именованного) должен быть одним из встроенных простых типов, object, Type, перечислением или одномерным массивом одного из этих типов.

Атрибут AttributeUsage

Атрибут AttributeUsage определяет типы элементов, к которым может быть применен объявляемый атрибут. У него имеется следующий конструктор:
AttributeUsage(AttributeTargets validOn)
где validOn обозначает один или несколько элементов, к которым может быть применен объявляемый атрибут, тогда как AttributeTargets — перечисление, в котором определяются приведенные в Таблице 1 значения.

Атрибут AttributeUsage

Атрибут AttributeUsage определяет типы элементов, к которым может быть применен объявляемый атрибут. У него имеется следующий конструктор:
AttributeUsage(AttributeTargets validOn)
где validOn обозначает один или несколько элементов, к которым может быть применен объявляемый атрибут, тогда как AttributeTargets — перечисление, в котором определяются приведенные в Таблице 1 значения.

Таблица 1: Значения перечисления AttributeTargets

All	Assembly	Class	Constructor
Delegate	Enum	Event	Field
GenericParameter	Interface	Method	Module
Parameter	Property	ReturnValue	Struct

Атрибут Conditional

Атрибут Conditional позволяет создавать условные методы, которые вызываются только в том случае, если с помощью директивы #define определен конкретный идентификатор, а иначе метод пропускается.

Атрибут Conditional

Атрибут Conditional позволяет создавать условные методы, которые вызываются только в том случае, если с помощью директивы #define определен конкретный идентификатор, а иначе метод пропускается.

Замечание

Для применения атрибута Conditional в исходный код программы следует включить пространство имен System.Diagnostics.

Атрибут Conditional

Атрибут Conditional позволяет создавать условные методы, которые вызываются только в том случае, если с помощью директивы #define определен конкретный идентификатор, а иначе метод пропускается.

Замечание

Для применения атрибута Conditional в исходный код программы следует включить пространство имен System.Diagnostics.

Пример:

```
#define тест
using System;
using System.Diagnostics;
class Test
{
    [Conditional("тест")]
    void Method {...}
}
```

Атрибут Conditional

Атрибут Conditional позволяет создавать условные методы, которые вызываются только в том случае, если с помощью директивы #define определен конкретный идентификатор, а иначе метод пропускается.

Замечание

Для применения атрибута Conditional в исходный код программы следует включить пространство имен System.Diagnostics.

Пример:

```
#define тест
using System;
using System.Diagnostics;
class Test
{
    [Conditional("тест")]
    void Method {...}
}
```

Упражнение 10.5

Продемонстрировать применение встроенного атрибута Conditional.

Атрибут Obsolete

Атрибут Obsolete позволяет пометить элемент программы как устаревший. Вот общая форма этого атрибута:

```
[Obsolete("сообщение")]
```

где сообщение выводится при компилировании элемента программы, помеченного как устаревший.

Атрибут Obsolete

Атрибут Obsolete позволяет пометить элемент программы как устаревший. Вот общая форма этого атрибута:
[Obsolete("сообщение")]
где сообщение выводится при компилировании элемента программы, помеченного как устаревший.

Вторая форма атрибута Obsolete

[Obsolete("сообщение", ошибка)]
где ошибка обозначает логическое значение. Если это значение истинно (true), то при использовании устаревшего элемента формируется сообщение об ошибке компиляции вместо предупреждения.

Атрибут Obsolete

Атрибут Obsolete позволяет пометить элемент программы как устаревший. Вот общая форма этого атрибута:
[Obsolete("сообщение")]
где сообщение выводится при компилировании элемента программы, помеченного как устаревший.

Вторая форма атрибута Obsolete

[Obsolete("сообщение", ошибка)]
где ошибка обозначает логическое значение. Если это значение истинно (true), то при использовании устаревшего элемента формируется сообщение об ошибке компиляции вместо предупреждения.

Упражнение 10.6

Продемонстрировать применение атрибута Obsolete.

Обобщение

Обобщение это параметризированный тип. Обобщения позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых обрабатываемые данные указываются в виде параметра.

Пример:

```
using System;
//В приведенном ниже классе Gen параметр типа T заменяется
//реальным типом данных при создании объекта типа Gen.
class Gen<T>
{
    T ob; //объявить переменную типа T
    public Gen(T o) //У конструктора имеется параметр типа T.
    {
        ob = o;
    }
    //Возвратить переменную экземпляра ob, которая относится к типу T.
    public T GetOb()
    {
        return ob;
    }
    public void ShowType()//Показать тип T.
    {
        Console.WriteLine("К типу T относится " + typeof(T));
    }
}
```

Упражнение 10.7

Продемонстрировать применение обобщенного класса.

Упражнение 10.7

Продемонстрировать применение обобщенного класса.

Упражнение 10.8

Написать "необобщенный" аналог класса Gen<T>* и переделать под него пример из предыдущего упражнения.
**Подсказка.* *Использовать переменную типа object.*

Упражнение 10.7

Продемонстрировать применение обобщенного класса.

Упражнение 10.8

Написать "необобщенный" аналог класса Gen<T>* и переделать под него пример из предыдущего упражнения.
**Подсказка.* *Использовать переменную типа object.*

Замечание

Обобщения позволяют создавать типизированный код, в котором ошибки несоответствия типов выявляются во время компиляции.

Обобщенный класс с несколькими параметрами

```
class имя_класса<список_параметров_типа> {//...}
```

Обобщенный класс с несколькими параметрами

```
class имя_класса<список_параметров_типа> {//...}
```

Общая форма объявления ссылки на обобщенный класс

```
имя_класса<список_аргументов_типа> имя_переменной =
new имя_класса<список_параметров_типа>(список_аргументов);
```

Обобщенный класс с несколькими параметрами

```
class имя_класса<список_параметров_типа> {//...}
```

Общая форма объявления ссылки на обобщенный класс

```
имя_класса<список_аргументов_типа> имя_переменной =
new имя_класса<список_параметров_типа>(список_аргументов);
```

Ограниченные типы

```
имя_класса<список_параметров_типа>
where параметр1_типа : ограничения
where параметр2_типа : ограничения
...
where параметрN_типа : ограничения { где ограничения указываются
списком через запятую.
```

В C# предусмотрен ряд ограничений на типы данных.

- Ограничение на базовый класс, требующее наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса. Разновидностью этого ограничения является неприкрытое ограничение типа, при котором на базовый класс указывает параметр типа, а не конкретный тип. Благодаря этому устанавливается взаимосвязь между двумя параметрами типа.
- Ограничение на интерфейс, требующее реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.
- Ограничение на конструктор, требующее предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора new().
- Ограничение ссылочного типа, требующее указывать аргумент ссылочного типа с помощью оператора class.
- Ограничение типа значения, требующее указывать аргумент типа значения с помощью оператора struct.

В C# предусмотрен ряд ограничений на типы данных.

- Ограничение на базовый класс, требующее наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса. Разновидностью этого ограничения является неприкрытое ограничение типа, при котором на базовый класс указывает параметр типа, а не конкретный тип. Благодаря этому устанавливается взаимосвязь между двумя параметрами типа.
- Ограничение на интерфейс, требующее реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.
- Ограничение на конструктор, требующее предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора new().
- Ограничение ссылочного типа, требующее указывать аргумент ссылочного типа с помощью оператора class.
- Ограничение типа значения, требующее указывать аргумент типа значения с помощью оператора struct.

В C# предусмотрен ряд ограничений на типы данных.

- Ограничение на базовый класс, требующее наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса. Разновидностью этого ограничения является неприкрытое ограничение типа, при котором на базовый класс указывает параметр типа, а не конкретный тип. Благодаря этому устанавливается взаимосвязь между двумя параметрами типа.
- Ограничение на интерфейс, требующее реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.
- Ограничение на конструктор, требующее предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора new().
- Ограничение ссылочного типа, требующее указывать аргумент ссылочного типа с помощью оператора class.
- Ограничение типа значения, требующее указывать аргумент типа значения с помощью оператора struct.

В C# предусмотрен ряд ограничений на типы данных.

- Ограничение на базовый класс, требующее наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса. Разновидностью этого ограничения является неприкрытое ограничение типа, при котором на базовый класс указывает параметр типа, а не конкретный тип. Благодаря этому устанавливается взаимосвязь между двумя параметрами типа.
- Ограничение на интерфейс, требующее реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.
- Ограничение на конструктор, требующее предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора new().
- Ограничение ссылочного типа, требующее указывать аргумент ссылочного типа с помощью оператора class.
- Ограничение типа значения, требующее указывать аргумент типа значения с помощью оператора struct.

В C# предусмотрен ряд ограничений на типы данных.

- Ограничение на базовый класс, требующее наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса. Разновидностью этого ограничения является неприкрытое ограничение типа, при котором на базовый класс указывает параметр типа, а не конкретный тип. Благодаря этому устанавливается взаимосвязь между двумя параметрами типа.
- Ограничение на интерфейс, требующее реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.
- Ограничение на конструктор, требующее предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора new().
- Ограничение ссылочного типа, требующее указывать аргумент ссылочного типа с помощью оператора class.
- Ограничение типа значения, требующее указывать аргумент типа значения с помощью оператора struct.

В C# предусмотрен ряд ограничений на типы данных.

- Ограничение на базовый класс, требующее наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса. Разновидностью этого ограничения является неприкрытое ограничение типа, при котором на базовый класс указывает параметр типа, а не конкретный тип. Благодаря этому устанавливается взаимосвязь между двумя параметрами типа.
- Ограничение на интерфейс, требующее реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.
- Ограничение на конструктор, требующее предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора new().
- Ограничение ссылочного типа, требующее указывать аргумент ссылочного типа с помощью оператора class.
- Ограничение типа значения, требующее указывать аргумент типа значения с помощью оператора struct.

Общая форма наложения ограничения на базовый класс

where T : имя_базового_класса
где T обозначает имя параметра типа, а имя_базового_класса — конкретное имя ограничиваемого базового класса.

Общая форма наложения ограничения на базовый класс

where T : имя_базового_класса
где T обозначает имя параметра типа, а имя_базового_класса — конкретное имя ограничиваемого базового класса.

Упражнение 10.9

На примере своих классов (A,B:A и C) продемонстрировать механизм наложения ограничения на базовый класс.

Общая форма наложения ограничения на базовый класс

where T : имя_базового_класса
где T обозначает имя параметра типа, а имя_базового_класса — конкретное имя ограничиваемого базового класса.

Упражнение 10.9

На примере своих классов (A,B:A и C) продемонстрировать механизм наложения ограничения на базовый класс.

Упражнение 10.10

Реализовать механизм управления списками телефонных номеров, чтобы можно было пользоваться разными категориями таких списков, в частности отдельными списками для друзей, поставщиков, клиентов.

Общая форма наложения ограничения на интерфейс

where T : имя_интерфейса
где T — это имя параметра типа, а имя_интерфейса — конкретное имя ограничиваемого интерфейса.

Общая форма наложения ограничения на интерфейс

where T : имя_интерфейса
где T — это имя параметра типа, а имя_интерфейса — конкретное имя ограничиваемого интерфейса.

Упражнение 10.11

Базовый класс для телефонных номеров (PhoneNumber) из предыдущего упражнения представить в форме интерфейса. Выполнить предыдущее упражнение используя данный интерфейс.

Ограничение new()

Ограничение new() на конструктор позволяет получать экземпляр объекта обобщенного типа.

Ограничение new()

Ограничение new() на конструктор позволяет получать экземпляр объекта обобщенного типа.

Пример:

```
class Test<T> where T : new()
{
    T obj;
    public Test()
    {
      //Этот код работоспособен благодаря наложению ограничения new().
      obj = new T(); //создать объект типа T
    }
    //...
}
```

Ограничение new()

Ограничение new() на конструктор позволяет получать экземпляр объекта обобщенного типа.

Пример:

```
class Test<T> where T : new()
{
    T obj;
    public Test()
    {
      //Этот код работоспособен благодаря наложению ограничения new().
      obj = new T(); //создать объект типа T
    }
    //...
}
```

Замечание

Ограничение new() можно использовать вместе с другими ограничениями, но последним по порядку. Ограничение new() позволяет конструировать объект, используя только конструктор без параметров. Ограничение new() нельзя использовать одновременно с ограничением типа значения.

Общая форма ограничения ссылочного типа

where T : class

В этой форме с оператором where ключевое слово class указывает на то, что аргумент T должен быть ссылочного типа.

Общая форма ограничения ссылочного типа

where T : class

В этой форме с оператором where ключевое слово class указывает на то, что аргумент T должен быть ссылочного типа.

Общая форма ограничения типа значения

where T : struct

В этой форме ключевое слово struct указывает на то, что аргумент T должен быть типа значения.

Общая форма ограничения ссылочного типа

where T : class

В этой форме с оператором where ключевое слово class указывает на то, что аргумент T должен быть ссылочного типа.

Общая форма ограничения типа значения

where T : struct

В этой форме ключевое слово struct указывает на то, что аргумент T должен быть типа значения.

Упражнение 11.1

Продемонстрировать наложение ограничения ссылочного типа.

Общая форма ограничения ссылочного типа

where T : class

В этой форме с оператором where ключевое слово class указывает на то, что аргумент T должен быть ссылочного типа.

Общая форма ограничения типа значения

where T : struct

В этой форме ключевое слово struct указывает на то, что аргумент T должен быть типа значения.

Упражнение 11.1

Продемонстрировать наложение ограничения ссылочного типа.

Упражнение 11.2

Продемонстрировать наложение ограничения типа значения.

Связь между двумя параметрами типа

```
class Gen<T, V> where V : T
```

Связь между двумя параметрами типа

```
class Gen<T, V> where V : T
```

Упражнение 11.3

Реализовать пример использования связи между двумя параметрами типа.

Связь между двумя параметрами типа

```
class Gen<T, V> where V : T
```

Упражнение 11.3

Реализовать пример использования связи между двумя параметрами типа.

Применение нескольких ограничений

```
class Gen<T> where T : Ограничение_1, Ограничение_2, Ограничение_3
```

Связь между двумя параметрами типа

```
class Gen<T, V> where V : T
```

Упражнение 11.3

Реализовать пример использования связи между двумя параметрами типа.

Применение нескольких ограничений

```
class Gen<T> where T : Ограничение_1, Ограничение_2, Ограничение_3
```

Пример:

```
class Gen<T> where T : MyClass, IMyInterface, new()
{
//...
```

Пример:

```
class Test<T>
{
    T obj;
    //...



}
```

Пример:

```
class Test<T>
{
    T obj;
    //...
    obj = null; // подходит только для ссылочных типов
    obj = 0; // подходит только для числовых типов и
             // перечислений, но не для структур
}
```

Пример:

```
class Test<T>
{
    T obj;
    //...
    obj = null; // подходит только для ссылочных типов
    obj = 0; // подходит только для числовых типов и
             // перечислений, но не для структур
}
```

Пример:

```
class Test<T>
{
    T obj;
    //...
    obj = null; // подходит только для ссылочных типов
    obj = 0; // подходит только для числовых типов и
             // перечислений, но не для структур
}
```

Пример:

```
class Test<T>
{
    T obj;
    //...
  • obj = null; // подходит только для ссылочных типов
  • obj = 0; // подходит только для числовых типов и
             // перечислений, но не для структур
}
```

Значения по умолчанию переменной параметра типа

```
class имя_класса<T>
{
    T obj;
    public Test()
    {
      obj = default(T);
    }
}
```

Пример:

```
class Test<T>
{
    T obj;
    //...
  • obj = null; // подходит только для ссылочных типов
  • obj = 0; // подходит только для числовых типов и
             // перечислений, но не для структур
}
```

Значения по умолчанию переменной параметра типа

```
class имя_класса<T>
{
    T obj;
    public Test()
    {
      obj = default(T);
    }
}
```

Упражнение 11.4

Продемонстрировать применение оператора default.

Общий вид

```
struct имя_структуры<параметры> {
```

Общий вид

```
struct имя_структуры<параметры> {
```

Ограничения

```
struct имя_структуры<T> where T : Ограничение { // ...
```

Общий вид

```
struct имя_структуры<параметры> {
```

Ограничения

```
struct имя_структуры<T> where T : Ограничение { // ...
```

Упражнение 11.5

В одном из предыдущих упражнение на обобщения заменить обобщенный класс - структурой.

Обобщенный метод

```
class имя_класса
{
    тип имя_метода<список_параметров>(параметры_метода) {... }
}
```

Обобщенный метод

```
class имя_класса
{
    тип имя_метода<список_параметров>(параметры_метода) {... }
}
```

Пример:

```
public static bool CopyInsert<T>(T[ ] arrayBase, T[ ] arrayTarget)
```

Обобщенный метод

```
class имя_класса
{
    тип имя_метода<список_параметров>(параметры_метода) {... }
}
```

Пример:

```
public static bool CopyInsert<T>(T[ ] arrayBase, T[ ] arrayTarget)
```

Замечание

Отметим два момента касающихся обобщенного метода. Параметр типа объявляется после имени метода, но перед списком его параметров. Обобщенный метод вызывается также как и обычный: без указания аргументов типа.

Обобщенный метод

```
class имя_класса
{
    тип имя_метода<список_параметров>(параметры_метода) {... }
}
```

Пример:

```
public static bool CopyInsert<T>(T[ ] arrayBase, T[ ] arrayTarget)
```

Замечание

Отметим два момента касающихся обобщенного метода. Параметр типа объявляется после имени метода, но перед списком его параметров. Обобщенный метод вызывается также как и обычный: без указания аргументов типа.

Упражнение 11.6

Написать обобщенный статический метод для копирования элементов из одного массива в другой.

Общая форма объявления обобщенного делегата

```
delegate возвращаемый_тип
имя_делегата<список_параметров_типа>(список_аргументов);
```

Общая форма объявления обобщенного делегата

delegate возвращаемый_тип
имя_делегата<список_параметров_типа>(список_аргументов);

Пример:

```
delegate T SomeOp<T>(T v);
```

Общая форма объявления обобщенного делегата

```
delegate возвращаемый_тип
имя_делегата<список_параметров_типа>(список_аргументов);
```

Пример:

```
delegate T SomeOp<T>(T v);
```

Замечание

Отметим, что тип T может служить в качестве возвращаемого типа, несмотря на то, что параметр типа T указывается после имени делегата SomeOp.

Общая форма объявления обобщенного делегата

delegate возвращаемый_тип
имя_делегата<список_параметров_типа>(список_аргументов);

Пример:
delegate T SomeOp<T>(T v);

Замечание

Отметим, что тип T может служить в качестве возвращаемого типа, несмотря на то, что параметр типа T указывается после имени делегата SomeOp.

Упражнение 11.7

Реализовать пример обобщенного делегата.

Объявление обобщенного интерфейса

interface имя_интерфейса<список_параметров_типа>

Объявление обобщенного интерфейса

interface имя_интерфейса<список_параметров_типа>

Объявление класса, реализующего обобщенный интерфейс

class имя_класса<список_параметров_типа> :
имя_интерфейса<список_параметров_типа>

Объявление обобщенного интерфейса

interface имя_интерфейса<список_параметров_типа>

Объявление класса, реализующего обобщенный интерфейс

class имя_класса<список_параметров_типа> :
имя_интерфейса<список_параметров_типа>

Пример:

```
public interface ISeries<T> {...}
class ByTwos<T> : ISeries<T> {...}
```

Объявление обобщенного интерфейса

interface имя_интерфейса<список_параметров_типа>

Объявление класса, реализующего обобщенный интерфейс

class имя_класса<список_параметров_типа> :
имя_интерфейса<список_параметров_типа>

Пример:

```
public interface ISeries<T> {...}
class ByTwos<T> : ISeries<T> {...}
```

Упражнение 11.8

Переделать упражнения 6.1 и 6.2 используя обобщенный интерфейс. В методе GetNext() использовать делегат(что позволит выбирать метод получения следующего члена ряда при создании экземпляра класса). Сами методы определить в отдельном классе.

Пример:

```
//Не годится!
public static bool IsIn<T>(T what, T[ ] obs)
{
    foreach (T v in obs)
        if (v == what) //Ошибка!
            return true;
        return false;
}
```

Пример:

```
//Не годится!
public static bool IsIn<T>(T what, T[ ] obs)
{
    foreach (T v in obs)
        if (v == what) //Ошибка!
            return true;
        return false;
}
```

Замечание

Два объекта параметров обобщенного типа должны реализовывать интерфейс IComparable или IComparable<T> и/или интерфейс IEquatable<T> для того чтобы их можно было сравнивать. В обоих вариантах интерфейса IComparable для этой цели определен метод CompareTo(), а в интерфейсе IEquatable<T> — метод Equals().

Пример:

```
//Не годится!
public static bool IsIn<T>(T what, T[ ] obs)
{
    foreach (T v in obs)
        if (v == what) //Ошибка!
            return true;
        return false;
}
```

Замечание

Два объекта параметров обобщенного типа должны реализовывать интерфейс IComparable или IComparable<T> и/или интерфейс IEquatable<T> для того чтобы их можно было сравнивать. В обоих вариантах интерфейса IComparable для этой цели определен метод CompareTo(), а в интерфейсе IEquatable<T> — метод Equals().

Форма объявления интерфейса IEquatable<T>

```
public interface IEquatable<T>
```

Пример:

```
//Требуется обобщенный интерфейс IEquatable<T>.
public static bool IsIn<T>(T what, T[ ] obs) where T : IEquatable<T>
{
    foreach(T v in obs)
        if(v.Equals(what)) //Применяется метод Equals().
            return true;
        return false;
}
```

Замечание

Два объекта параметров обобщенного типа должны реализовывать интерфейс IComparable или IComparable<T> и/или интерфейс IEquatable<T> для того чтобы их можно было сравнивать. В обоих вариантах интерфейса IComparable для этой цели определен метод CompareTo(), а в интерфейсе IEquatable<T> — метод Equals().

Форма объявления интерфейса IEquatable<T>

```
public interface IEquatable<T>
```

Форма объявления интерфейса IComparable<T>

```
public interface IComparable<T>
```

Форма объявления интерфейса IComparable<T>

```
public interface IComparable<T>
```

Метод CompareTo()

```
int CompareTo(T other)
```

Метод возвращает нуль, если вызывающий объект оказывается равен объекту other; положительное значение, если больше и отрицательное - если вызывающий объект оказывается меньше, чем объект other.

Форма объявления интерфейса IComparable<T>

```
public interface IComparable<T>
```

Метод CompareTo()

int CompareTo(T other)
Метод возвращает нуль, если вызывающий объект оказывается равен объекту other; положительное значение, если больше и отрицательное - если вызывающий объект оказывается меньше, чем объект other.

Пример:

```
//Требуется обобщенный интерфейс IComparable<T>. В данном методе
//предполагается, что массив отсортирован. Он возвращает значение
// true, если значение параметра what оказывается среди элементов
//массива, передаваемых параметру obs.
public static bool InRange<T>(T what, T[ ] obs) where T : IComparable<T>
{
   if(what.CompareTo(obs[0]) < 0 || what.CompareTo(obs[obs.Length-1]) > 0)
       return false;
   return true;
}
```

Форма объявления интерфейса IComparable<T>

```
public interface IComparable<T>
```

Метод CompareTo()

int CompareTo(T other)
Метод возвращает нуль, если вызывающий объект оказывается равен объекту other; положительное значение, если больше и отрицательное - если вызывающий объект оказывается меньше, чем объект other.

Упражнение 11.9

Продемонстрировать применение обобщенных интерфейсов IComparable<T> и IEquatable<T>, используя методы InRange и IsIn. Рассмотреть два массива: один из элементов типа int, второй - из объектов пользовательского типа MyClass

Иерархия обобщенных классов

```
class класс_2<T> : класс_1<T>
```

Иерархия обобщенных классов

```
class класс_2<T> : класс_1<T>
```

Замечание

Параметр типа T указывается в объявлении класса класс_2 и в то же время передается классу класс_1. Это означает, что любой тип, передаваемый классу класс_2, будет передаваться также классу класс_1.

Иерархия обобщенных классов

```
class класс_2<T> : класс_1<T>
```

Замечание

Параметр типа T указывается в объявлении класса класс_2 и в то же время передается классу класс_1. Это означает, что любой тип, передаваемый классу класс_2, будет передаваться также классу класс_1.

Упражнение 11.10

Реализовать обобщенный производный класс с двумя параметрами наследующий от обобщенного базового с одним параметром. В оба класса добавить конструкторы с непустым набором параметров типа.

Иерархия обобщенных классов

```
class класс_2<T> : класс_1<T>
```

Замечание

Параметр типа T указывается в объявлении класса класс_2 и в то же время передается классу класс_1. Это означает, что любой тип, передаваемый классу класс_2, будет передаваться также классу класс_1.

Упражнение 11.10

Реализовать обобщенный производный класс с двумя параметрами наследующий от обобщенного базового с одним параметром. В оба класса добавить конструкторы с непустым набором параметров типа.

Упражнение 11.11

Реализовать обобщенный производный класс от необобщенного базового

Пример:

```
//В этом обобщенном интерфейсе поддерживается ковариантность.
public interface IMyCoVarGenIF<out T>{ T GetObject(); }
//Реализовать интерфейс IMyCoVarGenIF.
class MyClass<T> : IMyCoVarGenIF<T>
{
    T obj;
    public MyClass(T v) { obj = v; }
    public T GetObject() { return obj; }
}
//Создать простую иерархию классов.
class Alpha
{
    string name;
    public Alpha(string n) { name = n; }
    public string GetName() { return name; }
}
class Beta : Alpha
{
    public Beta(string n) : base(n) { }
}
```

```
//Создать ссылку из интерфейса IMyCoVarGenIF на объект типа
MyClass<Alpha>.
//Это вполне допустимо как при наличии ковариантности, так и без нее.
IMyCoVarGenIF<Alpha> AlphaRef = new MyClass<Alpha>(new
Alpha("Alpha #1"));
Console.WriteLine("Имя объекта, на который ссылается переменная
AlphaRef: " + AlphaRef.GetObject().GetName());
//А теперь создать объект MyClass<Beta> и присвоить его переменной
AlphaRef.
//*** Эта строка кода вполне допустима благодаря ковариантности. ***
AlphaRef = new MyClass<Beta>(new Beta("Beta #1"));
Console.WriteLine("Имя объекта, на который теперь ссылается " +
"переменная AlphaRef: " + AlphaRef.GetObject().GetName());
```

```
//Создать ссылку из интерфейса IMyCoVarGenIF на объект типа
MyClass<Alpha>.
//Это вполне допустимо как при наличии ковариантности, так и без нее.
IMyCoVarGenIF<Alpha> AlphaRef = new MyClass<Alpha>(new
Alpha("Alpha #1"));
Console.WriteLine("Имя объекта, на который ссылается переменная
AlphaRef: " + AlphaRef.GetObject().GetName());
//А теперь создать объект MyClass<Beta> и присвоить его переменной
AlphaRef.
//*** Эта строка кода вполне допустима благодаря ковариантности. ***
AlphaRef = new MyClass<Beta>(new Beta("Beta #1"));
Console.WriteLine("Имя объекта, на который теперь ссылается " +
"переменная AlphaRef: " + AlphaRef.GetObject().GetName());
```

Обобщенный интерфейс IMyContraVarGenIF контравариантного типа.

```
public interface IMyContraVarGenIF<in T>
```

Упражнение 11.12

Реализовать пример переопределения виртуального метода в обобщенном классе.

Упражнение 11.12

Реализовать пример переопределения виртуального метода в обобщенном классе.

Упражнение 11.13

Перегрузка методов с параметрами типа может привести к неоднозначности. Продемонстрировать это на примере.

Упражнение 11.12

Реализовать пример переопределения виртуального метода в обобщенном классе.

Упражнение 11.13

Перегрузка методов с параметрами типа может привести к неоднозначности. Продемонстрировать это на примере.

Упражнение 11.14

Продемонстрировать ковариантность и контравариантность в обобщенном интерфейсе.

Упражнение 11.12

Реализовать пример переопределения виртуального метода в обобщенном классе.

Упражнение 11.13

Перегрузка методов с параметрами типа может привести к неоднозначности. Продемонстрировать это на примере.

Упражнение 11.14

Продемонстрировать ковариантность и контравариантность в обобщенном интерфейсе.

Упражнение 11.15

Продемонстрировать конвариантность и контравариантность в обобщенных делегатах.

Простой запрос

```
using System.Linq
...
int[ ] nums = { 1, -2, 3, 0, -4, 5 };
// Простой запрос на получение только положительных значений.
var posNums = from n in nums
              where n > 0
              select n;
```

Простой запрос

```
using System.Linq
...
int[ ] nums = { 1, -2, 3, 0, -4, 5 };
// Простой запрос на получение только положительных значений.
var posNums = from n in nums
              where n > 0
              select n;
```

Общая форма оператора from

```
from переменная_диапазона in источник_данных
```

Простой запрос

```
using System.Linq
...
int[ ] nums = { 1, -2, 3, 0, -4, 5 };
// Простой запрос на получение только положительных значений.
var posNums = from n in nums
              where n > 0
              select n;
```

Общая форма оператора from

from переменная_диапазона in источник_данных

Общая форма оператора where

where булево_выражение

Такое выражение иначе называется предикатом. В запросе можно указывать несколько операторов where.

Простой запрос

```
using System.Linq
...
int[ ] nums = { 1, -2, 3, 0, -4, 5 };
// Простой запрос на получение только положительных значений.
var posNums = from n in nums
              where n > 0
              select n;
```

Общая форма оператора from

from переменная_диапазона in источник_данных

Общая форма оператора where

where булево_выражение

Такое выражение иначе называется предикатом. В запросе можно указывать несколько операторов where.

Оператор select

Все запросы оканчиваются оператором select или group. В данном примере используется оператор select, точно определяющий, что именно должно быть получено по запросу.

Простой запрос

```
using System.Linq
...
int[ ] nums = { 1, -2, 3, 0, -4, 5 };
// Простой запрос на получение только положительных значений.
var posNums = from n in nums
              where n > 0
              select n;
```

Общая форма оператора from

from переменная_диапазона in источник_данных

Общая форма оператора where

where булево_выражение

Такое выражение иначе называется предикатом. В запросе можно указывать несколько операторов where.

Упражнение 12.1

Для массива целых чисел сформировать простой запрос и отобразить его результаты. Внести изменения в массив и повторно отобразить результаты запроса.

Таблица 1: Контекстно-зависимые ключевые слова, используемые в запросах.

Ascending	by	descending	equals
from	in	into	group
join	on	let	orderby
select	where		

Таблица 1: Контекстно-зависимые ключевые слова, используемые в запросах.

Ascending	by	descending	equals
from	in	into	group
join	on	let	orderby
select	where		

Таблица 1: Контекстно-зависимые ключевые слова, используемые в запросах.

Ascending	by	descending	equals
from	in	into	group
join	on	let	orderby
select	where		

Замечание

Запрос должен начинаться с ключевого слова from и оканчиваться ключевым словом select или group. Оператор select определяет тип значения, перечисляемого по запросу, а оператор group возвращает данные группами, причем каждая группа может перечисляться по отдельности.

Оператор where

```
int[ ] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };
var posNums = from n in nums
              where n > 0
              where n < 10
              select n;
```

Оператор where

```
int[ ] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };
var posNums = from n in nums
              where n > 0 & n < 10
              select n;
```

Оператор where

```
int[ ] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };
var posNums = from n in nums
              where n > 0 & n < 10
              select n;
```

Оператор orderby

orderby *элемент* порядок

Оператор where

```
int[ ] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };
var posNums = from n in nums
              where n > 0 & n < 10
              select n;
```

Оператор orderby

orderby *элемент* порядок

Пример:

```
var posNums = from n in nums
              orderby n
              select n;
```

Оператор where

```
int[ ] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };
var posNums = from n in nums
              where n > 0 & n < 10
              select n;
```

Оператор orderby

orderby *элемент* порядок

Пример:

```
var posNums = from n in nums
              orderby n descending
              select n;
```

Оператор where

```
int[ ] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };
var posNums = from n in nums
              where n > 0 & n < 10
              select n;
```

Оператор orderby

orderby *элемент* порядок

Сортировка по нескольким критериям

orderby элемент_A направление, элемент_B направление, элемент_C направление, ...

Оператор where

```
int[ ] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };
var posNums = from n in nums
              where n > 0 & n < 10
              select n;
```

Оператор orderby

orderby *элемент* порядок

Сортировка по нескольким критериям

orderby элемент_A направление, элемент_B направление, элемент_C направление, ...

Упражнение 12.2

Написать программу в которой, для базы данных клиентов банка, формируется запрос с сортировкой по трем критериям: фамилии клиента, имени и остатку на его счете. Вывести результаты на консоль.

Общая форма оператора select

select выражение

Общая форма оператора select

select выражение

Пример:

```
var sqrRoots = from n in nums
               where n > 0
               select Math.Sqrt(n);
```

Общая форма оператора select

select выражение

Упражнение 12.3

Для объектов класса EmailAddress, содержащего два свойства (имя пользователя и его e-mail) сформировать запрос и вывести по его результатам e-mail-ы пользователей.

Общая форма оператора select

select выражение

Упражнение 12.3

Для объектов класса EmailAddress, содержащего два свойства (имя пользователя и его e-mail) сформировать запрос и вывести по его результатам e-mail-ы пользователей.

Упражнение 12.4

Для объектов класса ContactInfo, содержащего три свойства (имя пользователя, его e-mail и телефон) сформировать запрос и вывести по его результатам объекты класса EmailAddress (см. предыдущий пример).

Общая форма оператора select

select выражение

Упражнение 12.3

Для объектов класса EmailAddress, содержащего два свойства (имя пользователя и его e-mail) сформировать запрос и вывести по его результатам e-mail-ы пользователей.

Упражнение 12.4

Для объектов класса ContactInfo, содержащего три свойства (имя пользователя, его e-mail и телефон) сформировать запрос и вывести по его результатам объекты класса EmailAddress (см. предыдущий пример).

Вложенные операторы from

```
var pairs = from n1 in Array_1
            from n2 in Array_2
            select new MyPairs(n1, n2);
```

Общая форма оператора select

select выражение

Упражнение 12.3

Для объектов класса EmailAddress, содержащего два свойства (имя пользователя и его e-mail) сформировать запрос и вывести по его результатам e-mail-ы пользователей.

Упражнение 12.4

Для объектов класса ContactInfo, содержащего три свойства (имя пользователя, его e-mail и телефон) сформировать запрос и вывести по его результатам объекты класса EmailAddress (см. предыдущий пример).

Вложенные операторы from

```
var pairs = from n1 in Array_1
            from n2 in Array_2
            select new MyPairs(n1, n2);
```

Упражнение 12.5

Использовать два вложенных оператора from для составления списка всех возможных результатов бросков двух игральных костей.

Общая форма оператора group

group переменная_диапазона by ключ

Общая форма оператора group

group переменная_диапазона by ключ

Замечание

Результатом выполнения оператора group является последовательность, состоящая из элементов типа IGrouping<TKey, TElement>, т.е. обобщенного интерфейса, объявляемого в пространстве имен System.Linq.

Общая форма оператора group

group переменная_диапазона by ключ

Замечание

Результатом выполнения оператора group является последовательность, состоящая из элементов типа IGrouping<TKey, TElement>, т.е. обобщенного интерфейса, объявляемого в пространстве имен System.Linq.

Замечание

Типом переменной запроса, возвращающего группу, является IEnumerable<IGrouping<TKey, TElement>>. В интерфейсе IGrouping определено также доступное только для чтения свойство Key, возвращающее ключ, связанный с каждой коллекцией.

Общая форма оператора group

group переменная_диапазона by ключ

Пример:
Garage - массив объектов Car с тремя свойствами:
имя владельца, марка машины и ее цвет

```
var MyCars = from SomeCar in Garage
                where SomeCar.Owner == "Имя владельца"
                group SomeCar by SomeCar.Model;
foreach (var Cars in MyCars)
{
    Console.WriteLine("Машины по маркам: " + Cars.Key);
    foreach (var Car in Cars)
        Console.WriteLine("Марка: " + Car.Model + "цвет: " + Car.Color);
    Console.WriteLine();
}
```

Общая форма оператора group

group переменная_диапазона by ключ

Пример:
Garage - массив объектов Car с тремя свойствами:
имя владельца, марка машины и ее цвет

```
var MyCars = from SomeCar in Garage
                where SomeCar.Owner == "Имя владельца"
                group SomeCar by SomeCar.Model;
foreach (var Cars in MyCars)
{
    Console.WriteLine("Машины по маркам: " + Cars.Key);
    foreach (var Car in Cars)
        Console.WriteLine("Марка: " + Car.Model + "цвет: " + Car.Color);
    Console.WriteLine();
}
```

Упражнение 12.6

Для массива содержащего список веб-сайтов (в формате string), сформировать запрос, в котором этот список группируется по имени домена самого верхнего уровня, например .org или .com.

Общая форма оператора into

into имя тело_запроса

где имя обозначает конкретное имя переменной диапазона, используемой для циклического обращения к временному результату в продолжении запроса, на которое указывает тело_запроса.

Общая форма оператора into

into имя тело_запроса
где имя обозначает конкретное имя переменной диапазона, используемой для циклического обращения к временному результату в продолжении запроса, на которое указывает тело_запроса.

Замечание

Оператор into используется вместе с оператором select или group и называется продолжением запроса, поскольку он продолжает запрос для получения окончательного результата

Общая форма оператора into

into имя тело_запроса
где имя обозначает конкретное имя переменной диапазона, используемой для циклического обращения к временному результату в продолжении запроса, на которое указывает тело_запроса.

Замечание

Оператор into используется вместе с оператором select или group и называется продолжением запроса, поскольку он продолжает запрос для получения окончательного результата

Пример:

```
var MyCars = from Car in Garage
             where Car.Owner == "Имя владельца"
             group Car by Car.Model;
             into ws
             where ws.Count() > 2
             select ws;
```

Общая форма оператора let

let имя = выражение
где имя обозначает идентификатор, получающий значение, которое дает выражение.

Общая форма оператора let

let имя = выражение
где имя обозначает идентификатор, получающий значение, которое дает выражение.

Пример:
Garages - массив объектов Garage с двумя свойствами:
Адрес владельца и массив объектов Car

```
var MyCars = from MyGarage in Garages
             where MyGarage.Address == "Адрес владельца "
             let Garage = MyGarage.Cars
             from SomeCar in Garage
             where SomeCar.Owner == "Имя владельца"
             group SomeCar by SomeCar.Model;
```

Общая форма оператора let

let имя = выражение
где имя обозначает идентификатор, получающий значение, которое дает выражение.

Пример:
Garages - массив объектов Garage с двумя свойствами:
Адрес владельца и массив объектов Car

```
var MyCars = from MyGarage in Garages
             where MyGarage.Address == "Адрес владельца "
             let Garage = MyGarage.Cars
             from SomeCar in Garage
             where SomeCar.Owner == "Имя владельца"
             group SomeCar by SomeCar.Model;
```

Упражнение 12.7

Сформировать запрос на получение отсортированной последовательности символов, возвращаемых из массива строк.

Общая форма оператора join

```
from переменная_диапазона_A in источник_данных_A
join переменная_диапазона_B in источник_данных_B
on переменная_диапазона_A.свойство equals
переменная_диапазона_B.свойство.
```

Общая форма оператора join

from переменная_диапазона_A in источник_данных_A
join переменная_диапазона_B in источник_данных_B
on переменная_диапазона_A.свойство equals
переменная_диапазона_B.свойство.

Упражнение 12.8

Написать программу для учета товаров на складе. Реализовать класс с двумя свойствами: наименование и инвентарный номер товара и класс со свойствами: инвентарный номер товара и количество единиц товара на складе. Сформировать запрос по результатам которого будет создан массив объектов содержащих информацию о наименование товара и количестве единиц на складе.

Общая форма оператора join

from переменная_диапазона_A in источник_данных_A
join переменная_диапазона_B in источник_данных_B
on переменная_диапазона_A.свойство equals
переменная_диапазона_B.свойство.

Упражнение 12.8

Написать программу для учета товаров на складе. Реализовать класс с двумя свойствами: наименование и инвентарный номер товара и класс со свойствами: инвентарный номер товара и количество единиц товара на складе. Сформировать запрос по результатам которого будет создан массив объектов содержащих информацию о наименование товара и количестве единиц на складе.

Общая форма объявления анонимного типа

new { имя_A = значение_A, имя_B = значение_B, ... }

Общая форма оператора join

from переменная_диапазона_A in источник_данных_A
join переменная_диапазона_B in источник_данных_B
on переменная_диапазона_A.свойство equals
переменная_диапазона_B.свойство.

Упражнение 12.8

Написать программу для учета товаров на складе. Реализовать класс с двумя свойствами: наименование и инвентарный номер товара и класс со свойствами: инвентарный номер товара и количество единиц товара на складе. Сформировать запрос по результатам которого будет создан массив объектов содержащих информацию о наименование товара и количестве единиц на складе.

Общая форма объявления анонимного типа

new { имя_A = значение_A, имя_B = значение_B, ... }

Пример:
var myOb = new { Count = 10, Max = 100, Min = 0 }

Общая форма оператора join

from переменная_диапазона_A in источник_данных_A
join переменная_диапазона_B in источник_данных_B
on переменная_диапазона_A.свойство equals
переменная_диапазона_B.свойство.

Упражнение 12.8

Написать программу для учета товаров на складе. Реализовать класс с двумя свойствами: наименование и инвентарный номер товара и класс со свойствами: инвентарный номер товара и количество единиц товара на складе. Сформировать запрос по результатам которого будет создан массив объектов содержащих информацию о наименование товара и количестве единиц на складе.

Общая форма объявления анонимного типа

new { имя_A = значение_A, имя_B = значение_B, ... }

Упражнение 12.9

Использовать анонимный тип для инкапсуляции результата, возвращаемого оператором join в предыдущем упражнении.

Оператор into

Оператор into можно использовать вместе с оператором join для создания группового объединения, образующего последовательность, в которой каждый результат состоит из элементов данных из первой последовательности и группы всех совпадающих элементов из второй последовательности.

Группа А	
А	X
Б	Y
А	Z

Группа Б	
А	1
Б	2
В	3
А	4
Б	5

join(by name)

А	X	А	1
А	Z	А	1
А	X	А	4
А	Z	А	4
Б	Y	Б	2
Б	Y	Б	5

into

А	X

А	1
А	4

А	Z

А	1
А	4

Б	Y

Б	2
Б	5

Оператор into

Оператор into можно использовать вместе с оператором join для создания группового объединения, образующего последовательность, в которой каждый результат состоит из элементов данных из первой последовательности и группы всех совпадающих элементов из второй последовательности.

Упражнение 12.10

Написать программу иллюстрирующую, как групповое объединение может использоваться для составления списка, в котором различные транспортные средства (автомашины, суда и самолеты) организованы по общим для них категориям транспорта: наземного, морского, воздушного и речного. Использовать анонимный тип.

Таблица 2: Основные методы запроса

Оператор запроса	**Эквивалентный метод запроса**
select	Select(selector)
where	Where(predicate)
orderby	OrderBy(keySelector) или OrderByDescending(keySelector)
join	Join(inner, outerKeySelector, innerKeySelector, resultSelector)
group	GroupBy(keySelector)

Таблица 2: Основные методы запроса

Оператор запроса	Эквивалентный метод запроса
select	Select(selector)
where	Where(predicate)
orderby	OrderBy(keySelector) или OrderByDescending(keySelector)
join	Join(inner, outerKeySelector, innerKeySelector, resultSelector)
group	GroupBy(keySelector)

Замечание

За исключением метода Join(), остальные методы запроса принимают единственный аргумент, который представляет собой объект некоторой разновидности обобщенного типа Func<T, TResult>. Аргумент метода запроса представляет собой метод, совместимый с формой встроенного делегата, объявляемый следующим образом:
delegate TResult Func<in T, out TResult>(T arg)
где TResult обозначает тип результата, который дает делегат, а T — тип элемента.

Простой запрос

```
int[ ] nums = { 1, -2, 3, 0, -4, 5 };
// Простой запрос на получение только положительных значений.
var posNums = from n in nums
              where n > 0 & n < 5
              select n;
```

Формирование запроса с помощью методов расширения

```
int[ ] nums = { 1, -2, 3, 0, -4, 5 };
// Простой запрос на получение только положительных значений.
var posNums = nums.Where(n => n>0 & n<5).Select(r => r);
```

Формирование запроса с помощью методов расширения

```
int[ ] nums = { 1, -2, 3, 0, -4, 5 };
// Простой запрос на получение только положительных значений.
var posNums = nums.Where(n => n>0 & n<5).OrderByDescending(j => j);
```

Формирование запроса с помощью методов расширения

```
int[ ] nums = { 1, -2, 3, 0, -4, 5 };
// Простой запрос на получение только положительных значений.
var posNums = nums.Where(n => n>0 & n<5).OrderByDescending(j => j);
```

Оператор group

```
Garage - массив объектов Car с тремя свойствами:
имя владельца, марка машины и ее цвет
var MyCars = from SomeCar in Garage
                where SomeCar.Owner == "Имя владельца"
                group SomeCar by SomeCar.Model;
```

Формирование запроса с помощью методов расширения

```
int[ ] nums = { 1, -2, 3, 0, -4, 5 };
// Простой запрос на получение только положительных значений.
var posNums = nums.Where(n => n>0 & n<5).OrderByDescending(j => j);
```

Метод расширения GroupBy

```
Garage - массив объектов Car с тремя свойствами:
имя владельца, марка машины и ее цвет
var MyCars = Garage.Where(SomeCar => SomeCar.Owner == "Имя
владельца") .GroupBy(SomeCar => SomeCar.Model);
```

Формирование запроса с помощью методов расширения

```
int[ ] nums = { 1, -2, 3, 0, -4, 5 };
// Простой запрос на получение только положительных значений.
var posNums = nums.Where(n => n>0 & n<5).OrderByDescending(j => j);
```

Метод расширения GroupBy

Garage - массив объектов Car с тремя свойствами:
имя владельца, марка машины и ее цвет

```
var MyCars = Garage.Where(SomeCar => SomeCar.Owner == "Имя
владельца") .GroupBy(SomeCar => SomeCar.Model);
```

Оператор join

```
var Массив_группы_B =
        from a in Группа_A
        join b in Группа_Б
        on a.Свойство_1 equals b.Свойство_1
        select new Группа_B(a.Свойство_2, b.Свойство_2);
```

Формирование запроса с помощью методов расширения

```
int[ ] nums = { 1, -2, 3, 0, -4, 5 };
// Простой запрос на получение только положительных значений.
var posNums = nums.Where(n => n>0 & n<5).OrderByDescending(j => j);
```

Метод расширения GroupBy

Garage - массив объектов Car с тремя свойствами:
имя владельца, марка машины и ее цвет

```
var MyCars = Garage.Where(SomeCar => SomeCar.Owner == "Имя
владельца") .GroupBy(SomeCar => SomeCar.Model);
```

Метод расширения Join

```
var Массив_группы_B =
Группа_A.Join(Группа_Б, a => a.Свойство_1, b => b.Свойство_1,
(a,b) => select new Группа_B(a.Свойство_2, b.Свойство_2));
```

Таблица 3: Дополнительные методы расширения

Метод	Описание
All(predicate)	Возвращает логическое значение true, если все элементы в последовательности удовлетворяют условию, задаваемому параметром predicate
Any(predicate)	Возвращает логическое значение true, если любой элемент в последовательности удовлетворяет условию, задаваемому параметром predicate
Average()	Возвращает среднее всех значений в числовой последовательности
Contains(value)	Возвращает логическое значение true, если в последовательности содержится указанный объект
Count()	Возвращает длину последовательности, т.е. количество составляющих ее элементов
First()	Возвращает первый элемент в последовательности
Last()	Возвращает последний элемент в последовательности
Max()	Возвращает максимальное значение в последовательности
Min()	Возвращает минимальное значение в последовательности
Sum()	Возвращает сумму значений в числовой последовательности

Таблица 3: Дополнительные методы расширения

Метод	Описание
All(predicate)	Возвращает логическое значение true, если все элементы в последовательности удовлетворяют условию, задаваемому параметром predicate
Any(predicate)	Возвращает логическое значение true, если любой элемент в последовательности удовлетворяет условию, задаваемому параметром predicate
Average()	Возвращает среднее всех значений в числовой последовательности
Contains(value)	Возвращает логическое значение true, если в последовательности со-
Cou	...их
First()	Возвращает первый элемент в последовательности
Last()	Возвращает последний элемент в последовательности
Max()	Возвращает максимальное значение в последовательности
Min()	Возвращает минимальное значение в последовательности
Sum()	Возвращает сумму значений в числовой последовательности

Упражнение 12.11

Написать программу для демонстрации методов из таблицы 3.

Общая форма метода расширения

static возвращаемый_тип имя (this тип_вызывающего_объекта ob, список_параметров)

Общая форма метода расширения

static возвращаемый_тип имя (this тип_вызывающего_объекта ob, список_параметров)

Пример:

```
static class MyExtMeths
{
    // Возвратить обратную величину числового значения типа double.
    public static double Reciprocal(this double v)
    {
        return 1.0 / v;
    }
}
class ExtDemo
{
   static void Main()
   {
      double val = 3.0;
       // Вызвать метод расширения Reciprocal().
      Console.WriteLine("Обратная величина {0} равна {1}",val,val.Reciprocal());
   }
}
```

Общая форма метода расширения

static возращаемый_тип имя (this тип_вызывающего_объекта ob, список_параметров)

Пример:

```
static class MyExtMeths
{
    // Возвратить обратную величину числового значения типа double.
    public static double Reciprocal(this double v)
```

Упражнение 12.12

Реализовать несколько методов расширения и использовать их в одном из ранее созданных проектов.

```
}
class ExtDemo
{
   static void Main()
   {
      double val = 3.0;
       // Вызвать метод расширения Reciprocal().
      Console.WriteLine("Обратная величина {0} равна {1}",val,val.Reciprocal());
   }
}
```