Содержание

1 Основные сведения о C#

Язык C# происходит от двух распространённых языков программирования: C и C++. От языка C он унаследовал синтаксис, многие ключевые слова и операторы, а от C++ – усовершенствованную объектную модель. Кроме того, C# близко связан с Java[1].

Имея общее происхождение, но во многом отличаясь, C# и Java похожи друг на друга как близкие, но не кровные родственники. В обоих языках поддерживается распределённое программирование и применяется промежуточный код для обеспечения безопасности и переносимости, но отличия кроются в деталях реализации. Кроме того, в обоих языках предоставляется немало возможностей для проверки ошибок при выполнении, обеспечения безопасности и управляемого исполнения, хотя и в этом случае отличия кроются в деталях реализации. Но в отличие от Java язык C# предоставляет доступ к указателям – средствам программирования, которые поддерживаются в C++. Следовательно, C# сочетает в себе эффективность, присущую C++, и типовую безопасность, характерную для Java. Более того, компромиссы между эффективностью и безопасностью в этом языке программирования тщательно уравновешены и совершенно прозрачны.

Однако по сравнению с C++, C# имеет ряд отличий, упрощающих синтаксис и устраняющих вероятность появления некоторых ошибок в программах.

1.1 Особенности языка

К особенностям языка C# (некоторые особенности заимствованы из C++) можно отнести следующие:

- язык является объектно-ориентированным, поэтому:
  - даже простейшая программа состоит, как минимум, из одного класса;
  - отсутствуют глобальные переменные и методы;
  - простейшие типы являются классами и поддерживают ряд базовых операций;
- язык чувствителен к регистру символов, т.е. идентификаторы `count` и `Count` считаются различными;
- при использовании методов требуется указание после идентификатора метода круглых скобок, даже если метод не имеет параметров;
- переменные могут быть описаны в любом месте программы, при этом область видимости переменных зависит от места (блока программы) их описания;
- все массивы могут изменять размеры (фактически путём создания нового массива);
- идентификаторы переменной и типа могут совпадать;
- используется «сборка мусора», поэтому явное освобождение памяти во многих случаях не используется.

[1] Здесь и далее некоторые фрагменты конспекта взяты из книг:
Шилдт Г. C# 3.0: Полное руководство / Г. Шилдт.–М.: ООО «И.Д. Вильямс»,2010.–992 с.:ил.
Шилдт Г. C# 4.0: Полное руководство / Г. Шилдт.–М.: ООО «И.Д. Вильямс»,2011.–1056 с.:ил.

1.2 Типы данных

Любые данные, используемые в программе, размещаются в оперативной памяти компьютера. Для того чтобы программа могла правильно интерпретировать содержимое памяти, ей требуется знать структуру данных, хранящихся в этой памяти, которая определяется ***типом данных***. Типы данных могут быть как заранее предопределёнными в среде программирования, так и созданными программистом при разработке программы.

В C# все общие типы данных делятся на два вида: ***типы значения*** и ***ссылочные типы***. Переменные типа значения содержат сами значения, в то время как переменные ссылочного типа содержат ссылку на место в памяти, где значения можно найти. Также переменная ссылочного типа может содержать значение ***`null`***, говорящее о том, что переменная ни на что не указывает.

Общая структура типов приведена на рисунке 1.1.

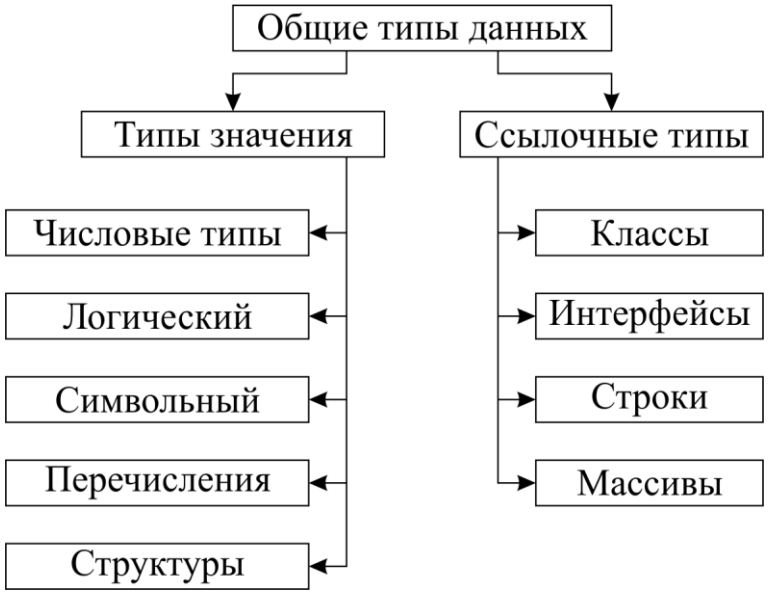

Рисунок 1.1 – Структура типов данных

Целочисленные типы данных. Целочисленные типы данных (см. таблицу 1.1) отличаются друг от друга размером занимаемой памяти и, следовательно, диапазоном целых чисел, которые они могут хранить.

Таблица 1.1 – Характеристика целочисленных типов данных[1]

Наименование	Размер, байт	Диапазон значений
`byte`	1	0…255
`sbyte`	1	–128 … 127

[1] Здесь и далее все значения даются для среды программирования Microsoft Visual C# 2008 Express Edition.

Продолжение таблицы 1.1

Наименование	Размер, байт	Диапазон значений
`char`[1]	2	от U+0000 до U+ffff
`short`	2	–32,768 … 32,767
`ushort`	2	0 … 65 535
`int`	4	–2 147 483 648 … 2 147 483 647
`uint`	4	0 … 4 294 967 295
`long`	8	–9 223 372 036 854 775 808 … 9 223 372 036 854 775 807
`ulong`	8	0 … 18 446 744 073 709 551 615

Вещественные типы данных. Применяются для хранения данных, имеющих дробную часть. В отличие от целочисленных типов, вещественные типы данных отличаются друг от друга не только диапазоном хранимых значений, но и точностью представления числа. Характеристики вещественных типов данных приведены в таблице 1.2.

Таблица 1.2 – Характеристика вещественных типов данных

Наименование	Размер, байт	Приблизительный диапазон значений	Число десятичных знаков
`float`	4	От ±1,5e−45 до ±3,4e38	7
`double`	8	От ±5,0e−324 до ±1,7e308	15-16

Десятичный тип данных `decimal` предназначен для применения в финансовых расчётах. Этот тип имеет разрядность 16 байт для представления числовых значений в пределах от ±1,0e−28 до ±7,9e28. При обычных арифметических вычислений с плавающей точкой характерны ошибки округления десятичных значений. Эти ошибки исключаются при использовании типа `decimal`, который позволяет представить числа с точностью 28-29 десятичных разрядов. Благодаря тому что этот тип данных способен представлять десятичные значения без ошибок округления, он особенно удобен для расчётов, связанных с финансами.

Логический тип данных имеет наименование `bool` и может принимать одно из двух значений: `true` (истина) или `false` (ложь).

Символьный тип данных, предназначенный для хранения одного символа Юникода, имеет наименование `char`. Символ задаётся в апострофах (одиночных кавычках).

Строковый тип данных имеет наименование `string` и предназначен для хранения последовательности символов (данный тип будет рассмотрен ниже в отдельном разделе). Строковые константы задаются в кавычках.

Составные и более сложные типы данных будут рассмотрены ниже.

[1] В большинстве случаев используется для хранения символьных данных. Для целочисленных значений лучше не использовать.

Числовые типы данных обладают некоторыми методами и полями, среди которых можно выделить:

- `Parse(s)` – преобразует строку `s` в число соответствующего типа, например `int.Parse("1")` преобразует строку «1» в число 1 типа `int`. В методе `Parse` могут быть указаны дополнительные параметры, характеризующие допустимые для преобразования форматы строки `s`;
- `TryParse(s, out r)` – преобразует строку `s` в число соответствующего типа и записывает результат в `r`. Метод возвращает логическое значение, показывающее, было ли выполнено преобразование. В методе `TryParse` могут быть указаны дополнительные параметры, характеризующие допустимые для преобразования форматы строки `s`. Например `double.TryParse("1.2", out d)` вернёт `true`, если разделителем дробной и целой части является точка.
- `MinValue, MaxValue` – возвращает минимальное или максимальное значение для заданного типа, например `int.MaxValue` вернёт максимальное значение для типа `int`.

1.3 Переменные

Для описания переменных используется конструкция, формальное описание которой[1] имеет вид:

```
<тип данных> <идентификатор 1>[=<значение идентификатора 1>]
[, <идентификатор2>[=<значение идентификатора 2>] ...];
```

Примеры:

```
double d;
int a, b=10;
int c = b+7;
int d = 0xFF; // d = 255;
```

Если при описании переменной ей сразу присваивается значение, и данная строчка выполняется несколько раз (например, в цикле), то значение присваивается переменной при каждом выполнении строки.

Переменные могут быть типизированы неявно. В этом случае вместо типа данных указывается ключевое слово `var` и требуется обязательное применение блока `=<значение идентификатора N>`.

[1] При описании конструкций будут использоваться следующие обозначения:

- <текст> описывает блок, вместо которого требуется ввести необходимые данные. Однако выделенные жирным угловые скобки обязательны;
- [текст] описывает блок, использование которого необязательно. Однако выделенные жирным квадратные скобки обязательны и не образуют необязательного блока;
- … характеризует возможность повторения необязательного блока, внутри которого это обозначение находится.

Примеры:

```
var d=1.2;
var i=7;
var c='h';
```

Тип переменной определяется по заданному значению, причём для целых значений используется тип `int` (или `long`, в зависимости от значения), а для вещественных – `double`. Чтобы указать другие типы, после значения указывается суффикс, например, в следующем объявлении

```
var a=1.2F;
```

переменная a будет иметь тип `float`.

Применимы следующие суффиксы:

- `u` или `U` – для типов `uint`, `ulong`;
- `l` или `L` – для типов `long`, `ulong`;
- `ul` , `lu` и их любые комбинации с учётом регистра – для типа `ulong`;
- `f` или `F` – для типов `float`;
- `d` или `D` – для типов `double`;
- `m` или `M` – для типов `decimal`.

В одной строке нельзя выполнить неявное типизирование двух и более переменных, т.е. следующая строка будет ошибочной

```
var a = 5, b = 7;
```

1.4 Константы (литералы)

Для описания констант используется конструкция, аналогичная описанию переменных, но перед указанием типа данных указывается модификатор `const`. При этом блоки `=<значение идентификатора N>` являются обязательными.

Примеры:

```
const double d=5.3;
const int a=7, b=8;
```

1.5 Операторы, используемые при построении выражений

Для получения новых значений в любом языке программирования используются выражения, состоящие из операндов и операторов. При построении сложных выражений требуется учитывать приоритеты операторов, а также порядок вычисления операторов одного приоритета

Операторы языка C#, используемые в выражениях, а также их приоритеты и порядок вычисления, приведены в таблице 1.3.

Таблица 1.3 – Операторы C#, используемые при построении выражений

Приоритет	Оператор	Описание
1	`()`	обычные скобки или скобки при вызове функций
	`[]`	обращение к элементам массива
	`.`	доступ к членам класса
	`++`	постфиксное увеличение
	`--`	постфиксное уменьшение
	`->`	разыменование указателя и доступ к члену
	`new`	создание объекта и вызов конструктора
	`typeof`	используется для получения объекта `System.Type` для типа
	`checked`	включение проверки переполнения при выполнении арифметических операций и преобразований с данными целого типа
	`unchecked`	подавление проверки переполнения при выполнении арифметических операций и преобразований с данными целого типа
2	`++`	префиксное увеличение
	`--`	префиксное уменьшение
	`~`	бинарная инверсия
	`!`	отрицание
	`-`	унарный минус
	`+`	унарный плюс
	`&`	получение адреса
	`()`	приведение типа
	`true`	оператор true
	`false`	оператор false
	`sizeof`	получение размера типа данных
3	`*`	умножение
	`/`	деление. Если оба операнда целочисленные, то будет производиться целочисленное деление; в противном случае – деление с получением вещественного числа
	`%`	остаток от деления (в т.ч. дробных чисел)
4	`+`	сложение
	`-`	вычитание
5	`<< / >>`	сдвиг влево / вправо
6	`> / <`	больше / меньше
	`>= / <=`	больше или равно / меньше или равно
	`is`	проверка типа
	`as`	преобразование типа
7	`== , !=`	равно / не равно
8	`&`	логическое «И» (полное)
9	`^`	логическое «исключающее ИЛИ»
10	`|`	логическое «ИЛИ» (полное)

Продолжение таблицы 1.3

Приоритет	Оператор	Описание
11	`&&`	логическое «И» (укороченное)
12	`||`	логическое «ИЛИ» (укороченное)
13	`??`	поддержка значения `null`
14	`? :`	условный оператор
15	`=`	присваивание
	`+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`	присваивание с выполнением действия
16	`=>`	лямбда-оператор

1.6 Класс Object

Данный класс является корнем иерархии всех типов и обладает рядом базовых методов, доступных для использования и часто переопределяемых в классах-потомках. К некоторым из этих методов относятся:

- `Equals(Object obj)` – определяет, равны ли между собой текущий объект и объект `obj`. Имеется также вариант метода с двумя параметрами `Equals(Object objA, Object objB)`, сравнивающий объекты `objA` и `objB` (при этом, обращение к методу должно осуществляться через тип данных). Результатом является логическое значение. Например:

```
int a=6, b=5, c=5;
bool d = a.Equals(b);       // d = false
bool e = int.Equals(b, c); // e = true
```

- `ToString()` – возвращает строковое представление объекта. Например:

```
int a=6;
string s = a.ToString();  // s = "6"
```

Также многие классы имеют метод `CompareTo(Object obj)`, позволяющий сравнивать текущий объект с объектом `obj`. Метод возвращает целое значение, которое в зависимости от результата сравнения:

- меньше нуля, если текущий объект меньше объекта, заданного в параметре;
- равно нулю, если объекты равны;
- больше нуля, если текущий объект больше объекта, заданного в параметре;

Например:

```
int a=7, b=5, c=5, d=2;
int e = b.CompareTo(a); // e = -1 (<0)
```

```
int f = b.CompareTo(c); // f = 0  (=0)
int g = b.CompareTo(d); // g = 1  (>0)
```

1.7 Класс Math

Класс `Math` обеспечивает доступ к ряду математических функций и констант, некоторые из которые приведены в таблице 1.4.

Таблица 1.4 – Некоторые методы и константы класса `Math`

Наименование	Описание	Тип результата
`Abs(X)`	абсолютное значение числа `X`	тип операнда
`Acos(X)`	арккосинус числа `X`	`double`
`Asin(X)`	арксинус числа `X`	`double`
`Atan(X)`	арктангенс числа `X`	`double`
`Atan2(Y,X)`	арктангенс отношения `Y/X`	`double`
`Cos(X)`	косинус числа `X`	`double`
`Cosh(X)`	гиперболический косинус числа `X`	`double`
`DivRem(A,B,out R)`	целочисленное деление `A/B`. Параметр `R` возвращает остаток от деления, например, `c = Math.DivRem(7,3,out r);// c=2, r=1`	целое
`Exp(X)`	возведение числа *e* в степень `X`	`double`
`Log(X[,A])`	натуральный логарифм (или логарифм по основанию `A`) числа `X`	`double`
`Log10(X)`	десятичный логарифм числа `X`	`double`
`Max(X,Y)`	наибольшее среди двух чисел `X` и `Y`	тип операнда
`Min(X,Y)`	наименьшее среди двух чисел `X` и `Y`	тип операнда
`Pow(X,Y)`	возведение числа `X` в степень `Y`	`double`
`Round(X[,N][,M])`	округление числа `X` до ближайшего целого, а в случае указания числа `N` – до `N` знаков после запятой. Параметр `M` может задавать метод округления в случае, если число находится точно по середине между двумя возможными результатами (например, при округлении числа 1,5 до целого)	`double`
`Sign(X)`	знак числа `X`: • -1, если число меньше нуля; • 0, если число равно нулю; • 1, если число больше нуля	`int`

Продолжение таблицы 1.4

Наименование	Описание	Тип результата
`Sin(X)`	синус числа `X`	`double`
`Sinh(X)`	гиперболический синус числа `X`	`double`
`Sqrt(X)`	квадратный корень из числа `X`	`double`
`Tan(X)`	тангенс числа `X`	`double`
`Tanh(X)`	гиперболический тангенс числа `X`	`double`
`Truncate(X)`	целая часть числа `X`	`double`
`E`	константа *e*	`double`
`PI`	константа π	`double`
Примечание: параметры всех тригонометрических функций задаются в радианах		

При обращении к членам класса `Math` требуется указывать сам класс, например, `double c = Math.Cos(Math.PI);`.

1.8 Класс Convert

Класс `Convert` предназначен для преобразования значения одного базового типа данных к другому базовому типу данных.

В таблице 1.5 приведены некоторые методы класса.

Таблица 1.5 – Некоторые методы класса `Convert`

Наименование	Описание	Тип результата

`ChangeType(O,T)`	возвращает объект с типом `T` и значением, эквивалентным заданному объекту `O`, например:
	`double d=-2.345;`
	`int i = (int)Convert.ChangeType (d, typeof(int));`
	`// i = -2`
		тип `T`

`To<тип>(<значение>)`	преобразует `<значение>` в тип данных `<тип>`, например:
	`double d=-2.345;`
	`string s = Convert.ToString(d);`
	`// s = "-2.345"`
		`<тип>`

При обращении к членам класса `Convert` требуется указывать сам класс, например, `int i = Convert.ToInt32(s);`.

1.9 Пространство имён

Пространство имён определяет область объявлений, в которой допускается хранить одно множество имён отдельно от другого. По существу, имена, объявленные в одном пространстве имён, не будут вступать в конфликт с аналогичными именами, объявленными в другой области.

Для каждой программы на C# автоматически предоставляется используемое по умолчанию глобальное пространство имён. Но во многих реальных программах приходится создавать собственные пространства имён или же организовать взаимодействие с другими пространствами имён.

Пространство имён объявляется с помощью ключевого слова `namespace`. Ниже приведена общая форма объявления пространства имён.

```
namespace <имя> { <члены> }
```

`<имя>` обозначает конкретное имя объявляемого пространства имён, а `<члены>` – все допустимые для C# конструкции (структуры, классы, перечисления и т.д.).

Для подключения пространства имён используется директива `using`, формальная запись которой имеет вид:

```
using <имя используемого пространства имен>;
```

Директива `using` может не использоваться вообще, однако в этом случае потребуется каждый раз использовать имя пространства имён при обращении к его членам. Например, если не указать использование пространства имён `System`, то вместо строки:

```
double d = Math.Sin(1);
```

придётся использовать строку

```
double d = System.Math.Sin(1);
```

Директива `using` может использоваться для создания псевдонима пространства имён. Формальное описание создания псевдонима имеет вид:

```
using <имя псевдонима> = <имя пространства имен>;
```

Пространства имён имеют аддитивный характер, т.е. если объявлено два пространства имён с одинаковым именем, то они складываются в единое пространство имён. Например, если в первом файле имеется описание:

```
namespace NS1
{
  class Class1
  {
    ...;
  }
}
```

```
namespace NS1
{
  class Class2
  {
    ...;
  }
}
```

а во втором файле производится его использование

```
using NS1;
```

то во втором файле доступны оба класса без явного указания пространства имён.

Одно пространство имён может быть вложено в другое, например

```
namespace NS1
{
  class Class1
  {
    ...;
  }
  namespace NS2
  {
    class Class2
    {
      ...;
    }
  }
}
```

Если использование пространства имён описано в виде строки

```
using NS1;
```

то это даёт прямой доступ только к классу `Class1`, и для обращения к классу `Class2` потребуется указание полного имени вложенного пространства имён:

```
Class1 cl1;
NS1.NS2.Class2 cl2;
```

Вложенное пространство имён также может быть использовано в директиве `using`, например:

```
using NS1.NS2;
```

В этом случае будет прямой доступ только к классу `Class2`, и для обращения к классу `Class1` потребуется явное указание имени пространства имён:

```
NS1.Class1 cl1;
Class2 cl2;
```

Пространства имён помогают предотвратить конфликты имён, но не устранить их полностью. Такой конфликт может, в частности, произойти, когда одно и то же имя объявляется в двух разных пространствах имён и затем предпринимается попытка сделать видимыми оба пространства. Допустим, что два пространства имён содержат класс `MyClass`. Если попытаться сделать видимыми оба пространства имён с помощью директив `using`, то имя `MyClass` из первого пространства вступит в конфликт с именем `MyClass` из второго пространства, обусловив появление ошибки неоднозначности.

Первым способом устранения ошибки может служить явное указание пространства имён при обращении к классу.

Второй способ подразумевает использование псевдонима пространства имён `::`, формальное описание которого имеет вид:

```
<псевдоним пространства имен>::<идентификатор>
```

где `<псевдоним пространства имен>` обозначает конкретное имя псевдонима пространства имён, а `<идентификатор>` – имя члена этого пространства. Например, если имеется описание пространств имён

```
namespace NS1
{
  class Class1
  {
    ...;
  }
}
namespace NS2
{
  class Class1
  {
    ...;
  }
}
```

и объявлено их совместное использование

```
using pNS1 = NS1;
using NS2;
```

то описание объекта класса `Class1` пространства имён `NS1` может быть выполнено следующим образом:

```
pNS1::Class1 cl1;
```[1]

Псевдонимы могут быть назначены не только пространству имён, но и, например, классу:

[1] При создании псевдонима само пространство имён не подключается, поэтому строка `Class1 cl1;` будет считаться допустимой и создаст объект класса `Class1` пространства имён `NS2`

```
using pNS1Class1 = NS1.Class1;
...
pNS1Class1 cl1;
```

1.10 Типы, допускающие значение null

На основе типов значений могут быть созданы типы, которые могут представлять правильный диапазон значений для своего базового типа значений и дополнительное пустое значение `null`. При описании такого типа после указания базового типа добавляется символ «?», например:

```
int? i = 5;
double? d = 6.78;
```

При совместном использовании базовых типов и типов, допускающих значение `null`, могут возникнуть ошибки совместимости, т.к. базовому типу нельзя присвоить значение типов, допускающего значение `null`, например:

```
int? i = 5;
int j = i; // Ошибка
```

Данная ошибка может быть исправлена применением метода `GetValueOrDefault()`, который возвращает текущее значение (если оно не `null`) или значение по умолчанию для базового типа (если текущее значение `null`):

```
int j = i.GetValueOrDefault();
```

Также данная ошибка может быть исправлена использованием оператора поддержки значения `null` «??». Формально формат оператора имеет вид:

```
<проверяемое значение> ?? <значение, если null>
```

Если `<проверяемое значение>` имеет значение, отличное от `null`, то результатом работы оператора будет `<проверяемое значение>`, иначе – `<значение, если null>`, например:

```
int? i = null;
int j = i ?? 5; // j = 5
```

Для проверки значения переменной может быть использовано свойство `HasValue`, которое возвращает `true`, если текущее значение не `null`, и `false` в противном случае, например:

```
int? i = null;
bool b = i.HasValue; // b = false
```

При вычислении значений выражений если один из операндов имеет значение `null`, то и результат будет `null`, например:

```
int? i = 5, j = null;
int? k = i+j; // k = null
```

2 Операторы и конструкции C#

В данном разделе рассматриваются операторы, применяемые при построении различных программных конструкций.

2.1 Операторы присваивания

Основным оператором присваивания является оператор =. Формальное описание данного оператора имеет вид:

```
<идентификатор> = <значение>;
```

<идентификатор> должен быть такого типа данных, который может вместить в себя присваиваемое значение, или который знает, как обработать присваиваемое значение.

<значение> может быть числовой константой, переменной или результатом вычисления выражения.

Если требуется изменить значение некоторой переменной с учётом её предыдущего значения, то могут быть использованы операторы присваивания +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=. Данные операторы выполняют указанную перед символом = операцию между операндами, расположенными слева и справа от него, и записывают результат в операнд, указанный слева. Например, выражение `a *= b + c;` равносильно выражению `a = a*(b + c);`

2.2 Приведение типов

При выполнении операторов присваивания (а также других операторов) в некоторых случаях может выполняться приведение (преобразование) типов, например, во фрагменте программы

```
int a=5;
double d=a;
```

во второй строке выполняется неявное (автоматическое) преобразование целого значения в вещественное. Автоматическое преобразование возможно, если:

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

Если требуется выполнить явное преобразование значения переменной или выражения к некоторому типу, то используется конструкция:

```
(<тип данных>)<преобразуемая величина>
```

Естественно, что возможность явного преобразования зависит от типа данных и преобразуемой величины.

Следует учитывать, что при выполнении выражений также производится преобразование типов в следующем порядке (т.е. сначала делается первая проверка, при её невыполнении вторая и т.д.):

- ЕСЛИ один операнд имеет тип `decimal`, ТО и второй 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 продвигается к типу `double`;
- ЕСЛИ один операнд имеет тип `float`, ТО и второй продвигается к типу `float`;
- ЕСЛИ один операнд имеет тип `ulong`, ТО и второй 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 продвигается к типу `long`;
- ЕСЛИ один операнд имеет тип `uint`, а второй имеет тип `sbyte`, `short` или `int`, ТО оба операнда продвигаются к типу `long`;
- ЕСЛИ один операнд имеет тип `uint`, ТО и второй продвигается к типу `uint`;
- ИНАЧЕ оба операнда продвигаются к типу `int`.

Таким образом, минимальный тип, используемый в выражениях – `int`. Поэтому во фрагменте программы

```
byte a=100, b=157, c;
c = a+b;
```

во второй строке переменные `a` и `b` будут преобразованы к типу `int`, при присваивании суммы переменной `c` возникнет ошибка и потребуется явное преобразование к `byte` (`c = (byte)(a+b);`). Кстати, в этом случае значение `c` будет 1.

Если при выполнении арифметических выражений требуется отслеживать переполнение, то может использоваться команда `checked`. В этом случае, при переполнении возникнет исключительная ситуация. Например, предыдущий пример с контролем переполнения записывается следующим образом: `c = checked((byte)(a + b));`

Проверяться на переполнение может не только отдельное выражение, но и блок операторов. В этом случае запись контроля переполнения имеет вид:

```
checked
{
  <проверяемые операторы>
}
```

Для отключения контроля переполнения в отдельных выражениях или блоках операторов используется команда `unchecked`. Синтаксис команды аналогичен синтаксису команды `checked`.

2.3 Операторы инкремента и декремента

При написании программ часто требуется увеличение (уменьшение) значения переменной на 1. В простейшем случае операцию увеличения можно выполнить с помощью конструкции

```
<переменная> = <переменная>+1;
```

Однако в C# (как, впрочем, и в языках-предшественниках) выполнение такой операции упрощено и записывается в виде

```
<переменная>++; или ++<переменная>;
```

Первый оператор называется ***постфиксным инкрементом***, а второй – ***префиксным инкрементом***.

При выполнении одиночной операции никаких различий между ними нет. Однако при использовании операторов в выражениях:

- для префиксного инкремента сначала выполняется инкремент, а потом используется переменная в выражении;
- для постфиксного инкремента сначала используется переменная в выражении, а потом выполняется инкремент.

Например, во фрагменте программы

```
int i=1, b, c;
b = i++;
c = ++i;
```

во второй строке сначала произойдёт присваивание, а потом будет выполнен инкремент (после выполнения строки `b`=1, `i`=2), а в третье строке сначала выполнится инкремент, а потом произойдёт присваивание (после выполнения строки `c`=3, `i`=3).

Аналогично операторам инкремента работают и операторы декремента

```
<переменная>--; или --<переменная>;
```

Использование операторов инкремента и декремента может приводить к плохо читаемому коду, например:

```
int i=1;
i = i++ + ++i; // i = 4;
```

2.4 Операторные скобки {}

Операторные скобки `{}` применяются в случае, когда необходимо объединить несколько операторов в единый сложный оператор. Необходимость в таких дей-

ствиях возникает, когда какой-либо оператор может выполнить только один другой оператор, а требуется выполнение нескольких (см., например, оператор `if`).

Также операторные скобки применяются для обозначения начала и окончания различных блоков программы, например, тела функции.

2.5 Условный оператор if

Условный оператор `if` применяется для реализации разветвления хода выполнения программы на два направления в зависимости от некоторого условия.

Формальное описание данного оператора имеет вид:

```
if (<логическое значение>) <оператор если истина>; [else <оператор
если ложь>;]
```

В качестве `<логическое значение>` могут выступать логические переменные, логические константы или выражения, дающее логическое значение.

Если `<логическое значение>` даёт результат «Истина», то выполняется оператор `<оператор если истина>`, иначе:

- если задан блок `else`, то выполняется `<оператор если ложь>`;
- если блок `else` не задан, то никаких действий не происходит.

`<оператор если истина>` и `<оператор если ложь>` являются одиночными операторами, поэтому если требуется выполнить более одного оператора, то они объединяются с использованием операторных скобок.

Пример 1: найти минимум из двух целочисленных переменных `a`, `b` и сохранить его в переменную `min`.[1]

```
int a, b, min;
a = ???;
b = ???;
if (a < b)
  min = a;
else
  min = b;
```

Пример 2: найти результат деления $\frac{a}{b+c}$ и сохранить его в переменную `res`. При этом, если `b+c` равно 0, то перед расчётом присвоить `b` значение 1, а `c` – значение 0.

```
double a, b, c, res;
a = ???;
b = ???;
c = ???;
```

[1] Здесь и далее приводятся только фрагменты программ, а не программы целиком. В листингах программ блоки `???` обозначают некоторые значения, которые присваиваются переменным. Источник получения этих значений неважен.

```
if (b+c == 0)
{
  b = 1;
  c = 0;
}
res = a/(b+c);
```

2.6 Логические операторы «И» и «ИЛИ»

Полные и укороченные логические операторы «И» и «ИЛИ» дают один и тот же результат, однако укороченные операторы могут работать быстрее и позволяют реализовывать более простые конструкции.

Полные логические операторы выполняют логическое выражение полностью. Например, в выражении `a < b & c > d` сначала будет вычислен результат `a < b`, потом `c > d` и только потом рассчитан результат всего выражения с помощью оператора `&`. Однако, если выражение `a < b` даёт результат `false`, то фактически уже известен конечный результат всего выражения. Укороченные операторы учитывают это обстоятельство и в этом случае не рассчитывают выражение `c > d`.

Как правило, полные логические операторы используются в том случае, если второй и последующие операторы сложного выражения меняют значения переменных. Укороченные операторы могут использоваться для гарантированной блокировки выполнения действий, которые могут привести к ошибке. Например, пусть требуется определить, делится ли значение переменной `n` нацело на значение переменной `m`. Это можно выполнить с помощью условия

```
if (n%m == 0){...}
```

Но в случае равенства `m` нулю при выполнении проверки возникнет ошибка. Её можно предотвратить используя два оператора `if`

```
if (m != 0)
  if (n%m == 0){...}
```

или используя один оператора `if` с укороченным логическим «И»

```
if (m != 0 && n%m == 0){...}
```

Использование полного логического «И» недопустимо, т.к. в этом случае при `m=0` выражение `n%m == 0` все равно будет вычисляться и в нем произойдёт ошибка.

2.7 Условный оператор ? :

Если для расчёта значения некоторой переменной требуется применение условного оператора `if` в виде `if (a) r = b; else r = c;`, то целесообразно

применять условный оператор ? :, так как при этом упрощается запись кода и его результат можно применять в качестве операнда более сложных выражений.

Структура условного оператора имеет вид:

```
<логическое значение> ? <выражение если истина> : <выражение если
ложь>;
```

В качестве <логическое значение> могут выступать логические переменные, логические константы или выражения, дающее логическое значение.

<выражение если истина> и <выражение если ложь> должны давать одинаковый тип результата.

Результат работы оператора должен быть присвоен некоторой переменной или являться частью более сложного выражения.

Пример 1: найти минимум из двух целочисленных переменных a, b и сохранить его в переменную min.

```
min = a < b ? a : b; // Это аналог if (a < b) min = a; else min = b;
```

Пример 2: найти максимум из двух целочисленных переменных a, b и сохранить его удвоенное значение в переменную max.

```
max = 2*(a > b ? a : b);
// Это аналог if (a > b) max = 2*a; else max = 2*b;
```

2.8 Оператор выбора switch и оператор прерывания break

Если требуется выполнить разветвление выполнения программы более чем на два направления, то возможно применение либо нескольких вложенных операторов if, либо оператора выбора switch.

Формальное описание оператора switch имеет вид:

```
switch (<значение switch>)
{
  case <значение 1>:
    <операторы 1>
    break;
  [case <значение 2>:
    <операторы 2>
    break; ...]
  [default:
    <операторы default>
    break;]
}
```

<значение switch> должно быть переменной или выражением целого, строкового, символьного, перечисляемого, логического типа. <значение N> должны быть значениями такого же типа. <значение N> должны быть уникальными.

Пример:

```
int n;
string s;
n = ???;
switch (n)
{
  case 1:
    s = "n=1";
    break;
  case 2:
    s = "n=2";
    break;
  default:
    s = "n<1 или n>2";
    break;
}
```

При выполнении оператора ищется <значение N>, равное <значение switch>. Если такое значение найдено, то выполняется соответствующий блок case. Если значение не найдено, то:

- при наличии блока default выполняется этот блок;
- при отсутствии блока default никаких действий не производится.

Каждый блок case и блок default должны заканчиваться оператором break, которые прерывает дальнейшее выполнение оператора switch. Исключением является ситуация, когда блок case не имеет операторов. Такое решение применяется, когда требуется, чтобы для нескольких значений выполнялись одни и те же действия, например:

```
int n;
string s;
n = ???;
switch (n)
{
  case 1:
  case 2:
    s = "n=1 или n=2";
    break;
  default:
    s = "n<1 или n>2";
    break;
}
```

Вместо оператора break может располагаться оператор goto case <N>, позволяющий перейти к блоку со значением <N>.

Пример: рассчитать процент скидки, если при покупке одного товара скидки нет, при покупке двух товаров – скидка 2%, трёх – 5%, четырёх и более – 10%.

```
int n; // Количество товаров
int c; // Процент скидки
```

```
n = ???;
switch (n)
{
  case 1:
    c = 0;
    break;
  case 2:
    c = 2;
    break;
  case 3:
    c = 5;
    break;
  default:
    c = 10;
    break;
}
```

2.9 Оператор цикла for

Предназначен для реализации итерационных алгоритмов. Формальная структура оператора имеет вид:

```
for ([<инициализация>]; [<условие>]; [<итерация>]) [<оператор>];
```

<инициализация> представляет собой операторы, подготавливающие цикл к работе. Они выполняются один раз до начала работы цикла. Как правило в этих операторах задаются начальные значения «параметров цикла».

<условие> определяет условие выхода из цикла и его результатом должно быть логическое значение. <условие> проверяется перед каждой итерацией цикла, поэтому тело цикла может не выполниться ни разу. Выход из цикла производится, если <условие> имеет значение **false**. Как правило, <условие> должно зависеть от «параметра цикла».

<итерация> определяет действия, выполняемые после каждой итерации цикла. Как правило, в них производится изменение «параметров цикла», причём изменение может осуществляться произвольным образом как в сторону увеличения, так и в сторону уменьшения.

<оператор> представляет собой одиночный оператор, выполняемый на каждой итерации цикла. Если в цикле необходимо выполнять несколько операторов, то используются операторные скобки.

Пример: рассчитать факториал числа *n*.

```
int n;
n = ???;
int f = 1;
for (int i=2; i<=n; i++)
  f *= i;
```

<инициализация> и <итерация> могут выполнять более одного оператора. В этом случае операторы разделяются запятой. Предыдущий пример может быть записан следующим образом:

```
int n,i,f;
n = ???;
for (i=2, f=1; i<=n; f *= i, i++);
```

Приведённый выше фрагмент показывает, что тело цикла может быть пустым. Также пустыми могут быть <инициализация>, <условие>, <итерация>. Например, предыдущий пример может быть записан так:

```
int n;
n = ???;
int f = 1;
int i = 2;
for (; i<=n; )
  f *= i++;
```

Несмотря на то, что в блоках <инициализация> и <итерация> имеется возможность выполнять несколько операторов, при этом оставляя тело цикла пустым, злоупотреблять этим не следует.

2.10 Оператор цикла while

Оператор цикла while фактически представляет собой оператор цикла for, у которого не заданы <инициализация> и <итерация>. Формальная структура данного оператора имеет вид:

```
while (<условие>) <оператор>;
```

Как и в цикле for, <условие> определяет условие выхода из цикла и его результатом должно быть логическое значение. <условие> проверяется перед каждой итерацией цикла, поэтому тело цикла может не выполниться ни разу. Выход из цикла производится, если <условие> имеет значение **false**.

<оператор> представляет собой одиночный оператор, выполняемый на каждой итерации цикла. Если в цикле необходимо выполнять несколько операторов, то используются операторные скобки.

Пример: рассчитать факториал числа *n*.

```
int n;
n = ???;
int i = 1;
int f = 1;
while (++i <= n)
  f *= i;
```

2.11 Оператор цикла do...while

Формальная структура данного оператора имеет вид:

```
do <оператор> while (<условие>);
```

Как и в предыдущих операторах цикла, `<условие>` определяет условие выхода из цикла и его результатом должно быть логическое значение. Однако, `<условие>` проверяется после каждой итерации цикла, поэтому тело цикла выполняется как минимум один раз. Выход из цикла производится, если `<условие>` имеет значение **false**.

`<оператор>` представляет собой одиночный оператор, выполняемый на каждой итерации цикла. Если в цикле необходимо выполнять несколько операторов, то используются операторные скобки.

Пример: рассчитать факториал числа *n*.

```
int n;
n = ???;
int i = 1;
int f = 1;
do
  f *= i++;
while (i <= n);
```

2.12 Операторы прерываний break (для циклов) и continue

Оператор `break`, используемый в операторе `switch`, также может применяться в циклах для их немедленного прерывания. При его использовании управление передаются оператору, следующему за циклом. При этом, если имеется ряд вложенных циклов, то оператор `break` прерывает только тот цикл, в теле которого он находится.

Пример: найти наименьший делитель числа *n*, больший 1:

```
int n;
n = ???;
int i;
for (i=2; i<=n; i++)
  if (n%i == 0)
    break;
```

Оператор `continue` используется для прерывания текущей итерации цикла и перехода к следующей (применяется довольно редко, т.к. как правило имеются более удобные способы написания требуемого кода).

При использовании в теле оператора `for` управление передаётся в блок `<итерация>`, после чего цикл продолжает работать по обычной схеме.

При использовании в теле операторов `while` и `do...while` управление передаётся в блок `<условие>`, после чего циклы продолжает работать по обычной схеме.

2.13 Оператор new

Оператор `new` используется для создания объектов и вызова конструкторов. При создании объектов для них выделяется память, а также вызывается конструктор по умолчанию[1], который инициализирует члены объектов значением по умолчанию. Например, две ниже приведённые строки приводят к одному результату – созданию переменной `i` со значением 0, т.к. конструктор по умолчанию типа `int` присваивает объекту значение 0:

```
int i=0;
int i = new int();
```

Значения по умолчанию для всех числовых типов имеют значение 0, для символов – пустой символ, для строки – `null`, для логического типа – `false`.

2.14 Массивы

Массив представляет собой совокупность переменных одного типа с общим для обращения к ним именем. В языке C# массивы могут быть как одномерными, так и многомерными, хотя чаще всего применяются одномерные массивы. Массивы служат самым разным целям, поскольку они предоставляют удобные средства объединения связанных вместе переменных.

Главное преимущество массива – в организации данных таким образом, чтобы ими было проще манипулировать. Как правило, обработка массива реализуется путём циклического обращения к его элементам.

При создании массива, всем его элементам присваивается значение 0.

2.14.1 Одномерные массивы

Для того чтобы воспользоваться массивом в программе, требуется двухэтапная процедура, поскольку в C# массивы реализованы в виде объектов. Во-первых, необходимо объявить переменную, которая может обращаться к массиву. И во-вторых, нужно создать экземпляр массива, используя оператор `new`. Для объявления одномерного массива обычно применяется следующая общая форма:

```
<тип>[] <идентификатор> = new <тип>[<размер>];
```

[1] Если при создании объекта указываются дополнительные параметры, то вызывается соответствующий им конструктор.

хотя возможно разделение данной строки на две части[1]:

```
<тип>[] <идентификатор>;
<идентификатор> = new <тип>[<размер>];
```

Пример обычного объявления целочисленного массива, состоящего из 10 элементов:

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

Нумерация элементов массива всегда начинается с нуля, поэтому в приведённом выше примере доступны элементы массива с индексами в диапазоне 0÷9.

Для обращения к элементу массива требуется указать идентификатор массива, после которого в квадратных скобках указать индекс требуемого элемента, например, присвоение последнему элементу массива `mas` из приведённого выше примера значения 25 будет записано как:

```
mas[9] = 25;
```

При объявлении массива возможна его инициализация. В этом случае, в команде объявления массива не требуется указания размера массива, так как он вычисляется по количеству введённых значений инициализации. Формально строка объявления с инициализацией имеет вид:

```
<тип>[] <идентификатор> = {<значение 1> [, <значение 2> ...]};
```

`<значение 1> [, <значение 2> ...` должны быть совместимы с `<тип>`.

Пример:

```
int[] mas = {7,12,3,14,65};
```

Также допустимым (хотя и излишним) является использование при инициализации оператора new, например:

```
int[] mas = new int[5] {7,12,3,14,65};
```

Однако в этом случае размер массива должен совпадать с количеством значений инициализации.

2.14.2 Многомерные массивы

Многомерные массивы отличаются использованием более одной размерности во всех операциях. При этом, размерности отделяются друг от друга запятой. Формальное описание многомерного массива может быть задано строкой

[1] Такое разделение во многих случаях бессмысленно, т.к. до указания размера воспользоваться переменной не будет возможности, а объявить переменную можно в любом месте программы.

```
<тип>[,[, ...]] <идентификатор> =
new <тип>[<размер 1>,<размер 2>[,<размер 3> ...]];
```

например, приведённый ниже фрагмент создаёт трёхмерный массив и присваивает одному из элементов значение 999

```
int[,,] mas = new int[3,4,5];
mas[1,2,3] = 999;
```

Количество элементов в многомерном массиве, созданном таким способом, определяется как произведение количеств элементов в каждой размерности (для приведённого выше примера: 3*4*5 → 60).

Многомерные массивы также могут быть инициализированы при создании. При этом, значения инициализации для каждой размерности заключаются в фигурные скобки. Например, строка

```
int[,,] mas = {{{1,2},{3,4},{5,6}},{{7,8},{9,10},{11,12}}};
```

инициализирует трёхмерный массив, имеющий размер размерностей 2,3,2.

2.14.3 Ступенчатые массивы

Ступенчатый массивы представляют собой особый тип многомерных массивов, у которого во второй и последующих размерностях может быть различное количество элементов.

Отличие в описании и использовании ступенчатых массивов, по сравнению с многомерными, заключается в следующем:

- каждая размерность заключается в отдельную пару квадратных скобок (вместо запятых, отделяющих размерности в многомерных массивах);
- т.к. каждая размерность представляет собой отдельный одномерный массив, при создании ступенчатого массива каждая размерность создаётся индивидуально отдельными командами. За одну команду не может быть создано несколько размерностей.

Формально, объявление и начало создания ступенчатого массива может быть записано строкой

```
<тип>[][][] ...] <идентификатор> = new <тип>[<размер 1>][][] ...];
```

Пример: создание ступенчатого массива с тремя размерностями (графическая иллюстрация приведена на рисунке 2.1)

```
int[][][] mas;
mas = new int[2][][];   // 1
mas[0] = new int[2][];  // 2
mas[1] = new int[3][];  // 3
mas[0][0] = new int[4]; // 4
mas[0][1] = new int[2]; // 5
mas[1][0] = new int[3]; // 6
```

```
mas[1][1] = new int[2]; // 7
mas[1][2] = new int[4]; // 8
```

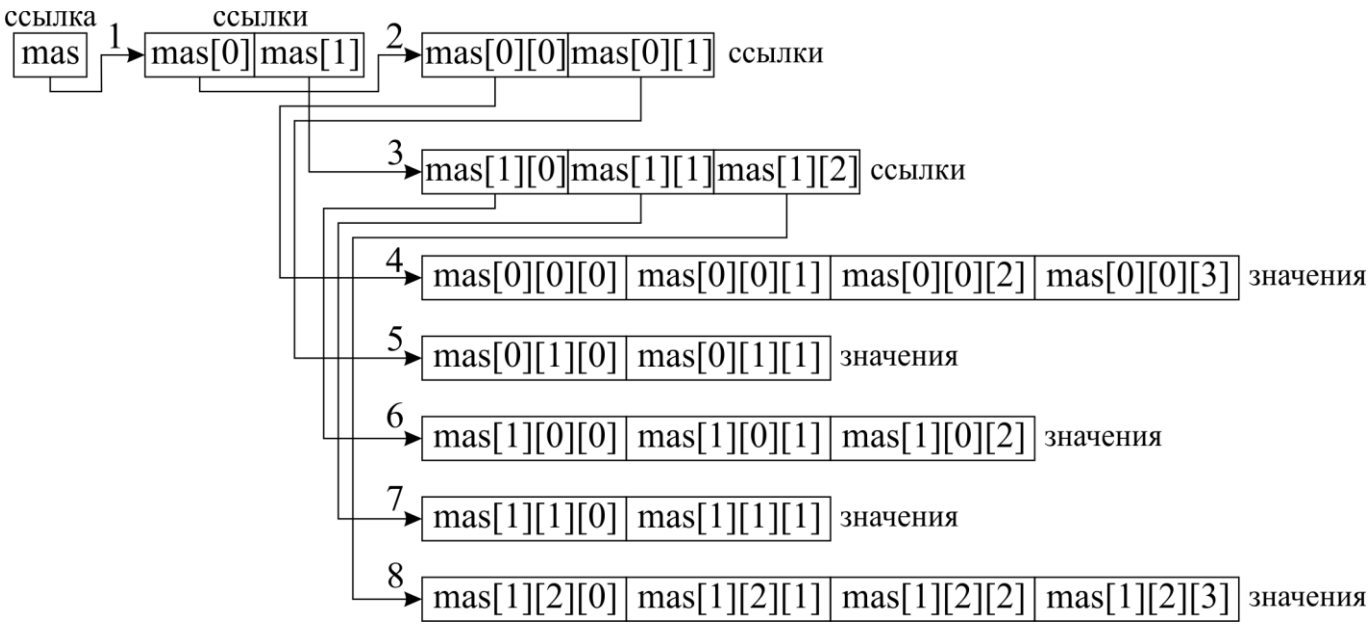

Рисунок 2.1 – Создание ступенчатого массива с тремя размерностями

2.14.4 Работа с массивами как с объектами

Переменная-массив фактически является ссылкой на область памяти, поэтому при создании массивов используется оператора `new`.

Однако значение переменной-массиву может быть присвоено не только путём создания нового объекта, но и путём присвоения ссылки на существующий объект, например фрагмент кода

```
int[] mas1 = {1,2};
int[] mas2 = mas1;
```

создаёт две ссылки на одну и туже область данных. Это можно проверить следующим кодом[1]:

```
mas1[0] = 3;
MessageBox.Show(mas1[0].ToString()+" - "+mas2[0].ToString());
```

Вторая строка кода выведет сообщение «3 - 3», что подтверждает, что обе переменные ссылаются на один и тот же массив.

Повторное создание массива для переменной `mas1` с использованием оператора `new` не изменит размерность существующего массива, а создаст новый массив, что может подтвердить код

```
mas1 = new int[3] {4,5,6};
MessageBox.Show(mas1[0].ToString()+" - "+mas2[0].ToString());
```

[1] Для вывода сообщения используется метод `Show()` класса `MessageBox`, которому в качестве параметра передаётся текст сообщения.

выполнение которого выведет сообщение «4 - 3». Это означает, что переменная `mas1` ссылается на новый массив, а переменная `mas2` – на старый. Если бы не было переменной `mas2`, то при повторном создании массива не осталось бы ни одной ссылки на исходный массив, и он бы был уничтожен в процессе «сборки мусора».

Так как каждый массив является объектом, то он обладает рядом единых для всех массивов свойств и методов, некоторые из которых приведены в таблице 2.1.

Таблица 2.1 – Некоторые свойства и методы массивов

Наименование	Описание

CopyTo (Array array, int index)[1]	Копирует все элементы из текущего одномерного массива в одномерный массив `array`, начиная их размещать начиная с позиции `index`, например:
	`int[] mas = {1,2,3,4,5,6};`
	`int[] mas2 = {7,8,9};`
	`mas2.CopyTo(mas,2);`
	`// mas = {1 2 7 8 9 6}`
	Если все элементы разместить невозможно, возникает исключение.

GetLength (int dimension)	Возвращает количество элементов в заданной размерности, например:
	`int[,] mas = {{1,2,3},{4,5,6}};`
	`int i = mas.GetLength(0); // i = 2`
	`int j = mas.GetLength(1); // j = 3`

GetLowerBound (int dimension)	Возвращает значение индекса нижней границы в заданной размерности, например:
	`int[,] mas = {{1,2,3},{4,5,6}};`
	`int i = mas.GetLowerBound(0); // i = 0`
	`int j = mas.GetLowerBound(1); // j = 0`

GetUpperBound (int dimension)	Возвращает значение индекса верхней границы в заданной размерности, например:
	`int[,] mas = {{1,2,3},{4,5,6}};`
	`int i = mas.GetUpperBound(0); // i = 1`
	`int j = mas.GetUpperBound(1); // j = 2`

Length	Возвращает суммарное количество элементов массива во всех размерностях, например:
	`int[,] mas = {{1,2,3},{4,5,6}};`
	`int i = mas.Length; // i = 6`

Rank	Возвращает ранг массива, например:
	`int[,] mas = {{1,2,3},{4,5,6}};`
	`int i = mas.Rank; // i = 2`

[1] Здесь и далее может приводиться только одна из реализаций метода. Полный список реализаций смотри в справке к Visual C#

2.15 Оператор цикла foreach

Оператор цикла `foreach` может использоваться при обработке массивов в случае, когда требуется обработать ***все*** элементы массива. Формальная структура оператора имеет вид:

```
foreach (<тип> <идентификатор переменной цикла> in
         <идентификатор массива>) <оператор>;
```

`<тип>` и `<идентификатор переменной цикла>` описывают переменную, в которую будет записываться текущее значение элемента массива `<идентификатор массива>` на каждом этапе цикла. Поэтому `<тип>` должен совпадать с типом элемента массива.

При работе цикла последовательно перебираются все элементы массива, независимо от его размерности (т.е. оператор может обрабатывать и многомерные массивы).

Например, для нахождения суммы элементов двухмерного массива может быть использован следующий код:

```
int[,] mas = ???;
int sum = 0;
foreach (int a in mas)
  sum += a;
```

При выполнении цикла переменная `<идентификатор переменной цикла>` доступна только для чтения, поэтому изменить значение элемента массива с её помощью нельзя.

Работу цикла можно прервать с использованием оператора `break`.

2.16 Строки

Строковый тип данных `string` (или `String`) представляет собой ссылку на объект, хранящий последовательность символов – строку.

При работе со строковыми константами используются кавычки, например, объявление и инициализация строки может быть записана как:

```
string s = "Пример";
```

Строка может быть задана с помощью массива символов, например:

```
char[] mas = {'П','р','и','м','е','р'};
string s = new string(mas); // s = "Пример"
```

Несмотря на то, что строки являются ссылками на объекты, некоторые операции с ними выполняются так, как если бы строки хранили сами значения:

- оператор +. Позволяет объединять две строки, например, оператор `s = "При"+"мер";` присваивает строке `s` текст «Пример»;
- операторы == и !=. Сравнивают строки по содержимому, а не по адресу. Например, во фрагменте кода

```
string s1 = "Пример";
string s2 = "При"+"мер";
bool b = (s1 == s2);
```

значение переменной `b` будет `true`. Следует отметить, что другие операторы сравнения при работе со строками недопустимы.

При выполнении операции сложения не требуется преобразование переменных к строке, например:

```
int n = 5;
string s = "Значение переменной n равно "+n;
//s = "Значение переменной n равно 5"
```

Строки являются ***неизменяемыми*** объектами[1]. Поэтому во всех операциях по изменению строки на самом деле создаются новые объекты и разрушаются старые, например, во фрагменте программы:

```
string s = "Пример";
s += " изменения строки";
```

создаётся новый объект-строка с текстом «Пример изменения строки», которая присваивается переменной `s`. Старый объект-строка (с текстом «Пример») будет уничтожен в процессе «сборки мусора».

Каждый элемент строки является ***символом***. Доступ к элементам строки осуществляется с использованием номера элемента (нумерация с нуля), заключённого в квадратные скобки, например, для проверки, является ли первый элемент строки символом «П» может быть использовано условие:

```
if (s[0] == 'П') ...;
```

Доступ к элементам строки не может быть использован для их модификации, т.е. оператор `s[1] = 'a';` является недопустимым.

Класс `string` предоставляет для работы со строками ряд методов и свойств, которые могут вызваны с использованием как переменной данного класса (таблица 2.2), так и с помощью самого класса (таблица 2.3)[2].

[1] Для работы с изменяемыми строками можно использовать класс `StringBuilder` или последовательность действий: перевод в массив символов - обработка - запись в строку.

[2] Большинство функций являются перегруженными, имеющими дополнительные параметры сравнения и локализации. В таблице указаны только некоторые варианты функций.

Таблица 2.2 – Некоторые методы и свойства класса `string`, вызываемые через переменную

Наименование	Описание

`Length`	Возвращает длину строки, например:
	`string s = "Пример";`
	`int i = s.Length; // i = 6`

`CompareTo` `(string s)`	Сравнивает текущую строку со строкой `s` (по алфавиту, а не по длине). Возвращает:
	• -1, если текущая строка расположена раньше (т.е. меньше), чем строка `s`;
	• 0, если строка `s` равна текущей;
	• 1, если текущая строка расположена позднее (т.е. больше), чем строка `s`.
	`string s1 = "абв";`
	`string s2 = "гд";`
	`string s3 = "абв";`
	`int i = s1.CompareTo(s2); // i = -1`
	`int j = s2.CompareTo(s3); // j = 1`
	`int k = s1.CompareTo(s3); // k = 0`

`Contains` `(string s)`	Возвращает `true`, если текущая строка содержит подстроку `s`, или подстрока `s` пустая (в противном случае возвращает `false`). Сравнение осуществляется с учётом регистра и без учёта региональных настроек. Например:
	`string s = "Пример";`
	`bool b1 = s.Contains("рим"); // b1 = true`
	`bool b2 = s.Contains("Рим"); // b2 = false`

`StartsWith` `(string s)`	Возвращает `true`, если текущая строка начинается с подстроки `s` (в противном случае возвращает `false`). Например:
	`string s = "Пример";`
	`bool b1 = s.StartsWith("Прим"); // b1 = true`
	`bool b2 = s.StartsWith("Прин"); // b2 = false`

`EndsWith` `(string s)`	Возвращает `true`, если текущая строка заканчивается подстрокой `s` (в противном случае возвращает `false`). Например:
	`string s = "Пример";`
	`bool b1 = s.EndsWith("мер");  // b1 = true`
	`bool b2 = s.EndsWith("мера"); // b2 = false`

`IndexOf` `(string s)`	Возвращает позицию первого вхождения подстроки `s` в текущей строке. Если подстрока не найдена, возвращается -1. Например:
	`string s = "Пример поиска подстроки";`
	`int i1 = s.IndexOf("по");  // i1 = 7`
	`int i2 = s.IndexOf("пол"); // i2 = -1`

Продолжение таблицы 2.2

Наименование	Описание

LastIndexOf (string s)	Возвращает позицию последнего вхождения подстроки s в текущей строке. Если подстрока не найдена, возвращается -1. Например:
	`string s = "Пример поиска подстроки";`
	`int i1 = s.LastIndexOf("по");  // i1 = 14`
	`int i2 = s.LastIndexOf("пол"); // i2 = -1`

Insert (int index, string s)	Возвращает строку, полученную путём вставки строки s в текущую строку начиная с позиции index. Например:
	`string s1 = "Слон ест мясо.";`
	`string s2 = s1.Insert(5,"не ");`
	`// s2 = "Слон не ест мясо."`

Remove (int sIndex [,int count])	Возвращает строку, полученную путём удаления из текущей строки всех (или count) символов начиная с позиции sIndex (все параметры не должны выходить за пределы строки, в т.ч. sIndex+count). Например:
	`string s1 = "Это не правда.";`
	`string s2 = s1.Remove(4,3); // s2 = "Это правда."`

Replace (string oldS, string newS)	Возвращает строку, полученную путём замены в текущей строке всех подстрок oldS на подстроки newS с учётом регистра и без учёта региональных настроек. Например:
	`string s1 = "трижды три будет 4";`
	`string s2 = s1.Replace("три","два");`
	`// s2 = "дважды два будет 4"`

Substring (int sIndex [,int count])	Возвращает строку, полученную путём извлечения из текущей строки всех (или count) символов начиная с позиции sIndex (все параметры не должны выходить за пределы строки, в т.ч. sIndex+count). Например:
	`string s1 = "Это не правда.";`
	`string s2 = s1.Substring(7,6); // s2 = "правда"`

Split (char[] sep)	Возвращает массив строк, полученный путём разделения текущей строке на подстроки, расположенные между разделителями sep. Если два разделителя расположены в строке подряд (а также, если разделителем является первый или последний символ), то в массив добавляются пустые строки (от этого можно отказаться используя расширенные варианты метода). Например:
	`string s = " один, два,три четыре,";`
	`char[] sep = { ' ', ',' };`
	`string[] mas = s.Split(sep);`
	`// mas = {"", "один", "", "два", "три", "четыре", ""}`

Продолжение таблицы 2.2

Наименование	Описание

ToCharArray ([int sIndex, int count])	Переводит текущую строку в массив символов. Если заданы параметры, то переводятся символы начиная с позиции sIndex в количестве count. Например:
	`string s = "Пример";`
	`char[] mas = s.ToCharArray(1,3);`
	`// mas = ('р','и','м')`

ToLower()
ToUpper()
	Возвращает строку, полученную путём приведения текущей строки к нижнему (верхнему) регистру. Например:
	`string s1 = "ПриМер";`
	`string s2 = s1.ToLower(); // s2 = "пример"`
	`string s3 = s1.ToUpper(); // s3 = "ПРИМЕР"`

Trim ([char**[]** tc])
TrimStart ([char**[]** tc])
TrimEnd ([char**[]** tc])
	Возвращает строку, полученную путём удаления из текущей строки всех начальных (Trim, TrimStart) и конечных (Trim, TrimEnd) пробелов (или символов, заданных в массиве tc). Например:
	`string s1 = "  Пример, ";`
	`char[] tc = {' ',','};`
	`string s2 = s1.Trim(); // s2 = "Пример,"`
	`string s3 = s1.Trim(tc); // s3 = "Пример"`

Таблица 2.3 – Некоторые методы и свойства класса string, вызываемые через сам класс

Наименование	Описание
Empty	Возвращает пустую строку (""). Т.е. строка есть, но состоит из 0 символов. Значение null означает, что строки вообще нет. `string s = String.Empty; // s = ""`

Copy (string s)	Возвращает строку, полученную путём копирования строки s. Например:
	`string s1 = "Пример";`
	`string s2 = String.Copy(s1); // s2 = "Пример"`
	Строка `string s2 = s1;` скопирует не значение, а ***ссылку***!

IsNullOrEmpty()	Возвращает true, если строка s имеет значение null или String.Empty (в противном случае возвращает false). Например:
	`string s = "Пример";`
	`bool b = String.IsNullOrEmpty(s); // b = false`

Продолжение таблицы 2.3

Наименование	Описание

Format (string s, Object a0 [,Object a1 ...])	Возвращает строку, полученную путём форматирования аргументов a0[,a1 ...] с использованием строки форматирования s.
	Строка форматирования содержит произвольный текст с указанием мест, в которые должны быть вставлены отформатированные определённым образом аргументы. Места задаются фигурными скобками, содержащими номер аргумента и, при необходимости, способ форматирования, отделённый от номера двоеточием.
	Способ форматирования состоит из описателя формата и количества значащих цифр или десятичных знаков.
	Некоторые описатели формата для чисел:
	• D или d – десятичный, используемый для целых чисел;
	• E или e – инженерный (экспоненциальный);
	• F или f – для вывода десятичных знаков;
	• N или n – числовой, с разделением разрядов;
	• X или x – шестнадцатеричный;
	• P или p – процентный.
	Например:
	`double d = 2.1;`
	`string s1 = String.Format(`
	`"Результат {0}*{0:F3} равен: {1:F1}",d,d*d);`
	`// s1 = "Результат 2,1*2,100 равен: 4,4"`
	`int i = 400;`
	`string s2 = String.Format(`
	`"Результат {0}*{0:D4} равен: {1:N1}",i,i*i);`
	`// s2 = "Результат 400*0400 равен: 160 000,0"`

При формировании строки возможно внедрение в неё Escape-последовательностей, обеспечивающих дополнительное форматирование строк при выводе. Каждая Escape-последовательность начинается с символа «\», после которого указывается тип последовательности в виде некоторого символа. Например, если строка задана как

```
string s = "Это строка\nиз двух строк";
```

то при выводе данной строки она будет разделена на две: «Это строка» и «из двух строк». Некоторые Escape-последовательности приведены в таблице 2.4.

Использование символа «\» для обозначения Escape-последовательности может привести к неудобствам и ошибкам, например, если имя файла задано в строке следующим образом

```
string s = "C:\new.txt";
```

то часть строки «\n» будет интерпретирована как переход на новую строку, и файл найден не будет.

Таблица 2.4 – Список Escape-последовательностей

Последовательность	Описание
\'	Вставка одиночной кавычки в текст строки
\"	Вставка двойной кавычки в текст строки
\\	Вставка обратной косой черты в текст строки
\n	Переход на новую строку
\r	Возврат каретки
\t	Вставка символа горизонтальной табуляции

Для отказа от использования в строке Escape-последовательностей, перед строкой указывается символ «@», т.е. предыдущий пример может быть записан одним из двух способов:

```
string s = "C:\\new.txt"; или
string s = @"C:\new.txt";
```

Так как при применении символа «@» текст воспринимается без изменений, то в нем могут быть введены символы табуляции, перехода на новую строку и т.п. Например, если строка задана в виде

```
string s = @"Это первая строка
Это вторая строка
Это третья строка";
```

то при выводе такой строки она будет разделена на три.

Единственное преобразование, которое выполняется в строках, перед которыми указана символ «@» – это вставка в текст двойных кавычек. Для этого в месте вставки двойной кавычки она указывается дважды:

```
string s = @"Он сказал ""Привет""";
```

2.17 Перечисления

Перечисление представляет собой набор имён, определяющих все возможные значения которые могут быть назначены переменной.

Использование перечислений повышает читаемость кода программы и снижает вероятность задания переменной недопустимого значения.

Формальное описание перечисления имеет вид:

```
enum <идентификатор перечисления>[: <тип>]
{
  <идентификатор элемента 1>[=<значение 1>]
```

```
  [,<идентификатор элемента 2>[=<значение 2>] ...]
}
```

По умолчанию тип элемента перечисления – `int`. Однако он может быть изменён на другой целочисленный тип путём указания блока `<тип>`. Задаваемые значения должны соответствовать типу перечисления.

Пример: перечисление дней недели:

```
enum DayOfWeek
{
  Monday,
  Tuesday=5,
  Wednesday,
  Thursday,
  Friday=7,
  Saturday,
  Sunday
}

DayOfWeek d;
d = DayOfWeek.Monday;
```

Если элементам перечисления не назначены значения, то они нумеруются последовательно с нуля. Если какому-нибудь элементу назначено значение, то следующий элемент без значения, расположенный сразу за ним, получает значение на 1 больше. Например в приведённом выше примере значения элементов перечисления будут следующими:

```
Monday=0, Tuesday=5, Wednesday=6, Thursday=7, Friday=7, Saturday=8,
Sunday=9
```

Значения, назначаемые элементам перечисления, могут быть абсолютно любыми, например:

```
enum WorldWar2Years
{
  Start=1939,
  End=1945,
  Stalingrad=1942,
  Kursk=1943
}
```

2.18 Обработка исключений

Исключительная ситуация (исключение) – это возникновение в программе ошибочной ситуации того или иного рода, например, деление на ноль, попытка преобразовать в число строку и т.д.

В случае возникновения исключения программа прекращает выполнение текущего блока и выдаёт сообщение об обнаруженной ошибке. Однако перехват и об-

работка возникающих ошибок позволяет улучшить контроль за выполнением программы.

В языке C# исключения представлены в виде классов. Все классы исключений являются производными от встроенного в C# класса `Exception`, являющегося частью пространства имён `System`.

Обработка исключительных ситуаций в C# организуется с помощью четырёх ключевых слов: `try`, `catch`, `throw` и `finally`. Они образуют взаимосвязанную подсистему, в которой применение одного из ключевых слов подразумевает применение другого.

2.18.1 Класс Exception и стандартные исключения

Класс `Exception`, являясь родителем всех классов исключений, предоставляет им единые свойства и методы, некоторыми из которых являются:

- `Message` – текст, описывающий исключение;
- `StackTrace` – позволяет получить стек вызовов для определения места возникновения исключения;
- `GetType()` – возвращает тип исключения. В сочетании с методом `ToString()` имеется возможность получить строковое представление типа исключения.

Некоторые стандартные исключения приведены в таблице 2.5.

Таблица 2.5 – Некоторые стандартные исключения

Наименование	Причина возникновения исключения
`AccessViolationException`	попытка чтения или записи в защищённую область памяти
`ArithmeticException`	ошибки в арифметических действий, а также операциях приведения к типу и преобразования
`DivideByZeroException`	попытка деления на ноль. Для вещественных чисел не возникает, т.к. там используются значения ± бесконечность или нечисловое значение
`OverflowException`	переполнение при выполнении арифметических операций, операций приведения типов и преобразования
`FormatException`	ошибка при преобразовании из одного типа данных в другой, например, при преобразовании из строки в число методами класса `Convert`
`IndexOutOfRangeException`	попытка обращения к элементу массива с индексом, который находится вне границ массива
`InvalidCastException`	недопустимое приведение или явное преобразование типов
`IOException`	ошибки ввода-вывода
`DirectoryNotFoundException`	невозможно найти часть файла или каталога

Продолжение таблицы 2.5

Наименование	Причина возникновения исключения
`DriveNotFoundException`	попытка доступа к недоступному диску или данным совместного использования
`EndOfStreamException`	попытка выполнить чтение за пределами потока
`FileNotFoundException`	попытка доступа к файлу, не существующему на диске
`PathTooLongException`	путь или имя файла превышает максимальную длину, определённую системой
`NullReferenceException`	попытка использовать пустую ссылку, т.е. ссылку, которая не указывает ни на один из объектов

Иерархия некоторых классов исключений имеет вид:

```
Exception
  SystemException
    AccessViolationException
    ArithmeticException
      DivideByZeroException
      OverflowException
    FormatException
    IndexOutOfRangeException
    InvalidCastException
    IOException
      DirectoryNotFoundException
      DriveNotFoundException
      EndOfStreamException
      FileNotFoundException
      PathTooLongException
    NullReferenceException
```

Знание иерархии классов важно для правильной обработки возникающих исключений.

2.18.2 Блок try...catch

Основу обработки исключительных ситуаций в C# составляет пара ключевых слов `try` и `catch`, формальное использование которых имеет вид:

```
try
{
  <блок кода, проверяемый на ошибки>
}
catch (<тип исключения 1> [<переменная исключения 1>])
{
  <обработка исключения типа 1>
}
[catch (<тип исключения 2> [<переменная исключения 2>])
```

```
{
  <обработка исключения типа 2>
} ...]
```

Принцип работы блока `try...catch` следующий. При возникновении ошибки в блоке `<блок кода, проверяемый на ошибки>` дальнейшее выполнение данного блока прекращается и управление передаются тому блоку `catch` (т.е. выполняется блок `<обработка исключения типа N>`), у которого `<тип исключения N>` совпадает с типом возникшей ошибки. Если при выполнении блока `<обработка исключения типа N>` требуется доступ к параметрам исключения, то в блоке `catch` может быть описана переменная `<переменная исключения N>`.

Если в блоке `<блок кода, проверяемый на ошибки>` не возникло ошибок, то все блоки `catch` пропускаются.

При использовании нескольких блоков `catch` все типы исключений, которые они обрабатывают, должны быть разными. При этом, если между двумя типами исключений есть связь «родитель–потомок», то сначала должна быть описана обработка «исключения–потомка».

Пример: найти результат целочисленного деления числа `1000` на число `a`, которое пользователь вводит в компоненте `A_TB` класса `TextBox`. Результат вывести в компонент `R_TB` класса `TextBox`.

```
try
{
  int a = Convert.ToInt32(A_TB.Text);
  R_TB.Text = String.Format(
    "Результат выражения 1000/{0} равен {1}", a, 1000/a);
}
catch (FormatException)
{
  R_TB.Text = "Ошибка: число A должно быть целым";
}
catch (DivideByZeroException)
{
  R_TB.Text = "Ошибка: деление на 0";
}
```

Блоки `try...catch` могут быть вложенными. При этом, если исключение возникает во внутреннем блоке, и там не обрабатывается, то оно передаётся во внешний блок, где может быть обработано[1].

Пример: найти результат целочисленного деления числа `1000` на число `a`, которое пользователь вводит в компоненте `A_TB` класса `TextBox`. Результат вывести в компонент `R_TB` класса `TextBox`. Если число введено неправильно, то считать его значение равным 1.

[1] В общем случае все блоки `try...catch` можно рассматривать как вложенные, т.к. все они находятся в блоке обработки исключений программы, вставляемом компилятором автоматически.

```
try
{
  int a;
  try
  {
    a = Convert.ToInt32(A_TB.Text);
  }
  catch (FormatException)
  {
    a = 1;
  }
  R_TB.Text = String.Format(
    "Результат выражения 1000/{0} равен {1}", a, 1000/a);
}
catch (DivideByZeroException)
{
  R_TB.Text = "Ошибка: деление на 0";
}
```

В блоке `catch` может не указываться тип исключения. В этом случае такой блок располагается самым последним и обрабатывает все типы исключений, не обработанные до него.

2.18.3 Блок try...finally

Блок `try...finally` применяется для создания фрагмента кода, который должен выполниться даже в том случае возникновения исключения. Формальная структура данного блока имеет вид:

```
try
{
  <блок кода, проверяемый на ошибки>
}
finally
{
  <обязательно выполняемый фрагмент кода>
}
```

Алгоритм работы блока `try...finally` следующий. При возникновении ошибки в блоке `<блок кода, проверяемый на ошибки>` дальнейшее выполнение данного блока прекращается и управление передаются блоку `finally` (т.е. выполняется блок `<обязательно выполняемый фрагмент кода>`).

Если в блоке `<блок кода, проверяемый на ошибки>` не возникло ошибок, то управление передаётся блоку `finally`.

Следует отметить, что возникающая ***ошибка останется необработанной***, т.е. будет выдано стандартное сообщение о необработанной ошибке.

Пример: найти результат целочисленного деления числа `1000` на число `a`, которое пользователь вводит в компоненте `A_TB` класса `TextBox`. Результат вывести в

компонент R_TB класса TextBox. Если число введено неправильно или равно 0, то считать его значение равным 1.

```
int a=1;
try
{
  a = Convert.ToInt32(A_TB.Text);
  if (a==0)
    a = 1;
}
finally
{
  R_TB.Text = String.Format(
    "Результат выражения 1000/{0} равен {1}", a, 1000/a);
}
```

2.18.4 Блок try...catch...finally

Чтобы не оставалось необработанных ошибок при использовании блока try...finally, он, как правило, применяется в сочетании с блоком try...catch, путём добавления после него блока finally. При этом, в некоторых случаях блоки catch могут остаться пустыми.

Пример: найти результат целочисленного деления числа 1000 на число a, которое пользователь вводит в компоненте A_TB класса TextBox. Результат вывести в компонент R_TB класса TextBox. Если число введено неправильно или равно 0, то считать его значение равным 1.

```
int a=1;
try
{
  a = Convert.ToInt32(A_TB.Text);
  if (a==0)
    a = 1;
}
catch (Exception)
{
}
finally
{
  R_TB.Text = String.Format(
    "Результат выражения 1000/{0} равен {1}", a, 1000/a);
}
```

2.18.5 Оператор throw

В приведённых выше примерах исключения генерировались программой автоматически. Однако в некоторых случаях бывает полезным сгенерировать исключение вручную. Формальное описание оператора генерации исключения имеет вид:

```
throw <объект класса исключения>;
```

Во многих случаях генерация исключения объединяется с созданием объекта, например:

```
throw new Exception("Неправильно введены данные");
```

Пример: найти результат целочисленного деления числа `1000` на число `a`, которое пользователь вводит в компоненте `A_TB` класса `TextBox`. Число `a` должно быть в диапазоне -100...100. Результат вывести в компонент `R_TB` класса `TextBox`.

```
try
{
  int a = Convert.ToInt32(A_TB.Text);
  if (a < -100 || a > 100)
    throw new Exception("Число A должно быть в диапазоне -100...100");
  R_TB.Text = String.Format(
    "Результат выражения 1000/{0} равен {1}", a, 1000/a);
}
catch (FormatException)
{
  R_TB.Text = "Ошибка: число A должно быть целым";
}
catch (DivideByZeroException)
{
  R_TB.Text = "Ошибка: деление на 0";
}
catch (Exception E)
{
  R_TB.Text = E.Message;
}
```

С помощью оператора `throw` можно повторно сгенерировать исключение после его обработки в блоке `catch` для того, чтобы оно также было обработано во внешнем блоке `catch`. Для этого во внутреннем блоке `catch` последним оператором указывается оператор `throw` без параметров.

Например, предыдущий пример может быть записан следующим образом:

```
try
{
  int a=0;
  try
  {
    a = Convert.ToInt32(A_TB.Text);
```

```
    if (a < -100 || a > 100)
      throw new Exception(
        "Число A должно быть в диапазоне -100...100");
  }
  catch (FormatException)
  {
    MessageBox.Show("Ошибка: число A должно быть целым");
    throw;
  }
  catch (Exception E)
  {
    MessageBox.Show(E.Message);
    throw;
  }
  R_TB.Text = String.Format(
    "Результат выражения 1000/{0} равен {1}", a, 1000/a);
}
catch (Exception)
{
  R_TB.Text = "Результат не найден из-за неправильного ввода числа A";
}
```

3 Классы. Основные понятия

Класс представляет собой тип, содержащий данные, а также методы, позволяющие оперировать этими данными. Все элементы, входящие в класс будем называть ***членами*** класса.

Для каждого объекта класса создаётся своя копия членов-данных, в то время, как члены-методы являются общими для всех объектов.

3.1 Общая схема

Общая схема определения класса имеет вид:

```
[<доступ>] class <идентификатор класса> [: <идентификатор класса-
родителя>]
{
  <члены класса>
}
```

где:

- `<доступ>` – спецификатор доступа и/или модификаторы;
- `<идентификатор класса>` – идентификатор создаваемого класса;
- `<идентификатор класса-родителя>` – идентификатор класса, от которого наследует свойства и поведение создаваемый класс;
- `<члены класса>` – поля, методы, свойства и другие структурные единицы класса.

3.2 Спецификаторы доступа

При объявлении класса или члена класса могут указываться ключевые слова, определяющие тип доступа к ним (спецификаторы доступа):

- `public` – доступен без каких-либо ограничений;
- `protected` – доступен внутри тела класса, а также для потомков класса;
- `internal` – доступен только внутри файлов одной и той же сборки. Внутренний доступ чаще всего используется в разработке на основе компонентов, так как он позволяет группе компонентов взаимодействовать в закрытой форме, не открывая доступ остальной части кода приложения;
- `protected internal` – комбинация доступа `protected` и `internal`;
- `private` – доступен только внутри тела класса, в котором он объявлен.

По умолчанию (т.е. если нет явного указания), члены класса имеют доступ `private`, а классы – `internal`;

Спецификаторы доступа `protected` и `private` не могут применяться к классам верхнего уровня, т.е. классам, не входящим в состав других классов.

Поля класса объявляются, как правило, с помощью спецификаторов доступа `protected` и `private`, т.к. доступ к данным класса извне реализуется с помощью свойств, методов или индексаторов. Использование для доступа к полям спецификатора `public` снижает надёжность программы[1].

3.3 Поля

Поле по своей сути представляет обычную переменную некоторого типа, для которой задан спецификатор доступа. Формальная схема описания полей имеет вид:

```
[<доступ>] <тип> <идентификатор 1>[=<значение идентификатора 1>][,
<идентификатор2>[=<значение идентификатора 2>] ...];
```

Примеры:

```
private double d;
protected int a, b=10;
```

Начальное значение поля может быть задано через конструктор класса, при этом порядок задания начальных значений следующий:

- присвоение полю значения по умолчанию для его типа;
- присвоение значения инициализации, заданного при объявлении поля;
- присвоение значения, указанного в конструкторе.

3.4 Создание объекта и доступ к его членам

В качестве начального примера работы с классами возьмём класс, описанный следующим образом:

```
class Figure
{
  public string sType;   // строковое представление типа фигуры
  public int posX, posY; // координаты фигуры
}
```

Объявление переменной, которая может ссылаться на экземпляр некоторого класса (объект) выполняется обычным для переменных способом, например:

```
Figure firstFigure;
```

Классы являются ссылочным типом, поэтому перед использованием переменных данного типа требуется либо задание ссылки на существующий объект, либо созданием объекта путём использования оператора `new` совместно с вызовом специального метода – ***конструктора класса***, Простейший конструктор (конструктор по

[1] В приводимых в конспекте примерах для уменьшения объёма кода и простоты изложения использование полей со спецификатором доступа `public` будем считать допустимым.

умолчанию) вызывается путём указания имени класса с круглыми скобками после него, например:

```
firstFigure = new Figure();
```

Часто операции по объявлению переменной и созданию объекта объединяются, например:

```
Figure firstFigure = new Figure();
```

Для обращения к члену класса используется оператор доступа «.», разделяющий идентификатор объекта и идентификатор члена, к которому осуществляется доступ. Например, задать строковое представление типа фигуры можно следующим образом:

```
firstFigure.sType = "Квадрат";
```

3.5 Методы

Хотя классы, содержащие только данные, вполне допустимы, у большинства классов должны быть также методы. Методы представляют собой подпрограммы, которые манипулируют данными, определёнными в классе, а во многих случаях они предоставляют доступ к этим данным. Как правило, другие части программы взаимодействуют с классом посредством его методов.

Формальная схема описания метода имеет вид:

```
[<доступ>] <возвращаемый тип> <метод>([<список параметров>])
{
  <тело метода>
}
```

где:

- `<доступ>` – спецификатор доступа и/или модификаторы;
- `<возвращаемый тип>` – тип данных, возвращаемых методом. Если метод не возвращает данных, то указывается ключевое слово **`void`**;
- `<метод>` – идентификатор создаваемого метода;
- `<список параметров>` – список параметров, передаваемых в метод или возвращаемых им;
- `<тело метода>` – код, определяющий набор действий, выполняемых методом.

В списке параметров, параметры разделяются запятыми. Описание каждого параметра имеет вид:

```
[<модификатор параметра>] <тип> <идентификатор параметра>
```

где:

- `<модификатор параметра>` – ключевое слово, определяющее способы работы с параметром;

- <тип> – тип данных, к которому принадлежат параметр;
- <идентификатор параметра> – идентификатор, под которым параметр будет известен внутри метода.

Пример: дополним класс `Figure` методами, позволяющими оперировать с полем `sType` (а также заблокируем прямой доступ к полю `sType` путём задания для него спецификатора доступа `private` и укажем его начальное значение).

```
class Figure
{
  private string sType="Не задан";
  public int posX, posY;
  public string GetSType()
  {
    return sType;
  }
  public void SetSType(string n_sType)
  {
    if (n_sType.Trim() != "")
      sType = n_sType.Trim();
  }
}
```

Каждый метод, у которого возвращаемый тип отличен от `void`, должен содержать оператор `return` (или несколько таких операторов для каждой ветви разветвляющегося кода), с указанием значения, которое возвращается методом. Указанное значение должно быть совместимо с возвращаемым типом. Например, в методе `GetSType()` возвращаемым типом является `string`, поэтому указание в качестве возвращаемого значения текущего значения поля `sType` вполне допустимо. Вызов оператора `return` приводит к ***немедленному выходу*** из метода.

Оператор `return` может применяться и в методах, имеющих возвращаемый тип `void`, для немедленного выхода их метода. В этом случае, никакого значения после оператора `return` не указывается.

Работа с данными через методы, а не напрямую, позволяет создавать более надёжные программы, а также производить первичную обработку данных. Например, в методе `SetSType` из заданного строкового представления типа фигуры удаляются все концевые пробелы и осуществляется контроль, чтобы данное поле имело хоть какое-нибудь значение.

Внесение изменений в класс изменит способы работы с полем `sType`:

```
Figure firstFigure = new Figure();
// firstFigure.sType = "Квадрат"; // Теперь этот оператор недопустим
firstFigure.SetSType("Квадрат"); // Правильный способ задать значение
```

Метод может возвращать не только переменные типов значения, но и созданные в методе объекты (в том числе строки, массивы, т.е. элементы ***любого ссылочного типа***). Например, дополним класс `Figure` методом дублирования объекта:

```
public Figure Duplicate()
{
  Figure tempFigure = new Figure();
  tempFigure.sType = sType;
  tempFigure.posX = posX;
  tempFigure.posY = posY;
  return tempFigure;
}
```

Объект, созданный в методе и возвращаемый из него ***не уничтожается при выходе из метода***. Он будет уничтожен, как только не будет ссылок на него после окончания работы метода.

Параметры, указанные в описании метода, называются ***формальными*** параметрами (параметр `n_sType` в методе `SetSType`), в то время как параметры, указанные при вызове метода – ***фактическими*** (например, слово «Квадрат» в последней строки примера, показанного выше). При выполнении метода каждому формальному параметру присваивается значение фактического, однако метод присваивания зависит от ***модификатора параметра***:

- если модификатор ***не указан***, то формальному параметру присваивается копия фактического. Поэтому изменение формального параметра внутри метода не скажется на значении фактического параметра. В качестве фактического параметра может выступать переменная, константное значение, результат вычислений и т.д.;
- `ref` – параметр передаётся по ссылке, поэтому изменение формального параметра в методе приведёт к изменению фактического параметра. В качестве фактического параметра должна использоваться ***инициализированная*** переменная. При вызове метода перед фактическим параметром также указывается ключевое слово `ref`;
- `out` – параметр передаётся по ссылке, поэтому изменение формального параметра в методе приведёт к изменению фактического параметра. В качестве фактического параметра должна использоваться переменная, которая может быть ***неинициализированной***. В методе формальному параметру ***должно быть*** присвоено значение. При вызове метода перед фактическим параметром также указывается ключевое слово `out`;
- `params` – позволяет определить параметр метода, принимающий аргумент, в котором количество аргументов является переменным. В объявлении метода после ключевого слова `params` дополнительные параметры не допускаются, и в объявлении метода допускается только одно ключевое слово `params`. Параметр должен быть одномерным массивом, а все фактические параметры должны быть совместимы с типом элемента массива. Например:

```
class MyClass
{
  public int Sum(params int[] mas)
  {
    int sum=0;
    for (int i=0; i<mas.Length; i++)
      sum += mas[i];
```

```
      return sum;
    }
  }

  MyClass c = new MyClass();
  int[] m = new int[4] {5,6,7,8};
  int s1 = c.Sum(m);        // s1 = 26
  int s2 = c.Sum(1, 2, 3); // s2 = 6
  int s3 = c.Sum(1, 2.1, 3); // ошибка
```

У метода идентификаторы формальных параметров могут совпадать с идентификаторами полей класса, к которому принадлежит метод. В этом случае, внутри метода видимым является формальный параметр, а для доступа к полю класса может быть использовано ключевое слово **`this`**, например:

```
class MyClass
{
  private int x;
  public void SetX(int x)
  {
    this.x = x;
  }
}
```

Ключевое слово `this` означает объект, который вызвал метод, поэтому оно может быть использовано в любом методе класса, например, в конструкторе.

3.5.1 Перегрузка методов

Язык C# допускает наличие в классе нескольких методов с одинаковым идентификатором. Такое описание называется ***перегрузкой методов***. В этом случае все методы с одинаковым идентификатором должны отличаться количеством и/или типом параметров. Это требование необходимо для того, чтобы компилятор мог распознать необходимый для применения метод. Если ни один метод не подходит для заданного набора параметров, то программа не может быть скомпилирована.

Пример: класс, в котором определено вещественное поле и перегруженный метод, позволяющий выдавать значение этого поля в разных форматах.

```
class MyClass
{
  public double d;
  public string GetD()
  {
    return d.ToString();
  }
  public string GetD(int n)
  {
    return String.Format("{0:F"+n+"}",d);
  }
```

```
  public string GetD(int n, char c)
  {
    return String.Format("{0:"+c+n+"}", d);
  }
}

MyClass c = new MyClass();
c.d = 1234.567;
string s1 = c.GetD();      // s1 = "1234,567"
string s2 = c.GetD(2);     // s2 = "1234,57"
string s3 = c.GetD(2,'N'); // s3 = "1 234,57"
```

При перегрузке методов возможно изменение возвращаемого типа, но только с изменением типа и/или количества параметров.

Пример: класс, имеющий перегруженный метод, выполняющий целочисленное или обычное деление одного числа на другое в зависимости от их типа.

```
 class MyClass
{
  public int Divide(int a, int b)
  {
    return a/b;
  }
  public double Divide(double a, double b)
  {
    return a/b;
  }
}

MyClass c = new MyClass();
int i = c.Divide(7,2);       // i = 3
double d1 = c.Divide(7,2);   // d1 = 3
double d2 = c.Divide(7,2.0); // d2 = 3.5
```

Перегруженные методы могут иметь одинаковое количество и тип параметров в случае, если в одной реализации метода часть параметров передаётся по ссылке (`ref` или `out`), а в другой – по значению.

```
public void Method(int a) {...};
public void Method(ref int a) {...};
```

Однако нельзя перегрузить метод, если в одной реализации передача осуществляется по ссылке `out`, а в другой – по ссылке `ref`.

```
public void Method(out int a) {...};
public void Method(ref int a) {...};
```

3.5.2 Новое в версии C# 4.0

Начиная с версии C# 4.0 параметрам метода, конструктора или индексатора могут быть заданы значения по умолчанию. Это позволяет делать параметры необязательными. Для указания значения по умолчанию необходимо при описании параметра присвоить ему значение, например:

```
public class MyClass
{
  public int Method(int a = 1, int b = 1, int c = 1)
  {
    return a * b * c;
  }
}

MyClass cl = new MyClass();
int i1 = cl.Method(2, 3, 4); // i1 = 24
int i2 = cl.Method(5, 6);    // i2 = 30
int i3 = cl.Method(7);       // i3 = 7
int i4 = cl.Method();        // i4 = 1
```

Если в методе требуются обязательные и необязательные параметры, то в описании метода обязательные параметры ***должны указываться первыми***, т.е. описание метода

```
public int Method(int a = 1, int b, int c = 1) {return a * b * c;}
```

будет недопустимым.

Введение для параметров значений по умолчанию может привести к сложностям при использовании перегрузки методов, например:

```
public class MyClass
{
  public int Method(int a, int b = 1)
  {
    return a * b;
  }
  public double Method(int a, double b = 1.0)
  {
    return a * b;
  }
}

MyClass cl = new MyClass();
int i = cl.Method(2, 3);      // i = 6
double d = cl.Method(2, 3.0); // d = 6.0
d = cl.Method(2);             // Неоднозначность
i = cl.Method(2);             // Неоднозначность
```

В строке `d = cl.Method(2);` возникает неоднозначность, т.к. компилятор не может определить, какой из перегруженных методов использовать. Изменение типа

возвращаемого значения для сужения диапазона возможных значений (`i = cl.Method(2)`) также не приведёт к устранению неоднозначности. В таких случаях, лучше не использовать значения по умолчанию в перегружаемых методах.

Ещё одним новшеством при работе с методами является возможность использовать имена параметров при вызове метода. Формат описания параметра при этом имеет вид:

```
<идентификатор параметра> : <значение>
```

Применение именованных параметров позволяет указывать их не в той последовательности, в которой они описаны в методе, например:

```
public class MyClass
{
  public int Method(int a, int b)
  {
    return a - b;
  }
}

MyClass cl = new MyClass();
int i1 = cl.Method(5, 3);       // i1 = 2
int i2 = cl.Method(a: 5, b: 3); // i2 = 2
int i3 = cl.Method(b: 5, a: 3); // i3 = -2
int i4 = cl.Method(5, b: 3);    // i4 = 2
int i5 = cl.Method(5, a: 3);    // Ошибка
```

Строка (`int i5 = cl.Method(5, a: 3)`) недопустима, т.к. в ней выполняется повторное присваивание значения параметру «`a`».

Именованные параметры могут применяться в комбинации с значениями по умолчанию, что позволяет указывать при вызове метода только требуемые параметры, например:

```
public class MyClass
{
  public double Method(double a = 1.0, double b = 1.0, double c = 1.0)
  {
    return (a - b) / c;
  }
}

MyClass cl = new MyClass();
double d1 = cl.Method(a: 2); // d1 = 1
double d2 = cl.Method(b: 2); // d2 = -1
double d3 = cl.Method(c: 2); // d3 = 0
```

3.6 Конструкторы

Конструкторы представляют собой специальные методы, вызываемые при создании объекта. Каждый класс имеет конструктор по умолчанию[1], не имеющий параметров и устанавливающий значения всех членов-данных в «нулевое» состояние (если не переопределён), а также может иметь произвольное количество конструкторов с разным набором параметров.

Основное назначение конструктора – инициализация членов-данных и подготовка объекта к работе.

Формальная схема конструктора имеет вид:

```
<доступ> <имя_класса>([<список параметров>])
{
  <тело конструктора>
}
```

Чаще всего `<доступ>` имеет значение `public`, т.к. конструктор вызывается при создании объектов из-за пределов класса.

В классе может быть описано несколько конструкторов, т.е. конструкторы, как и любые методы, могут быть перегружены.

Пример: дополним класс `Figure` конструкторами с разными наборами параметров. Будем считать, что значения по умолчанию для полей `posX`, `posY` – 1, а для поля `sType` – «Не задан». Инициализацию полей `posX`, `posY`, ***для примера***, будем проводить ***только*** в конструкторах.

```
class Figure
{
  private string sType="Не задан";
  public int posX, posY;
  public string GetSType()
  {
    return sType;
  }
  public void SetSType(string n_sType)
  {
    if (n_sType.Trim() != "")
      sType = n_sType.Trim();
  }
  public Figure Duplicate()
  {
    return new Figure(sType,posX,posY);
  }
  public Figure() // Переопределение конструктора по умолчанию
  {
    posX = posY = 1;
  }
  public Figure(string n_sType) // Конструктор с одним параметром
```

[1] Конструктор по умолчанию скрывается, если определен хотя бы один другой конструктор.

```
  {
    SetType(n_sType);
    posX = posY = 1;
  }
  // Конструктор с тремя параметрами
  public Figure(string n_sType, int n_posX, int n_posY)
  {
    SetType(n_sType);
    posX = n_posX;
    posY = n_posY;
  }
}

// Вызов конструктора по умолчанию
Figure figure1 = new Figure();
// Вызов конструктора с одним параметром
Figure figure2 = new Figure("Квадрат");
// Вызов конструктора с тремя параметрами
Figure figure3 = new Figure("Круг",5,6);
// Ошибка: конструктора с одним целочисленным параметром нет
// Figure figure4 = new Figure(5);
```

Также в приведённом выше примере был изменён метод `Duplicate`, т.к. имея конструктор с параметрами можно легко создать объект с копией свойств.

Если у класса определено несколько конструкторов, то один из них может вызывать другой. Для этого используется конструкция вида:

```
<доступ> <имя_класса>([<список параметров1>]) : this([<список
параметров2>])
{
  <тело конструктора>
}
```

Здесь `<список параметров1>` определяет набор формальных параметров текущего конструктора, а `<список параметров2>` – набор фактических параметров вызываемого конструктора. При этом, в `<список параметров2>` могут использоваться параметры из `<список параметров1>`.

В некоторых случаях `<тело конструктора>` может оставаться пустым, т.к. все действия выполняются в другом вызываемом конструкторе.

Например, конструкторы без параметров и с одним параметром из приведённого выше примера могут быть описаны с использованием конструктора с тремя параметрами следующим образом:

```
public Figure() : this("Не задан",1,1)
{
}
public Figure(string n_sType) : this(n_sType,1,1)
{
}
```

Если один конструктор вызывает другой, то сначала выполняется вызываемый конструктор, а потом вызывающий.

В отличии от некоторых других языков программирования в C# отсутствует конструктор копии, но его можно создать самостоятельно. Например, вышеприведённый класс может быть дополнен таким конструктором:

```
public Figure(Figure sourceFigure)
{
  sType = sourceFigure.sType;
  posX = sourceFigure.posX;
  posY = sourceFigure.posY;
}
```

Реализованный конструктор копии в сочетании с ключевым словом `this` позволяет улучшить метод `Duplicate` следующим образом:

```
public Figure Duplicate()
{
  return new Figure(this);
}
```

Применение конструктора копий целесообразнее, чем просто конструктора с параметрами, т.к. по определению этот конструктор должен полностью копировать все члены-данные существующего объекта, в то время, как любой конструктор с параметрами может использовать только часть членов-данных класса.

3.7 Деструкторы

В языке C# имеется возможность определить метод, который будет вызываться непосредственно перед окончательным уничтожением объекта. Такой метод называется ***деструктором*** и может использоваться в ряде особых случаев.

Формальная схема деструктора имеет вид:

```
~<имя_класса>()
{
  <тело деструктора>
}
```

Деструктор вызывается непосредственно перед «сборкой мусора». Это означает, что заранее нельзя знать, когда именно следует вызывать деструктор[1]. Кроме того, программа может завершиться до того, как произойдёт «сборка мусора», а следовательно, деструктор может быть вообще не вызван.

Как правило, деструктор должен воздействовать только на переменные экземпляра, определённые в его классе.

[1] Существует возможность принудительно выполнить «сборку мусора», вызвав метод `Collect`, но в большинстве случаев этого следует избегать, потому что это может привести к проблемам с производительностью.

Деструктор не имеет возвращаемого параметра, спецификатора доступа, списка принимаемых параметров и не может быть перегружен.

В силу редкого применения деструкторов в C#, для класса `Figure` описывать деструктор не будем.

3.8 Инициализаторы объектов

Задание начальных свойств объекта возможно при его создании, даже если класс не имеет конструкторов с параметрами. Для этого используется структура с ***инициализаторами***, формальный вид которой:

```
new <идентификатор класса>[(<параметры>)]
{<идентификатор 1>=<значение 1> [,<идентификатор 2>=<значение 2> ...
]};
```

где

- `<идентификатор N>` – идентификатор поля или свойства;
- `<значение N>` – значение, присваиваемое полю или свойству.

При создании объекта с помощью инициализаторов сначала вызывается требуемый конструктор, а затем указанным полям и свойствам присваиваются заданные значения. Порядок следования полей значения не имеет.

Пример: создания объекта класса `Figure` с начальным значением `posY` равным 10 с помощью инициализаторов.

```
Figure firstFigure = new Figure {posY=10};
// sType="Не задан", posX=1, posY=10

Figure firstFigure = new Figure("Круг") {posY=10};
// sType="Круг", posX=1, posY=10
```

3.9 Свойства

Свойство – это программная конструкция, обладающая следующими характеристиками:

- свойство объединяет поле с методами доступа к нему, что обеспечивает контроль за правильностью работы с ним и создание в целом более надёжного кода;
- свойство выглядит для использующего его как поле. Это упрощает синтаксис за счёт лёгкого использования свойства в выражениях и задания значения свойства с помощью операторов присваивания.

Формальная схема описания свойства имеет вид:

```
[<доступ>] <тип> <идентификатор> {get {<код метода доступа get>} set
{<код метода доступа set>}}
```

где:

- <код метода доступа get> – код, определяющий действия, выполняемые при запросе значения свойства;
- <код метода доступа set> – код, определяющий действия, выполняемые при установке значения свойства.

Метод доступа get должен содержать оператор return, определяющий значение, возвращаемое методом get при запросе значения свойства.

Метод доступа set имеет неявно заданный параметр value, который позволяет указать значение, присваиваемое свойству.

Однако свойство само по себе не хранит никакого значения, поэтому оно должно либо быть соотнесено с некоторым полем, объявленным в классе (как правило, с спецификатором доступа private), либо быть расчётным на основе других членов класса.

Пример: перепишем класс Figure с применением свойств. Будут использованы следующие методы именования идентификаторов: поля класса начинаются с символа «_»; свойства – идентификатор поля без символа «_». На положение фигуры накладывается ограничение: координаты не могут быть отрицательными.

```
class Figure
{
  private string _sType;
  private int _posX, _posY;
  public string sType
  {
    get { return _sType; }
    set { if (value.Trim() != "") _sType = value.Trim(); }
  }
  public int posX
  {
    get { return _posX; }
    set { if (value >= 0) _posX = value; }
  }
  public int posY
  {
    get { return _posY; }
    set { if (value >= 0) _posY = value; }
  }
  public Figure Duplicate()
  {
    return new Figure(this);
  }
  public Figure() : this("Не задан",1,1)
  {
  }
  public Figure(string n_sType) : this(n_sType,1,1)
  {
  }
  public Figure(string n_sType, int n_posX, int n_posY)
  {
    _sType = n_sType.Trim() != "" ? n_sType.Trim() : "Не задан";
```

```
    _posX = n_posX >= 0 ? n_posX : 1;
    _posY = n_posY >= 0 ? n_posY : 1;
  }
  public Figure(Figure sourceFigure)
  {
    _sType = sourceFigure._sType;
    _posX = sourceFigure._posX;
    _posY = sourceFigure._posY;
  }
}

Figure firstFigure = new Figure();
// Теперь этот оператор снова допустим, но контроль ввода также
// проводится
firstFigure.sType = "Квадрат";
```

Если при описании свойства не указан метод `set`, то свойство доступно только для чтения, а если метод `get` – то только для записи.

Пример: класс, описывающий квадрат, с расчётным свойством, работающим по принципу «только для чтения».

```
class Square
{
  private double _side;
  public double side // сторона
  {
    get { return _side; }
    set { if (value > 0) _side = value; }
  }
  public double area // площадь
  {
    get { return Math.Pow(_side, 2); }
  }
}
```

Методы `get` и `set` по умолчанию имеют такой же уровень доступа, как и само свойство (т.е. в большинстве случаев `public`). Однако им могут быть указаны собственные спецификаторы доступа. Например, если требуется, чтобы значение свойства могло быть изменено только из самого класса, то свойство может быть описано следующим образом:

```
class MyClass
{
  public int p { get {...} private set{...} }
}
```

В классе могут быть объявлены ***автоматически реализуемые*** свойства. Такие свойства не имеют реализации методов `get` и `set` и не требуют явного указания поля для хранения значения, т.к. такое поле выделяется для свойства автоматически. Пример автоматически реализуемого свойства:

```
class MyClass
{
  public int p { get; set; }
}
```

На значения автоматически реализуемых свойств нельзя наложить никаких ограничений, их нельзя сделать «только для чтения» или «только для записи»[1]. Поэтому применение таких свойств достаточно ограничено.

3.10 Индексаторы

Индексаторы позволяют рассматривать объект класса как массив, т.е. обращаться к его элементам с использованием индексов, указываемых в квадратных скобках.

Формальная структура описания индексатора имеет вид:

```
[<доступ>] <тип> this[<список индексов>] {get {<код метода доступа get>} set {<код метода доступа set>}}
```

где:

- `<список индексов>` – набор индексов, перечисленных через запятую, каждый из которых имеет структуру:

  ```
  <тип> <идентификатор>;
  ```

  чаще всего `<тип>` является целым, хотя может быть любым;
- `<код метода доступа get>` – код, определяющий действия, выполняемые при запросе значения индексатора и использующий индексы, определённые в `<список индексов>`;
- `<код метода доступа set>` – код, определяющий действия, выполняемые при установке значения индексатора и использующий индексы, определённые в `<список индексов>`.

Метод доступа `get` должен содержать оператор `return`, определяющий значение, возвращаемое методом `get` при запросе значения индексатора.

Метод доступа `set` имеет неявно заданный параметр `value`, который позволяет указать значение, присваиваемое индексатору.

Как и свойство, индексатор обычно связан с полем, имеющим спецификатор доступа `private` или `protected`. В большинстве случаев, данное поле является массивом типа `<тип>`, имеющим количество размерностей, равное количеству параметров в `<список индексов>`.

Пример: класс, имеющий одномерный индексатор, возвращающий ние -1, если происходит обращение к элементу с недопустимым индексом.

[1] Подразумевается, что нельзя пропустить метод `get` или `set`. Однако это можно реализовать используя спецификаторы доступа для этих методов

```
class MyClass
{
  private int[] _mas;
  public MyClass(int count)
  {
    if (count > 0)
      _mas = new int[count];
    else
      _mas = new int[0];
  }
  public int this[int index]
  {
    get
    {
      return (index >= 0 && index < _mas.Length) ? _mas[index] : -1;
    }
    set
    {
      if (index >= 0 && index < _mas.Length)
        _mas[index] = value;
    }
  }
}

MyClass cl = new MyClass(10);
cl[5] = 7;
int i1 = cl[4];  // i1 = 0;
int i2 = cl[5];  // i2 = 7;
int i3 = cl[-1]; // i3 = -1;
int i4 = cl[10]; // i4 = -1;
```

Пример: класс описывающий шахматную доску и имеющий двухмерный индексатор с индексами типа char (буквы от 'a' до 'h') и типа byte (цифры от 1 до 8) возвращающий тип фигуры, стоящей на заданной клетке, признак пустой клетки или значение ошибки, если происходит обращение к элементу с недопустимым индексом.

```
enum ChessFigure {King,Queen,Rook,Bishop,Knight,Pawn,Empty,Error};

class MyClass
{
  private string s = "abcdefgh";
  private ChessFigure[,] _mas;
  public MyClass()
  {
    _mas = new ChessFigure[8,8];
    for (int i=0; i<8; i++)
      for (int j=0; j<8; j++)
        _mas[i,j] = ChessFigure.Empty;
  }
  public ChessFigure this[char c,byte n]
  {
    get
```

```
    {
      int i = s.IndexOf(c);
      if (i >= 0 && n >= 1 && n <= 8)
        return _mas[i,n-1];
      else
        return ChessFigure.Error;
    }
    set
    {
      int i = s.IndexOf(c);
      if (i >= 0 && n >= 1 && n <= 8)
        _mas[i,n-1] = value;
    }
  }
}

MyClass cl = new MyClass();
cl['b',3] = ChessFigure.Queen;
ChessFigure f1 = cl['b',3]; // f1 = Queen
ChessFigure f2 = cl['e',8]; // f2 = Empty
ChessFigure f3 = cl['k',2]; // f3 = Error
ChessFigure f4 = cl['h',0]; // f4 = Error
```

Как и у свойства, у индексатора могут отсутствовать методы `get` или `set` для создания индексаторов «только для записи» или «только для чтения» соответственно.

Индексатор может быть перегружен. В этом случае при обращении используется тот из индексаторов, у которого индексы наиболее подходят заданным. Поэтому при перегрузке индексаторов количество и/или тип индексов должны отличаться.

Пример: класс, позволяющий получить либо доступ к заданному элементу матрицы, либо сумму элементов заданной строки матрицы с помощью индексаторов. Считать, что значение элемента или суммы элементов строки при неправильном задании индекса равно -1.

```
class MyClass
{
  private int[,] _mas;
  public MyClass(int countRow,int countCol)
  {
    if (countRow > 0 && countCol > 0)
      _mas = new int[countRow,countCol];
    else
      _mas = new int[0,0];
  }
  public int this[int indexRow,int indexCol]
  {
    get
    {
      if (indexRow >= 0 && indexRow < _mas.GetLength(0) &&
          indexCol >= 0 && indexCol < _mas.GetLength(1))
        return _mas[indexRow,indexCol];
      else
```

```
        return -1;
    }
    set
    {
      if (indexRow >= 0 && indexRow < _mas.GetLength(0) &&
          indexCol >= 0 && indexCol < _mas.GetLength(1))
        _mas[indexRow,indexCol] = value;
    }
  }
  public int this[int indexRow]
  {
    get
    {
      if (indexRow > 0 && indexRow < _mas.GetLength(0))
      {
        int sum=0;
        for (int i=0; i < _mas.GetLength(1); i++)
          sum += _mas[indexRow,i];
        return sum;
      }
      else
        return -1;
    }
  }
}

MyClass cl = new MyClass(3,3);
for (int i=0; i<3; i++)
  for (int j=0; j<3; j++)
    cl[i,j] = i*3+j+1;
int i1 = cl[1,0]; // i1 = 4
int i2 = cl[3,3]; // i2 = -1
int i3 = cl[1];   // i3 = 15
int i4 = cl[4];   // i4 = -1
```

При использовании индексаторов имеются следующие ограничения:

- индексатор нельзя передавать в метод в качестве параметра `ref` или `out`;
- индексатор не может быть `static` (назначение данного ключевого слова будет рассмотрено ниже).

4 Классы. Расширенное использование

4.1 Статические классы и члены классов

Применение ключевого слова `static` при объявлении члена класса указывает, что данный член класса принадлежит всему классу, а не конкретному объекту.

Такой член класса существует все время выполнения программы, даже если ни одного объекта этого класса не создано. Доступ к такому члену класса возможен ***только*** через сам класс, а не через объекты класса.

Пример: создание класса, подсчитывающего количество объектов этого класса, созданных за время выполнения программы.

```
class MyClass
{
  public static int count;
  public MyClass()
  {
    count++;
  }
}

int i1 = MyClass.count; // i1 = 0;
MyClass c;
for (int i=0; i<5; i++)
  c = new MyClass();
int i2 = MyClass.count; // i2 = 5;
```

Фактически статический член-данные является глобальной переменной в рамках некоторого класса. Поэтому лучше работу с такими членами-данными класса осуществлять через статические методы класса (обратиться через обычные методы тоже возможно, но ведь для этого необходимо создать объект!) . Например, предыдущий пример можно записать следующим образом:

```
class MyClass
{
  private static int count;
  public MyClass()
  {
    count++;
  }
  public static int GetCount()
  {
    return count;
  }
}

int i1 = MyClass.GetCount(); // i1 = 0;
MyClass c;
for (int i=0; i<5; i++)
```

```
  c = new MyClass();
int i2 = MyClass.GetCount(); // i2 = 5;
```

Так как статические члены-методы вызываются через класс, то при описании их реализации есть некоторые ограничения:

- в теле метода должно отсутствовать ключевое слово `this`, т.к. метод выполняется безотносительно к какому-либо объекту;
- в теле метода допускается вызов только статических членов-методов класса (но возможен вызов нестатических методов через объект, переданный в метод в качестве параметра);
- в теле метода возможно обращение только к статическим членам-данным класса.

Статическим может быть и конструктор, который применяются для инициализации статических членов-данных. Однако при описании такого конструктора не указывается спецификатор доступа (т.е. он имеет доступ по умолчанию – `private` – и не может быть явно вызван). Например, класс, описанный в предыдущем примере, может быть расширен статическим конструктором (хотя с точки зрения программы он и излишен) следующим образом:

```
class MyClass
{
  private static int count;
  static MyClass()
  {
    count = 0;
  }
  public MyClass()
  {
    count++;
  }
  public static int GetCount()
  {
    return count;
  }
}
```

Статический конструктор не может иметь параметров, и, следовательно, не может быть перегружен.

Вызов статического конструктора выполняется автоматически при запуске программы, использующей класс.

Статическими могут быть не только отдельные члены класса, но и класс целиком. Это достигается применением ключевого слова `static` в описании самого класса. Статический класс обладает двумя основными свойствами:

- объекты статического класса создать нельзя;
- статический класс должен содержать только статические члены.

Одно из основное назначение статического класса – создание класса, объединяющего статические члены, назначение или физическая сущность которых схожи. Примером статического класса может служить класс `Math`.

4.2 Наследование

Наследование является одним из трёх основополагающих принципов объектно-ориентированного программирования, поскольку оно допускает создание иерархических классификаций.

Класс, который наследуется, называется базовым, а класс, который наследует, – производным. Следовательно, производный класс представляет собой специализированный вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы (**НО НЕ КОНСТРУКТОРЫ**), определяемые в базовом классе, добавляя к ним свои собственные элементы.

Если вновь создаваемый класс является производным от некоторого базового класса[1], то после идентификатора производного класса ставится двоеточие, после которого указывается идентификатор базового класса.

Пример: Создание класса «квадрат» на основе разработанного выше класса «фигура», дополнение его свойством «размер» (с допустимым значением >0) и методом расчёта площади.

```
class Square : Figure
{
  private double _size;
  public double size
  {
    get { return _size; }
    set { if (value > 0) _size = value; }
  }
  public double Area()
  {
    return _size*_size;
  }
}

Square square = new Square();
square.sType = "Квадрат";
square.posX = square.posY = 1;
square.size = 5.1;
double d = square.Area(); // d = 26.01
Figure figure = square.Duplicate();
```

При подробном рассмотрении приведённого выше кода (в том числе и анализе кода класса `Figure`) можно выделить несколько негативных моментов, требующих доработки как производного, так и базового классов:

1. т.к. наследование конструкторов не осуществляется, то при создании объекта используется конструктор по умолчанию, который присваивает «нулевое» значение полю `size`. Это нарушает правила работы класса, т.к. если не будут явно указаны все значения полей, то объект будет содержать поля с недопустимыми

[1] В отличии от C++, в C# базовый класс может быть только один, т.е. не допускается множественное наследование.

значениями. Поэтому для класса `Square` требуется описание, как минимум, конструктора без параметров;

2. несмотря на то, что у класса `Square` значение поля `_sType` может быть только «Квадрат» (т.к. класс описывает именно такую фигуру), при использовании класса возможно присвоение этому полю произвольного значения. Для устранения этого недочёта в базовом классе свойство `sType` должно быть «только для чтения»;
3. метод `Duplicate` создаст объект класса `Figure`, что будет неверным с точки зрения назначения этого метода. Поэтому метод должен быть переопределён;
4. т.к. в класса `Figure` поля `_sType`, `_posX` и `_posY` объявлены со спецификатором доступа `private`, то доступ к ним в производном класса возможен только через свойства, хотя поля и являются часть производного класса. Для устранения этого недостатка[1] с сохранением блокировки доступа из-за пределов производных классов, нужно сменить спецификатор доступа у этих полей в базовом классе на `protected`.

Изменённый в соответствии с пунктами 2 и 4 класс `Figure` будет выглядеть следующим образом:

```
class Figure
{
  protected string _sType;
  protected int _posX, _posY;
  public string sType
  {
    get { return _sType; }
  }[2]
  public int posX
  {
    get { return _posX; }
    set { if (value >= 0) _posX = value; }
  }
  public int posY
  {
    get { return _posY; }
    set { if (value >= 0) _posY = value; }
  }
  public Figure() : this("Не задан",1,1)
  {
  }
  public Figure(string n_sType) : this(n_sType,1,1)
  {
  }
  public Figure(string n_sType, int n_posX, int n_posY)
```

[1] В принципе, возможность доступа через свойства не является недостатком, т.к. обеспечивает защищенность данных, хотя и уменьшает быстродействие программы. В данном примере, изменение спецификатора доступа на `protected` используется для его демонстрации. Применение этого спецификатора полностью оправдано, когда имеется поле, которое не связано с каким-либо свойством для блокировки доступа к нему извне, но должно быть доступно в производных классах.

[2] Жирностью выделен измененный или добавленный код.

```
  {
    _sType = n_sType.Trim() != "" ? n_sType.Trim() : "Не задан";
    _posX = n_posX >= 0 ? n_posX : 1;
    _posY = n_posY >= 0 ? n_posY : 1;
  }
  public Figure(Figure sourceFigure)
  {
    _sType = sourceFigure._sType;
    _posX = sourceFigure._posX;
    _posY = sourceFigure._posY;
  }
  public Figure Duplicate()
  {
    return new Figure(this);
  }
}
```

Далее рассматриваются способы устранения недостатков пунктов 1 и 3.

4.2.1 Наследование и конструкторы

Для устранения недостатка 1 дополним класс `Square` конструктором без параметров, и конструктором копии:

```
class Square : Figure
{
  private double _size;
  public double size
  {
    get { return _size; }
    set { if (value > 0) _size = value; }
  }
  public double Area()
  {
    return _size*_size;
  }
  public Square()
  {
    _sType = "Квадрат";
    _size = 1;
  }
  public Square(Square sourceSquare)
  {
    _sType = sourceSquare._sType;
    _posX = sourceSquare._posX;
    _posY = sourceSquare._posY;
    _size = sourceSquare._size;
  }
}
```

Создание в производном классе конструкторов возможно, если выполняется одно из трёх условий:

- ***базовый класс не имеет конструкторов***. В этом случае выполняется конструктор по умолчанию для базового класса, а потом конструктор производного класса;
- ***базовый класс имеет конструкторы, и среди них есть конструктор без параметров***. В этом случае сначала выполняется конструктор без параметров базового класса, а потом конструктор производного класса;
- ***производный класс явно вызывает конструктор базового класса***. В этом случае сначала вызывается указанный конструктор базового класса, а потом конструктор производного класса.

Если базовый класс в свою очередь является производным от другого класса, то вызов его конструктора будет выполняться по этой же схеме, т.е. при многоуровневой иерархии всегда сначала вызываются конструкторы классов более высокого уровня, а уже потом более низкого.

Для создания конструктора, явно вызывающего конструктор базового класса, используется конструкция вида:

```
<доступ> <имя_класса>([<список параметров1>]) : base([<список
параметров2>])
{
  <тело конструктора>
}
```

Здесь `<список параметров1>` определяет набор формальных параметров конструктора производного класса, а `<список параметров2>` – набор фактических параметров вызываемого конструктора базового класса. При этом, в `<список параметров2>` могут использоваться параметры из `<список параметров1>`.

Пример: изменение конструктора без параметров у класса `Square`.

```
public Square() : base("Квадрат")
{
  _size = 1;
}
```

4.2.2 Переопределение членов класса

Для устранения недостатка 3 добавим в класс `Square` переопределённый вариант метода `Duplicate`:

```
class Square : Figure
{
  private double _size;
  public double size
  {
    get { return _size; }
    set { if (value > 0) _size = value; }
  }
```

```
  public double Area()
  {
    return _size*_size;
  }
  public Square() : base("Квадрат")
  {
    _size = 1;
  }
  public Square(Square sourceSquare)
  {
    _sType = sourceSquare._sType;
    _posX = sourceSquare._posX;
    _posY = sourceSquare._posY;
    _size = sourceSquare._size;
  }
  public Square Duplicate()
  {
    return new Square(this);
  }
}
```

При переопределении члена-метода с тем же набором параметров, что и у базового класса, возникает эффект сокрытия метода базового класса, о чем компилятор сообщит в виде предупреждения. Если такое переопределение делается сознательно, то для устранения предупреждения перед возвращаемым типом указывается ключевое слово `new`. Например, предыдущий вариант описания заголовка метода `Duplicate` может быть записан так:

```
new public Square Duplicate()
```

Если в производном классе при переопределении метода изменяется число параметров, то сокрытия ***не происходит***.

Аналогично, может осуществляться сокрытие членов-данных, например:

```
class Class1
{
  public int a;
}
class Class2 : Class1
{
  public new double a;
}
```

Если произошло сокрытие члена базового класса, то доступ к нему все же может быть осуществлён с использованием ключевого слова `base`, например:

```
class Class1
{
  public int a = 1;
  public void Mul()
  {
    a *= 2;
```

```
  }
}
class Class2 : Class1
{
  public new double a;
  public new void Mul()
  {
    a *= 3;
  }
  public void Calc()
  {
    base.Mul(); // Class1.a = 2
    a = base.a; // Class2.a = 2.0
    Mul();      // Class2.a = 6.0;
  }
}
```

4.3 Полиморфизм

Язык C# является строго типизированным языком. Поэтому, строго говоря, переменная ссылочного типа может ссылаться только на объект этого типа. Однако из этого правила есть одно важное исключение: ***переменной-ссылке на объект базового класса может быть присвоена ссылка на объект любого производного от него класса***. Такое присваивание считается вполне допустимым, поскольку экземпляр объекта производного класса наследует все члены базового класса. Однако, доступа к членам, имеющимся только в производном классе, в этом случае не будет.

Пример:

```
Figure figure = new Square();
string s = figure.sType; // s = "Квадрат"
// У базового класса нет свойства size, поэтому следующая строка
// не допустима
// double d = figure.size;
```

Возможность присвоения переменной-ссылке на базовый класс ссылки на производный класс может быть использована, например, при реализации конструкторов копии.

Пример: изменение конструктора копии класса `Square` с применением вызова конструктора копии базового класса.

```
public Square(Square sourceSquare) : base(sourceSquare)
{
  _size = sourceSquare._size;
}
```

Возможность переменной ссылки на базовый класс хранить объекты любого производного класса и правильно выполнять набор операций, определённых в базовом классе, и называется ***полиморфизмом***.

Именно из-за этой возможности часто создаются обобщённые базовые классы, на основе которых строятся производные классы с конкретными особенностями реализации.

Пример: перестроим иерархию классов, добавив в неё класс «прямоугольник» и разместив метод расчёта площади в базовом классе.

```
class Figure
{
  protected string _sType;
  protected int _posX, _posY;
  public string sType
  {
    get { return _sType; }
  }
  public int posX
  {
    get { return _posX; }
    set { if (value >= 0) _posX = value; }
  }
  public int posY
  {
    get { return _posY; }
    set { if (value >= 0) _posY = value; }
  }
  public Figure() : this("Не задан",1,1)
  {
  }
  public Figure(string n_sType) : this(n_sType,1,1)
  {
  }
  public Figure(string n_sType, int n_posX, int n_posY)
  {
    _sType = n_sType.Trim() != "" ? n_sType.Trim() : "Не задан";
    _posX = n_posX >= 0 ? n_posX : 1;
    _posY = n_posY >= 0 ? n_posY : 1;
  }
  public Figure(Figure sourceFigure)
  {
    _sType = sourceFigure._sType;
    _posX = sourceFigure._posX;
    _posY = sourceFigure._posY;
  }
  public Figure Duplicate()
  {
    return new Figure(this);
  }
  public double Area()
  {
    return 0; // Метод должен что-то возвращать
  }
}
```

```
class Square : Figure
{
  private double _size;
  public double size
  {
    get { return _size; }
    set { if (value > 0) _size = value; }

  }
  public Square() : base("Квадрат")
  {
    _size = 1;
  }
  public Square(Square sourceSquare) : base(sourceSquare)
  {
    _size = sourceSquare._size;
  }
  new public Square Duplicate()
  {
    return new Square(this);
  }
  new public double Area()
  {
    return _size*_size;
  }
}

class Rectangle : Figure
{
  private double _size1, _size2;
  public double size1
  {
    get { return _size1; }
    set { if (value > 0) _size1 = value; }
  }
  public double size2
  {
    get { return _size2; }
    set { if (value > 0) _size2 = value; }
  }
  public Rectangle() : base("Прямоугольник")
  {
    _size1 = _size2 = 2; // Просто для различия площади
  }
  public Rectangle(Rectangle sourceRectangle) : base(sourceRectangle)
  {
    _size1 = sourceRectangle._size1;
    _size2 = sourceRectangle._size2;
  }
  new public Rectangle Duplicate()
  {
    return new Rectangle(this);
  }
```

```
  new public double Area()
  {
    return _size1*_size2;
  }
}

Figure f = new Figure();
string s = String.Format("{0}. Площадь: {1:N2}", f.sType, f.Area());
// s = "Не задан. Площадь: 0,00"
f = new Square();
s = String.Format("{0}. Площадь: {1:N2}", f.sType, f.Area());
// s = "Квадрат. Площадь: 0,00"
f = new Rectangle();
s = String.Format("{0}. Площадь: {1:N2}", f.sType, f.Area());
// s = "Прямоугольник. Площадь: 0,00"
```

Результаты оказались неожиданными. Несмотря на то, что в каждой из строк правильно выведен тип фигуры, расчёт площади выполнен не верно. Это происходит из-за того, что в момент компиляции производится связь между данными объекта и методами класса, к которому принадлежит переменная – ***раннее связывание***.

Кроме того, если в программу добавить строку

```
Rectangle r = f.Duplicate();
```

то данная строка не будет скомпилирована, т.к. будет использоваться метод `Duplicate` класса `Figure` из-за того, что переменная `f` имеет данный тип, а присвоение переменной-ссылке на объект-родитель ссылки на объект-потомок недопустимо.

Для устранения недостатка по расчёту площади используются ***виртуальные методы***. Устранение недостатка по дублированию тоже возможно с использованием виртуальных методов, однако это потребует изменения метода в производных классах.

4.3.1 Виртуальные методы

Виртуальные методы используют технологию ***позднего связывания***, в которой определение того, метод какого класса должен быть вызван, производится ***во время работы программы*** не по типу переменной, а по объекту, ссылку на который она хранит.

Для того, чтобы осуществлялся правильный расчёт площади фигуры, сделаем метод `Area` виртуальным, путём использования модификатора `virtual` в базовом классе `Figure` и модификатора `override` в производных классах `Square` и `Rectangle`. Также требуется убрать ключевое слово `new` в методе `Area` классов `Square` и `Rectangle`.

```
class Figure
{
  ...
```

```
  public virtual double Area()
  {
    return 0;
  }
}
class Square : Figure
{
  ...
  public override double Area()
  {
    return _size*_size;
  }
}
class Rectangle : Figure
{
  ...
  public override double Area()
  {
    return _size1*_size2;
  }
}

Figure f = new Figure();
string s = String.Format("{0}. Площадь: {1:N2}", f.sType, f.Area());
// s = "Не задан. Площадь: 0,00"
f = new Square();
s = String.Format("{0}. Площадь: {1:N2}", f.sType, f.Area());
// s = "Квадрат. Площадь: 1,00"
f = new Rectangle();
s = String.Format("{0}. Площадь: {1:N2}", f.sType, f.Area());
// s = "Прямоугольник. Площадь: 4,00"
```

При переопределении виртуального метода в производном классе его название, возвращаемый тип, список параметров должны быть ***идентичны значениям в базовом классе***. Поэтому, разработка виртуальных методов требует тщательного продумывания их структуры.

Именно из-за необходимости поддерживать идентичность при переопределении виртуальных методов невозможно простое превращение метода `Duplicate` в виртуальный, т.к. в разных классах данный метод возвращает объекты разных классов. Решением проблемы могло бы стать изменение во всех классах возвращаемого типа на `Figure`, однако в этом случае при создании новых объектов путём дублирования пришлось бы постоянно выполнять преобразование типов.

В дальнейшем, из классов будет удалён метод `Duplicate`, т.к. наличия конструктора копии вполне достаточно для дублирования объекта.

Виртуальные методы не могут быть `static`, т.к. они подразумевают переопределение в классах-потомках, что недопустимо для метода класса.

4.3.2 Абстрактные классы и члены классов

Рассматривая разработанную выше иерархию классов можно отметить, что класс `Figure` фактически служит только для того, чтобы на основе его создать порождённые классы. Создание объекта такого класса не имеет смысла. Кроме того, в методе `Area` данного класса потребовалось возвращать какое-нибудь значение (возвращается 0), хотя рассчитывать площадь неопределённой фигуры бессмысленно.

Для того, чтобы не описывать в базовом классе реализацию методов, которые вводятся в класс для обеспечения единообразия списка методов у порождённых классов, данные методы в базовом классе делают ***абстрактными***. Абстрактный метод имеет следующий особенности:

- перед методом указывается модификатор `abstact`;
- метод является виртуальным, но модификатор `virtual` в базовом классе не указывается;
- абстрактный метод не может быть статическим (т.е. у него не может быть модификатора `static`);
- в порождённом классе требуется указание модификатора `override`;
- ***реализация*** абстрактного метода ***в базовом классе не требуется*** (да и не допустима);
- ***реализация*** абстрактного метода ***в порождённом классе обязательна***, если данный класс не является абстрактным.

Если в классе имеется хотя бы один абстрактный метод, то весь класс становится абстрактным и в строке заголовка класса должен быть указан модификатор `abstract`. Однако класс может быть объявлен абстрактным, даже если он не имеет ни одного абстрактного метода.

Объект абстрактного класса не может быть создан.

Абстрактными также могут быть свойства и индексаторы.

Пример: модифицируем иерархию классов, сделав класс `Figure` и метод `Area` этого класса абстрактными.

```
abstract class Figure
{
  ...
  public abstract double Area();
}

class Square : Figure
{
  ...
}
class Rectangle : Figure
{
  ...
}

Figure f;
string s;
```

```
// f = new Figure(); Теперь данная строка недопустима
f = new Square();
s = String.Format("{0}. Площадь: {1:N2}", f.sType, f.Area());
// s = "Квадрат. Площадь: 1,00"
f = new Rectangle();
s = String.Format("{0}. Площадь: {1:N2}", f.sType, f.Area());
// s = "Прямоугольник. Площадь: 4,00"
```

4.3.3 Операторы as и is

В предыдущем примере использовались возможности полиморфизма, позволяющие присвоить переменной класса `Figure` объекты порождённых классов:

```
Figure f = new Square();
```

Хотя мы и знаем, что в переменной `f` хранится ссылка на объект класса `Square`, обратиться к свойству `size` этого объекта без дополнительных преобразований будет невозможно, т.к. базовый класс не имеет такого свойства. Одним из возможных способов преобразования является приведение типов, которое может быть выполнено следующим образом:

```
((Square)f).size = 5;
```[1]

Однако явное преобразование при работе со ссылочными типами использовать не рекомендуется. Целесообразнее использовать специальный оператор преобразований ссылочных типов **`as`**, формальная схема которого имеет вид:

```
<объект> as <требуемый тип>
```

При помощи данного оператора присвоение вышеприведённый пример работы со свойством `size` может быть записан следующим образом:

```
(f as Square).size = 5;
```

Если преобразование выполнить невозможно, то возвращается `null`, например:

```
public class Class1 {}
public class Class2 : Class1 {}
public class Class3 : Class1 {}

Class2 cl2 = new Class2();
Class3 cl3 = new Class3();
Class1 cl1;
Class2 cl;
cl1 = cl2;
cl = cl1 as Class2;
```

[1] Строка `(Square)f.size = 5;` будет неправильной, т.к. операция обращения к члену класса выполняется ранее операции приведения типа

```
if (cl != null) // Истина
  textBox1.Text = cl.ToString();// Выполняется эта строка
else
  textBox1.Text = "null";
cl1 = cl3;
cl = cl1 as Class2;
if (cl != null) // Ложь
  textBox2.Text = cl.ToString();
else
  textBox2.Text = "null"; // Выполняется эта строка
```

Ещё одним оператором, используемым при работе с ссылочными типами, является оператор **`is`**, используемый для проверки принадлежности объекта к заданному типу[1]. Формальная схема данного оператора имеет вид:

```
<объект> is <проверяемый тип>
```

Оператор возвращает `true`, если объект может быть преобразован к проверяемому типу и `false` в противном случае (в том числе, если переменная имеет значение `null`). ***При проверке рассматривается внутреннее устройство объекта, а не тип переменной, которая проверяется***.

Пример: имеется массив, каждый элемент которого может хранить ссылку на объект класса, порождённого от класса `Figure`. Требуется присвоить свойству `size` каждого объекта класса `Square` значение 3, а свойствам `size1` и `size2` каждого объекта класса `Rectangle` значения 4 и 5 соответственно.

```
Figure[] mas = new Figure[???];
for (int i=0; i<mas.Length; i++)
  mas[i] = ???;
for (int i=0; i<mas.Length; i++)
{
  if (mas[i] is Square)
    (mas[i] as Square).size = 3;
  else
    if (mas[i] is Rectangle)
    {
      (mas[i] as Rectangle).size1 = 4;
      (mas[i] as Rectangle).size2 = 5;
    }
}
```

Как видно из примера, использование возможностей полиморфизма, а также операторов `as` и `is` позволяет производить обработку объектов разных (но имеющих единого родителя) классов единообразно.

[1] Правильнее сказать, что оператор `is` проверяет возможность преобразования. Для точной проверки сравнивается результат запроса типа объекта с помощью метода `Object.GetType()` с результатом получения типа с помощью оператора `typeof(<тип>)`. Если типы совпадают, то оба результата должны ссылаться на один и тот же объект типа `System.Type`.

4.3.4 Модификатор sealed

Несмотря на эффективность и полезность наследования, оно иногда бывает не желательным. Если на основе некоторого класса не может быть создано порождённых классов, то при описании класса используется модификатор **`sealed`**, например:

```
class MyClass1
{
  public int a;
}

sealed class MyClass2 : MyClass1 // Такое наследование допустимо
{
  public double b;
}

class MyClass3 : MyClass2 // А это уже не допустимо
{
  public int c;
}
```

Модификатор `sealed` ***не может быть применён для абстрактного класса***, т.к. абстрактный класс подразумевает наследование от него.

Модификатор `sealed` может также применяться при описании перегруженных виртуальных методов (т.е. описанных с модификатором `override`) для предотвращения дальнейшей их перегрузки, например:

```
class MyClass1
{
  public virtual void Method1() { ... }
  public virtual void Method2() { ... }
}

class MyClass2 : MyClass1
{
  public sealed override void Method1() { ... }
  public override void Method2() { ... }
}

class MyClass3 : MyClass2
{
  public override void Method1() { ... } // Недопустимо
  public override void Method2() { ... }
}
```

4.4 Перегрузка операторов

В языке C# перегруженными могут быть не только методы, но и операторы. Перегрузка операторов позволяет применять обычные операторы для операций над

созданными классами. Перегруженными могут быть как унарные, так и бинарные операторы. Формальная запись перегрузки оператора имеет вид:

- для унарных:

```
public static <возвращаемый тип> operator <оператор>
(<тип операнда> <операнд>) {<операции>}
```

- для бинарных:

```
public static <возвращаемый тип> operator <оператор>
(<тип операнда1> <операнд1>,<тип операнда2> <операнд2>) { <операции> }
```

где:

- `<возвращаемый тип>` – тип результата выполненной операции. Чаше всего совпадает с типом класса, для которого перегружается оператор;
- `<оператор>` – перегружаемый оператор;
- `<тип операнда>` – тип одного из операндов. Для унарного оператора должен совпадать с типом класса. Для бинарного оператора должен совпадать с типом класса хотя бы у одного операнда;
- `<операнд>` – идентификатор одного из операндов;
- `<операции>` – действия, выполняемые для получения требуемого результата.

При перегрузке операторов следует учитывать следующие особенности:

- не все операторы могут быть перегружены в том или ином классе (зависит от назначения и логики класса);
- ***всегда*** в методе, описывающем перегрузку оператора, ***создаётся новый объект*** (а не модифицируется существующий);
- в большинстве случаев требуется наличие в классе конструктора по умолчанию (без параметров);
- модификаторы `ref` и `out` не допустимы при описании параметров перегружаемого оператора.

Пример: жидкость характеризуется объёмом и плотностью. Создать класс, описывающий жидкость, и обеспечить перегрузку некоторых операций сложения, вычитания, сравнения.

```
public class Liquid
{
  double Ro, V;
  public Liquid(double NewRo, double NewV)
  {
    Ro = NewRo;
    V = NewV;
  }
  public Liquid() : this(0, 0) { }
  // Сложение двух жидкостей
  public static Liquid operator +(Liquid L1, Liquid L2)
  {
    Liquid result = new Liquid();
    double M = L1.Ro * L1.V + L2.Ro * L2.V; // Расчет массы
    result.V = L1.V + L2.V;                 // Расчет объёма
```

```
    result.Ro = M / result.V;                // Расчет плотности
    return result;
  }
  // Сложение жидкости и числа (увеличение объема)
  public static Liquid operator +(Liquid L, double V)
  {
    Liquid result = new Liquid();
    result.Ro = L.Ro;
    result.V = L.V + V;
    return result;
  }
  // Сложение числа и жидкости (увеличение объема)
  public static Liquid operator +(double V, Liquid L)
  {
    return L + V;
  }
  // Вычитание числа из жидкости (уменьшение объема)
  public static Liquid operator -(Liquid L, double V)
  {
    Liquid result = new Liquid();
    result.Ro = L.Ro;
    // Контроль за неотрицательностью объема
    result.V = L.V > V ? L.V - V : 0;
    return result;
  }
  // Увеличение объема на 1
  public static Liquid operator ++(Liquid L)
  {
    return L + 1;
  }
  // Уменьшение объема на 1
  public static Liquid operator --(Liquid L)
  {
    return L - 1;
  }
  // Проверка на равенство
  public static bool operator ==(Liquid L1, Liquid L2)
  {
    return (L1.Ro == L2.Ro) && (L1.V == L2.V);
  }
  // Проверка на неравенство
  public static bool operator !=(Liquid L1, Liquid L2)
  {
    return !(L1 == L2);
  }
  // Перегрузка методы выдачи строкового представления класса
  public override string ToString()
  {
    return String.Format("Ro: {0:F2}; V: {1:F2}", Ro, V);
  }
}
```

```
Liquid L1 = new Liquid(100, 2);
Liquid L2 = new Liquid(200, 3);
Liquid L3;
bool b;                          // Ниже приведены значения L3 и b
L3 = L1 + L2;                    // Ro: 160,00; V: 5,00
L3 = L1 + 2;                     // Ro: 100,00; V: 4,00
L3 = 3 + L1;                     // Ro: 100,00; V: 5,00
L3 = L1 - 1.5;                   // Ro: 100,00; V: 0,50
L3 = L1 - 2.5;                   // Ro: 100,00; V: 0,00
L3++;                            // Ro: 100,00; V: 1,00
L3 = ++L3 - 0.5;                 // Ro: 100,00; V: 1,50
--L3;                            // Ro: 100,00; V: 0,50
L3--;                            // Ro: 100,00; V: 0,00
L3 = L1;
b = L1 == L3;                    // b = True
b = L1 != L3;                    // b = False
```

В приведённом выше примере также можно было бы определить оператор сравнения жидкости с дробным числом (например, через объем).

Операторы == и !=, < и >, <= и >= ***должны перегружаться попарно***.

При перегрузке операторов == и != также требуется переопределить методы `Object.Equals` и `Object.GetHashCode`[1], например следующим образом:

```
public override bool Equals(object obj)
{
  if (obj != null && GetType() == obj.GetType())
    return this == (obj as Liquid);
  else
    return false;
}
public override int GetHashCode()
{
  return V.GetHashCode() ^ Ro.GetHashCode();
}
```

Также могут быть перегружены и другие операторы. Однако ряд операторов перегрузить нельзя: все операторы присваивания, `&&`, `||`, `()`, `[]`, `.`, `?`, `??`, `=>`, `->`, `checked`, `unchecked`, `default`, `as`, `is`, `new`, `sizeof`, `typeof`.

[1] Функция `GetHashCode()` должна для двух равных объектов возвращать одно и то же значение, но не обязана для двух различных объектов возвращать разные значения

5 Интерфейсы

Как отмечалось выше, для абстрактного метода указывается только описание его заголовка, но не указывается реализация. Таким образом, абстрактный метод представляет собой *интерфейс* работы с методом.

Абстрактный класс может сочетать в себе как абстрактные, так и обычные методы, свойства, индексаторы, а также хранить поля. Тем не менее, он может содержать и ***только абстрактные методы, свойства, индексаторы***.

Класс, содержащий ***только абстрактные методы, свойства, индексаторы***, может быть реализован не только как абстрактный класс, но и как ***интерфейс***. Формальная схема описания интерфейса имеет вид:

```
interface <идентификатор интерфейса> [:
  <идентификатор интерфейса-родителя1>[,
  <идентификатор интерфейса-родителя2>...]]
{
  <возвращаемый тип> <идентификатор метода1>([<список параметров>]);
  [<возвращаемый тип> <идентификатор метода2>([<список параметров>]);
  ...]
  <тип> <идентификатор свойства1> {get; set;}
  [<тип> <идентификатор свойства2> {get; set;}
  ...]
  <тип> this[<список параметров 1>] {get; set;}
  [<тип> this[<список параметров 2>] {get; set;}
  ...]
}
```

При описании членов интерфейса, ***доступ не указывается***, т.к. по умолчанию доступ всех членов интерфейса `public`.

Реализация членов интерфейса должна быть выполнена в классе, реализующем интерфейс. При этом, при реализации членов интерфейса они должны быть объявлены со спецификатором доступа **`public`**[1]. Для указания того, что класс реализует интерфейс используется конструкция:

```
[<доступ>] class <идентификатор класса> : <идентификатор интерфейса>
{
  <члены класса и реализация интерфейса>
}
```

Пример: создание и использование интерфейса с методом, свойством и индексатором. Код реализации выбран произвольно.

```
interface MyInterface
{
  int Method();    // Объявление метода
  int Property     // Объявление свойства
```

[1] Спецификатор доступа `public` не указывается, если явно указывается имя интерфейса. Явное указание имени интерфейса необходимо для сокрытия членов интерфейса в классе или для реализации одноименных членов нескольких интерфейсов в одном классе. Примеры такой реализации будут рассмотрены ниже

```
  {
    get;
    set;
  }
  int this[int n] // Объявление индексатора
  {
    get;
    set;
  }
}

class MyClass : MyInterface
{
  private int x;
  private int[] mas = new int[10];
  public int Method()    // Реализация метода
  {
    return 1;
  }
  public int Property    // Реализация свойства
  {
    get { return x; }
    set { x = value; }
  }
  public int this[int n] // Реализация индексатора
  {
    get {return (n >= 0 && n < mas.Length ? mas[n] : 0); }
    set {if (n >= 0 && n < mas.Length) mas[n] = value; }
  }
}

MyClass cl = new MyClass();
int i1 = cl.Method(); // i1 = 1
cl.Property = 5;
int i2 = cl.Property; // i2 = 5;
cl[1] = 4;
int i3 = cl[0];       // i3 = 0;
int i4 = cl[1];       // i4 = 4;
```

Для сокрытия членов интерфейса при их реализации явно указывается имя интерфейса и ***не указывается спецификатор доступа***, например:

```
interface MyInterface
{
  int A();
}

class MyClass : MyInterface
{
  int MyInterface.A()
  {
    return 1;
  }
```

```
  public int ClassA()
  {
    return ((MyInterface)this).A();
  }
}

MyClass cl = new MyClass();
int i;
i = cl.A();      // Нет доступа к методу интерфейса
i = cl.ClassA(); // i = 1;
```

Класс может реализовывать несколько интерфейсов. В этом случае они перечисляются после двоеточия через запятую. Если класс также является производным от некоторого базового класса, то после двоеточия ***сначала указывается базовый класс***, а после него через запятую реализуемые интерфейсы.

Таким образом, использование интерфейсов фактически обеспечивает ***множественное наследование, пусть и в усечённой форме***.

Если класс реализует несколько интерфейсов, имеющих одинакового члена (совпадает его описание), то, ***по умолчанию***, реализация в классе выполняется один раз и считается, что она относится к обоим интерфейсам, например:

```
interface MyInterface1
{
  int Method();
}

interface MyInterface2
{
  int Method();
}

class MyClass : MyInterface1, MyInterface2
{
  public int Method() // Реализация метода обоих интерфейсов
  {
    return 1;
  }
}

MyClass cl = new MyClass();
int i = cl.Method(); // i = 1
```

Однако если требуется, чтобы реализация у интерфейсов была разной, необходимо описать каждую из них отдельно с указанием перед членом идентификатора интерфейса с точкой. При этом, ***доступ к члену не указывается***. Обращение к такому члену выполняется через преобразование объекта к требуемому интерфейсу, например:

```
class MyClass : MyInterface1, MyInterface2
{
  int MyInterface1.Method() // Реализация метода MyInterface1
```

```
  {
    return 1;
  }
  int MyInterface2.Method() // Реализация метода MyInterface2
  {
    return 2;
  }
}

MyClass cl = new MyClass();
int i1 = ((MyInterface1)cl).Method(); // i1 = 1
int i2 = ((MyInterface2)cl).Method(); // i2 = 2
```

Явное указание интерфейса при реализации требуется и в том случае, если в двух интерфейсах используется свойство и метод с одинаковым именем, например:

```
interface MyInterface1
{
  int A();
}

interface MyInterface2
{
  int A { get; }
}

class MyClass : MyInterface1, MyInterface2
{
  public int A()
  {
    return 1;
  }
  int MyInterface2.A
  {
    get { return 2; }
  }
}

MyClass cl = new MyClass();
int i1 = cl.A();                // i1 = 1
int i2 = ((MyInterface2)cl).A; // i2 = 2
```

Пример: создание интерфейсов для двумерных (метод расчёта периметра) и трёхмерных фигур (метод расчёта объёма), добавление к классам `Square` и `Rectangle` интерфейса двумерной фигуры и создание класса «Куб» с интерфейсом трёхмерной фигуры и дополнительной координатой по оси `Z`.

```
interface I2D
{
  double Perimeter();
}
```

```
interface I3D
{
  double Capacity();
}

abstract class Figure
{
  protected string _sType;
  protected int _posX, _posY;
  public string sType
  {
    get { return _sType; }
  }
  public int posX
  {
    get { return _posX; }
    set { if (value >= 0) _posX = value; }
  }
  public int posY
  {
    get { return _posY; }
    set { if (value >= 0) _posY = value; }
  }
  public Figure() : this("Не задан",1,1)
  {
  }
  public Figure(string n_sType) : this(n_sType,1,1)
  {
  }
  public Figure(string n_sType, int n_posX, int n_posY)
  {
    _sType = n_sType.Trim() != "" ? n_sType.Trim() : "Не задан";
    _posX = n_posX >= 0 ? n_posX : 1;
    _posY = n_posY >= 0 ? n_posY : 1;
  }
  public Figure(Figure sourceFigure)
  {
    _sType = sourceFigure._sType;
    _posX = sourceFigure._posX;
    _posY = sourceFigure._posY;
  }
  public abstract double Area();
}

class Square : Figure, I2D
{
  private double _size;
  public double size
  {
    get { return _size; }
    set { if (value > 0) _size = value; }
  }
  public Square() : base("Квадрат")
```

```
  {
    _size = 1;
  }
  public Square(Square sourceSquare) : base(sourceSquare)
  {
    _size = sourceSquare._size;
  }
  public override double Area()
  {
    return _size*_size;
  }
  public double Perimeter()
  {
    return _size*4;
  }
}

class Rectangle : Figure, I2D
{
  private double _size1, _size2;
  public double size1
  {
    get { return _size1; }
    set { if (value > 0) _size1 = value; }
  }
  public double size2
  {
    get { return _size2; }
    set { if (value > 0) _size2 = value; }
  }
  public Rectangle() : base("Прямоугольник")
  {
    _size1 = _size2 = 2;
  }
  public Rectangle(Rectangle sourceRectangle) : base(sourceRectangle)
  {
    _size1 = sourceRectangle._size1;
    _size2 = sourceRectangle._size2;
  }
  public override double Area()
  {
    return _size1*_size2;
  }
  public double Perimeter()
  {
    return (_size1+_size2)*2;
  }
}

class Cube : Figure, I3D
{
  private int _posZ;
  private double _size;
```

```
  public int posZ
  {
    get { return _posZ; }
    set { if (value >= 0) _posZ = value;}
  }
  public double size
  {
    get { return _size; }
    set { if (value > 0) _size = value; }
  }
  public Cube() : base("Куб")
  {
    _posZ = 1;
    _size = 3; // Просто для разницы начальных значений классов
  }
  public Cube(Cube sourceCube) : base(sourceCube)
  {
    _posZ = sourceCube._posZ;
    _size = sourceCube._size;
  }
  public override double Area()
  {
    return _size*_size*6;
  }
  public double Capacity()
  {
    return _size*_size*_size;
  }
}

Figure[] mas = new Figure[???];
string[] s = new string[mas.Length];
for (int i=0; i<mas.Length; i++)
  mas[i] = ???;
for (int i=0; i<mas.Length; i++)
  if (mas[i] is I2D)
    s[i] = String.Format("{0}. Площадь: {1:N2}. Периметр: {2:N2}.",
           mas[i].sType,mas[i].Area(),(mas[i] as I2D).Perimeter());
  else
    if (mas[i] is I3D)
      s[i] = String.Format("{0}. Площадь: {1:N2}. Объем: {2:N2}.",
             mas[i].sType,mas[i].Area(),(mas[i] as I3D).Capacity());
    else
      s[i] = String.Format("{0}. Площадь: {1:N2}.",
             mas[i].sType,mas[i].Area());

// Например:
// s[0] = "Квадрат. Площадь: 1,00. Периметр: 4,00."
// s[1] = "Прямоугольник. Площадь: 4,00. Периметр: 8,00."
// s[2] = "Куб. Площадь: 54,00. Объем: 27,00."
// s[3] = "XXXXXXXXXXXX. Площадь: X,XX."
// ...
```

6 Делегаты, лямбда-выражения и события

6.1 Делегаты

Делегаты представляют собой объекты, которые могут ссылаться на метод и вызывать его. При этом, имеется возможность менять метод, на который ссылается делегат, т.е. использовать один и тот же объект-делегат для вызова разных методов.

Формальная схема описания делегата имеет вид:

```
delegate <возвращаемый тип> <имя делегата>([<список параметров>]);
```

Объявление делегата и присвоение ему метода осуществляется по схеме

```
<тип делегата> <идентификатор> = new <тип делегата>(<метод>);
```

или по упрощённой схеме (***групповое преобразование методов***)

```
<тип делегата> <идентификатор> = <метод>;
```

Пример: создать делегат, обрабатывающий целое число определённым образом. Создать класс, имеющий методы добавления и умножения исходного числа на два. Произвести обработку числа.

```
delegate int IntProcessing(int res);

public static class MyClass
{
  public static int Add2(int res)
  {
    return res + 2;
  }
  public static int Mul2(int res)
  {
    return res * 2;
  }
}

int i = 5;
IntProcessing IntProc = MyClass.Add2;
i = IntProc(i); // i = 7
IntProc = MyClass.Mul2;
i = IntProc(i); // i = 14
```

Применение упрощённой схемы позволяет группировать методы, т.е. при вызове делегата выполнять несколько методов одновременно. Это достигается операциями сложения и вычитания методов. При этом, операции выполняются в порядке их добавления. Например, предыдущий пример (его основная программа) может быть реализован следующим образом:

```
int i = 5;
IntProcessing IntProc = MyClass.Add2;
IntProc += MyClass.Mul2;
i = IntProc(i); // i = 10
```

Результат получился не тем, что ожидалось, т.к. каждому из методов было передано одно и то же начальное значение переменной `i` и, фактически, это результат работы последнего метода. Для получения нужного результата изменим делегат и методы следующим образом:

```
delegate void IntProcessing(ref int res);

public static class MyClass
{
  public static void Add2(ref int res)
  {
    res += 2;
  }
  public static void Mul2(ref int res)
  {
    res *= 2;
  }
}

int i = 5;
IntProcessing IntProc = MyClass.Add2;
int i1, i2, i3, i4;
i1 = i2 = i3 = i4 = i;
IntProc(ref i1);               // i1 = 7  {5 + 2}
IntProc += MyClass.Mul2;
IntProc(ref i2);               // i2 = 14 {(5 + 2) * 2}
IntProc -= MyClass.Add2;
IntProc(ref i3);               // i3 = 10 {5 * 2}
IntProc += MyClass.Add2;
IntProc(ref i4);               // i4 = 12 {(5 * 2) + 2}
```

Попытка удалить из делегата метод, который ему не присвоен, не вызовет никаких действий.

Делегаты становятся ещё более гибкими средствами программирования благодаря двум свойствам: ***ковариантности*** и ***контравариантности***. Как правило, метод, передаваемый делегату, должен иметь такой же возвращаемый тип и сигнатуру, как и делегат. Но в отношении производных типов это правило оказывается не таким строгим благодаря ковариантности и контравариантности.

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

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

Такие присвоения возможны из-за того, что классу-родителю всегда можно присвоить класс-потомок.

Следующий пример иллюстрирует применение принципов ковариантности и контравариантности.

```
class MyClass1 { public int A, B; }
class MyClass2 : MyClass1 { public int C, D; }

static class Change
{
  public static MyClass1 Change1(MyClass1 Arg)
  {
    MyClass1 Temp = new MyClass1();
    Temp.A = Arg.B;
    Temp.B = Arg.A;
    return Temp;
  }
  public static MyClass2 Change2(MyClass2 Arg)
  {
    MyClass2 Temp = new MyClass2();
    Temp.A = Arg.A;
    Temp.B = Arg.B;
    Temp.C = Arg.D;
    Temp.D = Arg.C;
    return Temp;
  }
  public static string GetS(MyClass1 temp)
  {
    if (temp is MyClass2)
    {
      MyClass2 cl2 = temp as MyClass2;
      return "MyClass2, A=" + cl2.A + ", B=" + cl2.B +
        ", C=" + cl2.C + ", D=" + cl2.D;
    }
    else
      return "MyClass1, A=" + temp.A + ", B=" + temp.B;
  }
}

delegate MyClass1 RunChange(MyClass2 Ard);

MyClass2 cl = new MyClass2() { A = 1, B = 2, C = 3, D = 4 };
RunChange RC = Change.Change1; // контравариантность
MyClass1 temp = RC(cl);
string s = Change.GetS(temp);  // s = "MyClass1, A=2, B=1"
RC = Change.Change2;           // ковариантность
temp = RC(cl);
s = Change.GetS(temp);         // s = "MyClass2, A=1, B=2, C=4, D=3"
```

6.2 Анонимные методы и лямбда-выражения

Метод, на который ссылается делегат, нередко используется только для этой цели. Иными словами, единственным основанием для существования метода служит то обстоятельство, что он может быть вызван посредством делегата, но сам он не вызывается вообще. В подобных случаях можно воспользоваться ***анонимной функцией***, чтобы не создавать отдельный метод. Анонимная функция, по существу, представляет собой безымянный кодовый блок, передаваемый конструктору делегата.

Начиная с версии 3.0, в C# предусмотрены две разновидности анонимных функций: анонимные методы и лямбда-выражения.

Для создания ***анонимного метода*** достаточно указать кодовый блок после ключевого слова `delegate` (или скобок с параметрами, если параметры имеются). Кодовый блок указывается в фигурных скобках, ***после которых ставится точка с запятой***. Например:

```
delegate int CalcSum(int start, int end);

CalcSum Sum = delegate(int start, int end)
{
  int sum = 0;
  for (int i = start; i <= end; i++)
    sum += i;
  return sum;
};
int i1 = Sum(5, 8); // i1 = 26
int i2 = Sum(1, 3); // i2 = 6
```

В ***лямбда-выражениях*** применяется новый лямбда-оператор =>, который разделяет лямбда-выражение на две части. В левой его части указывается входной параметр (или несколько параметров), а в правой части – тело лямбда-выражения.

Если тело лямбда-выражения состоит из одного оператора, то формальная запись лямбда-выражения имеет вид:

```
(<список параметров>) => <оператор>;
```

при этом результатом лямбда-выражения будет результат выполнения оператора.

Формальная запись `<списка параметров>` имеет вид:

```
[<тип параметра 1>] <параметр 1>[,[<тип параметра 2>]<параметр2>,...]
```

В большинстве случаев типы параметров не указываются. Кроме того, если имеется только один параметр, то его не обязательно заключать в круглые скобки.

Например, используем лямбда-выражение для создания делегата, умножающего целое число на два.

```
delegate int Mul2(int value);
```

```
Mul2 Func = value => value * 2;
int i = Func(5); // i = 10;
```

Если тело лямбда-выражения состоит из нескольких операторов, то оно заключается в фигурные скобки, ***после которых ставится точка с запятой***. Например, приведённый ранее пример с анонимными методами может быть записан с использованием лямбда-выражений следующим образом:

```
delegate int CalcSum(int start, int end);

CalcSum Sum = (start, end) =>
{
  int sum = 0;
  for (int i = start; i <= end; i++)
    sum += i;
  return sum;
};
int i1 = Sum(5, 8); // i1 = 26;
int i2 = Sum(1, 3); // i2 = 6
```

Как видно из примера, в теле блочного лямбда-выражения требуется использовать оператор `return` для указания результата работы.

6.3 События

Событие, по существу, представляет собой автоматическое уведомление о том, что произошло некоторое действие. События действуют по следующему принципу: объект, проявляющий интерес к событию, регистрирует обработчик этого события. Когда же событие происходит, вызываются все зарегистрированные обработчики этого события.

Обработчики событий обычно представлены делегатами, для чего используется следующая формальная запись:

```
event <имя делегата> <имя события>;
```

Кроме того, обычно в классе имеется метод, вызываемый при необходимости активизации события (обычно название метода начинается с `On`).

Пример: создание события и вызов его для различных экземпляров класса.

```
delegate void MyEventHandler();         // Делегат для события

class RunEvent                          // Класс события
{
  public event MyEventHandler MyEvent; // Само событие
  public void OnMyEvent()               // Метод активизации события
  {
    if (MyEvent != null)
      MyEvent();
  }
```

```
}
class MyClass          // Класс, заинтересованный в событии
{
  public int i = 5;
  public void Add2()   // Метод, выполняемый при возникновении события
  {
    i += 2;
  }
}

RunEvent clRun = new RunEvent();
MyClass cl1 = new MyClass();
MyClass cl2 = new MyClass();
clRun.OnMyEvent();                    // cl1: 5, cl2: 5
clRun.MyEvent += cl1.Add2;
clRun.OnMyEvent();                    // cl1: 7, cl2: 5
clRun.MyEvent += cl2.Add2;
clRun.OnMyEvent();                    // cl1: 9, cl2: 7
clRun.MyEvent -= cl1.Add2;
clRun.OnMyEvent();                    // cl1: 9, cl2: 9
```

События получаются объектами в порядке регистрации интереса к ним. Добавление обработчиков событий в цепочку обработчиков событий, а также удаление из неё выполняется, по умолчанию, автоматически. Однако, при необходимости, можно реализовать действия, выполняемые при добавлении или удалении обработчика событий. Формальная запись реализации такого события имеет вид:

```
event <имя делегата> <имя события>
{
  add { ... }
  remove { ... }
}
```

В блоках `add` и `remove` полученный обработчик события доступен через автоматически созданное свойство `value`.

При реализации события с обработкой добавления и удаления становится невозможным использовать его нигде, кроме левой части операторов += и −=. Поэтому запоминание всех обработчиков и работа с ними должна осуществляться вручную.

Модификация предыдущего примера с пользовательской обработкой добавления и удаления обработчиков и подсчётом количества обработчиков будет иметь вид:

```
delegate void MyEventHandler();

class RunEvent
{
  public int Count = 0;                       // Количество обработчиков
  private MyEventHandler ListHandler = null;// Цепочка обработчиков
  public event MyEventHandler MyEvent
  {
    add
```

```
    {
      Count++;
      ListHandler += value;
    }
    remove
    {
      Count--;
      ListHandler -= value;
    }
  }
  public void OnMyEvent()
  {
    if (ListHandler != null) // Теперь здесь MyEvent
      ListHandler();         // использовать нельзя
  }
}
class MyClass
{
  public int i = 5;
  public void Add2()
  {
    i += 2;
  }
}

RunEvent clRun = new RunEvent();
MyClass cl1 = new MyClass();
MyClass cl2 = new MyClass();
clRun.OnMyEvent();              // Count: 0, cl1: 5, cl2: 5
clRun.MyEvent += cl1.Add2;
clRun.OnMyEvent();              // Count: 1, cl1: 7, cl2: 5
clRun.MyEvent += cl2.Add2;
clRun.OnMyEvent();              // Count: 2, cl1: 9, cl2: 7
clRun.MyEvent -= cl1.Add2;
clRun.OnMyEvent();              // Count: 1, cl1: 9, cl2: 9
```

В качестве обработчика события может также использоваться анонимный метод или лямбда-выражение.

В среде .NET Framework обработчик события (делегат) имеет вид:

```
delegate void <имя делегата>(object sender, EventArgs e);
```

где `sender` – отправитель события, а `e` – дополнительная информация о событии.

Модифицируем предыдущий пример таким образом, чтобы он был совместим с .NET Framework, объект класса события подсчитывал, сколько раз он активизировал событие, и передавал это значение в качестве параметра. Переданное значение будет добавляться вместо двойки в методе `Add2`.

```
delegate void MyEventHandler(object sender, EventArgs e);

class MyEventArgs : EventArgs { public int num; }
class RunEvent
```

```
{
  public int Count = 0;
  public int CountEvt = 0;  // Количество активаций события у объекта
  private MyEventHandler ListHandler = null;
  public event MyEventHandler MyEvent
  {
    add
    {
      Count++;
      ListHandler += value;
    }
    remove
    {
      Count--;
      ListHandler -= value;
    }
  }
  public void OnMyEvent()
  {
    CountEvt++;
    if (ListHandler != null)
    {
      MyEventArgs e = new MyEventArgs();
      e.num = CountEvt;
      ListHandler(this, e);
    }
  }
}
class MyClass
{
  public int i = 5;
  public void Add2(object sender, EventArgs e)
  {
    i += (e as MyEventArgs).num;
  }
}

RunEvent clRun = new RunEvent();
MyClass cl1 = new MyClass();
MyClass cl2 = new MyClass();
clRun.OnMyEvent();           // Count: 0, CountEvt:1, cl1: 5, cl2: 5
clRun.MyEvent += cl1.Add2;
clRun.OnMyEvent();           // Count: 1, CountEvt:2, cl1: 7, cl2: 5
clRun.MyEvent += cl2.Add2;
clRun.OnMyEvent();           // Count: 2, CountEvt:3, cl1: 10, cl2: 8
clRun.MyEvent -= cl1.Add2;
clRun.OnMyEvent();           // Count: 1, CountEvt:4, cl1: 10, cl2: 12
```

Также в среде .NET Framework имеется делегат для обработки событий `EventHandler<TEventArgs>`[1], который позволяет не создавать собственных делегатов для обработки событий. Кроме того, если событию не требуются дополни-

[1] Формат и смысл такой записи будет рассмотрен позже в разделе универсальных типов

тельные параметры, то оно может быть описано с использованием делегата `EventHandler`. С применением этих делегатов событие могло бы быть описано следующим образом:

```
public event EventHandler<MyEventArgs> MyEvent { ... }
```

или

```
public event EventHandler MyEvent { ... }
```

7 Универсальные типы[1]

Как известно, многие алгоритмы очень похожи по своей логике независимо от типа данных, к которым они применяются. Например, механизм, поддерживающий очередь, остаётся одинаковым независимо от того, предназначена ли очередь для хранения элементов типа `int`, `string`, `object` или для класса, определяемого пользователем. До появления универсальных типов для обработки данных разных типов приходилось создавать различные варианты одного и того же алгоритма. А благодаря универсальным типам можно сначала выработать единое решение независимо от конкретного типа данных, а затем применить его к обработке данных самых разных типов без каких-либо дополнительных усилий.

При описании универсального типа указывается ***параметр типа*** (или параметры, если их несколько), который далее используется во всех членах данного типа (чаще всего называется `T`). Фактически, параметр типа является ***местозаполнителем*** для фактического типа, указываемого при создании экземпляра типа.

7.1 Общая схема

Общая форма описания универсального типа имеет вид[2]:

```
class <имя класса><<список параметров типа>>
{
  <члены класса>
}
```

а форма объявления переменной такого типа и создания объекта:

```
<имя класса><<список аргументов типа>> <идентификатор переменной>=
new <имя класса><<список аргументов типа>>
([<список параметров конструктора>]);
```

Пример: создать универсальный класс, имеющий одно поле и методы для доступа к нему.

```
class MyClass<T>
{
  T Value;
  public T Get() { return Value; }
  public void Set(T NewValue) { Value = NewValue; }
}

MyClass<int> ic = new MyClass<int>();
ic.Set(5);
int i = ic.Get();     // i = 5
MyClass<string> sc = new MyClass<string>();
```

[1] Также такие типы называют обобщениями, параметризованными типами

[2] Универсальными могут быть не только классы, но и другие элементы языка, например, интерфейсы

```
sc.Set("Пример");
string s = sc.Get(); // s = "Пример"
ic = sc;             // Ошибка при компиляции
i = sc.Get();        // Ошибка при компиляции
```

Так как универсальные классы поддерживают контроль типов параметров, то это позволяет создавать более защищённые программы (по сравнению с программами, использующими в качестве параметра тип `object`).

Универсальный класс может иметь более одного параметра типа. В этом случае, при описании класса она перечисляются через запятую.

Дополним приведённый выше класс методом, «обнуляющим» значение поля `Value`. При реализации такого метода возникает вопрос: какое значение присвоить в качестве «нулевого», ведь для разных типов это может быть `0`, `null`, `""` и др. Для решения этой проблемы служит оператор `default(<тип>)`:

```
class MyClass<T>
{
  T Value;
  public T Get() { return Value; }
  public void Set(T NewValue) { Value = NewValue; }
  public void Clear() { Value = default(T); }
}
```

7.2 Ограничения по параметрам типа

В простейшем случае, в качестве аргументов типа может выступать любой тип данных. Однако возможна ситуация, когда требуется ввести ограничения на список типов, которые могут быть использованы в качестве аргументов типа.

Общая форма описания класса с ограничениями на параметры типа имеет вид:

```
class <имя класса><<список параметров типа>>
where <параметр 1> : <ограничение 1_1>[,<ограничение 1_2>...]
[where <параметр 2> : <ограничение 2_1>[,<ограничение 2_2>...]...]
{
  <члены класса>
}
```

На параметры могут быть наложения ограничения следующих типов:

- *ограничение на базовый класс*, требующее наличия определённого базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса;
- *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`.

При наличии на один параметр нескольких ограничений первым должно быть указано ограничение `class` либо `struct`, если оно присутствует, или же ограничение на базовый класс, если оно накладывается. Указывать ограничения `class` или `struct` одновременно с ограничением на базовый класс не разрешается. Далее по списку должно следовать ограничение на интерфейс, а последним по порядку – ограничение `new()`.

7.2.1 Ограничение на базовый класс

Ограничение на базовый класс позволяет указывать базовый класс, который должен наследоваться аргументом типа. Ограничение на базовый класс служит двум главным целям:

- во-первых, оно позволяет использовать в универсальном типе доступные члены того базового класса, на который указывает данное ограничение. В отсутствие ограничения на базовый класс компилятору ничего не известно о типе членов, которые может иметь аргумент типа;
- во-вторых, ограничение на базовый класс гарантирует использование только тех аргументов типа, которые поддерживают указанный базовый класс. Это означает, что для любого ограничения, накладываемого на базовый класс, аргумент типа должен обозначать сам базовый класс или производный от него класс. Если же попытаться использовать аргумент типа, не соответствующий указанному базовому классу или не наследующий его, то в результате возникнет ошибка во время компиляции.

Пример: создать универсальный класс, хранящий наследников класса `Figure` и позволяющий рассчитать их общую площадь.

```
class MasFigures<T> where T : Square
{
  T[] Mas = new T[10];
  int Count = 0;
  public void Add(T Fig)
  {
    if (Count < 10)
      Mas[Count++] = Fig;
  }
  public T MaxFigure()
  {
    if (Count > 0)
    {
      T Max = Mas[0];
      for (int i = 1; i < Count; i++)
        if (Max.size < Mas[i].size)
```

```
            Max = Mas[i];
        return Max;
      }
      else
        return null;
    }
}

MasFigures<Square> MasSquare = new MasFigures<Square>();
Square Fig = new Square();
Fig.size = 3;
MasSquare.Add(Fig);
Fig = new Square();
Fig.size = 5;
MasSquare.Add(Fig);
Fig = new Square();
Fig.size = 4;
MasSquare.Add(Fig);
Fig = MasSquare.MaxFigure(); // Вторая фигура, у которой size=5
Rectangle Fig2 = new Rectangle();
MasSquare.Add(Fig2); // Такой действие невозможно
```

Если в классе `MasFigures` заменить ограничение на тип `Figure`, то найти самую большую фигуру через размер будет невозможно, т.к. у класса `Figure` нет размера. Однако, будет возможно реализовать поиск через площадь, и в этом случае добавление в массив прямоугольника будет вполне допустимым.

7.2.2 Ограничение на интерфейс

Ограничение на интерфейс служит тем же целям, что и ограничение на базовый класс.

Пример: создать универсальный класс, хранящий классы с реализацией интерфейса `I2D` и позволяющий рассчитать их общий периметр.

```
class MasI2D<T> where T : I2D
{
  T[] Mas = new T[10];
  int Count = 0;
  public void Add(T Fig)
  {
    if (Count < 10)
      Mas[Count++] = Fig;
  }
  public double SumPerimeter()
  {
    double Sum = 0;
    for (int i = 0; i < Count; i++)
      Sum += Mas[i].Perimeter();
    return Sum;
```

```
  }
}

Square A = new Square();
A.size = 5;
Rectangle B = new Rectangle();
B.size1 = 4;
B.size2 = 7;
Cube C = new Cube();
C.size = 13;
MasI2D<I2D> Mas = new MasI2D<I2D>();
Mas.Add(A);
Mas.Add(B);
Mas.Add(C); // Эта строка не допустима
double SumPerimeter = Mas.SumPerimeter(); // SumPerimeter = 42
```

7.2.3 Ограничение на конструктор

Ограничение на конструктор необходимо, чтобы в универсальном классе можно было создавать объекты класса «аргумент типа». Если такое ограничение не наложить, то не гарантируется, что в классе «аргумент типа» есть конструктор без параметров. Например:

```
class MyClass1
{
  int A;
  public MyClass1() { A = 5; }
  public MyClass1(int NewA) { A = NewA; }
}
class MyClass2
{
  int A;
  public MyClass2(int NewA) { A = NewA; }
}
class MyClass3<T> where T : new()
{
  T B;
  public MyClass3()
  {
    // Без ограничения «new()» в строке заголовка класса
    // нижеприведенная строка будет выдавать ошибку при компиляции
    B = new T();
  }
}

MyClass3<MyClass1> cl1 = new MyClass3<MyClass1>(); // Допустимо
// Следующая строка выдаст ошибку компиляции, т.к. в классе
// MyClass2 нет конструктора без параметров
MyClass3<MyClass2> cl2 = new MyClass3<MyClass2>();
```

7.2.4 Ограничения ссылочного типа и типа значения

Данные ограничения требуют, чтобы «аргумент типа» принадлежал заданной группе типов.

Наложение ограничения ссылочного типа «`class`»:

- позволяет внутри класса работать с параметром типа, как с переменной, допускающей значение `null`, например:

```
class MyClass<T> where T : class
{
  T A;
  public MyClass()
  {
    // Без ограничения «class» в строке заголовка класса
    // нижеприведенная строка будет выдавать ошибку при компиляции
    A = null;
  }
}
```

- блокирует использование в качестве аргумента типа типов значения, например:

```
MyClass<int> i = new MyClass<int>();        // Ошибка компиляции
MyClass<string> s = new MyClass<string>(); // А эта строка допустима
```

Наложение ограничения типа значения «`struct`» приводит прямо к противоположным результатам, например:

```
class MyClass<T> where T : struct
{
  T A;
  public MyClass(T NewA)
  {
    A = null; // Теперь эта строка вызывает ошибку компиляции
    A = NewA; // А эта строка вполне допустима
  }
}

MyClass<int> i = new MyClass<int>(10);        // Эта строка допустима
MyClass<string> s = new MyClass<string>(10); // Ошибка компиляции
```

7.2.5 Установление связи между двумя параметрами с помощью ограничения

Ещё одной особенностью ограничений является возможность требования наличия связи «родитель-потомок» между параметрами типа. Например, если требуется, чтобы параметр типа `V` был таким же, или являлся наследником параметра типа T, то запись такого ограничения будет иметь вид:

```
class MyClass<T, V> where V : T { ... }
```

Например для двух классов A и B

```
Class A { ... }
Class B : A { ... }
```

и указанного выше универсального типа MyClass, рассмотрим следующие описания переменных и создания объектов:

```
MyClass<A, B> cl1 = new MyClass<A, B>(); // Допустимо
MyClass<A, A> cl2 = new MyClass<A, A>(); // Допустимо
MyClass<B, A> cl3 = new MyClass<B, A>(); // Не допустимо
MyClass<B, B> cl3 = new MyClass<B, B>(); // Допустимо
```

7.3 Параметры типы в методах

Универсальность может применяться не только в универсальных типах, но и в отдельных методах, в т.ч. в обычных классов. В этом случае после имени метода указывается параметр типа (или параметры), который может быть использован в параметрах метода и в теле реализации метода.

Пример: создать класс, метод которого позволяет заменить все вхождения искомого значения на заданное.

```
class MyClass
{
  public static void Replace<T>(T[] Mas, T Old, T New)
  {
    for (int i = 0; i < Mas.Length; i++)
      if (Mas[i].Equals(Old))
        Mas[i] = New;
  }
}

int[] IntMas = { 1, 2, 1, 3, 4, 1 };
MyClass.Replace(IntMas, 1, 5); // IntMas = {5, 2, 5, 3, 4, 5}
string[] StrMas = { "один", "два", "один" };
MyClass.Replace(StrMas, "один", "три");
                                // StrMas = {"три", "два", "три"}
```

Как видно из примера, при вызове такого метода указывать фактический тип данных не требуется, компилятор его распознает автоматически. Если компилятор не может распознать тип данных, то формируется ошибка компиляции. Например, в следующей строке невозможно распознать тип <T>, т.к. на основе массива он должен быть int, а на основе параметра Old – double:

```
MyClass.Replace(IntMas, 1.0, 5.0); // Ошибка компиляции
```

При вызове метода возможно явное указание используемого типа. для чего после имени метода указывается требуемый тип, например:

```
MyClass.Replace<int>(IntMas, 1, 5);     // Допустимо
MyClass.Replace<string>(IntMas, 1, 5); // Ошибка компиляции
```

На параметр типа, используемый в методе, также могут накладываться ограничения. Они имеют ту же конструкцию, что и ограничения на параметр типа универсального типа. Расположение ограничений осуществляется после закрывающейся круглой скобки метода. Например, если наложить ограничение `struct` на параметр из приведённого выше примера, то работа с массивом строк станет недопустимой:

```
class MyClass
{
  public static void Replace<T>(T[] Mas, T Old, T New)
    where T : struct
  {
    for (int i = 0; i < Mas.Length; i++)
      if (Mas[i].Equals(Old))
        Mas[i] = New;
  }
}

int[] IntMas = { 1, 2, 1, 3, 4, 1 };
MyClass.Replace(IntMas, 1, 5); // IntMas = {5, 2, 5, 3, 4, 5}
string[] StrMas = { "один", "два", "один" };
MyClass.Replace(StrMas, "один", "три"); // Ошибка компиляции
```

7.4 Некоторые универсальные типы C#

Ниже рассматриваются некоторые распространённые универсальные типы C#, а также классы, в которых есть методы с параметрами типов[1].

7.4.1 Класс Array

Данный класс выступает в роли базового класса для всех массивов. Помимо членов класса, указанных в разделе «Массивы», класс имеет ряд статическим методов, некоторые из которых являются методами с параметром типа. В таблице 7.1 приведены некоторые статические методы класса `Array`.

[1] Для многих приведённых здесь классов имеются аналоги, работающие с классом `object`, т.е. позволяющие одновременно хранить разнотипные элементы

Таблица 7.1 – Некоторые статические методы класса `Array`

Наименование	Описание

`Resize<T>` `(ref T[] array, int newSize)`	Устанавливает длину массива `array` в значение `newSize` без потери существующих значений. Например:
	`class MyClass`
	`{`
	`  public int value;`
	`  public MyClass(int NewValue) {value = NewValue;}`
	`}`
	
	`MyClass[] Mas = new MyClass[4];`
	`Mas[0] = new MyClass(1);`
	`Mas[1] = new MyClass(2);`
	`Mas[2] = new MyClass(3);`
	`Mas[3] = new MyClass(4); // Значения {1,2,3,4}`
	`Array.Resize(ref Mas,3); // Значения {1,2,3}`
	`Array.Resize(ref Mas,5);`
	`// Значения {1,2,3,null,null}`

`IndexOf<T>` `(T[] array, T value)`
`LastIndexOf<T>` `(T[] array, T value)`
	Производят поиск в массиве `array` первого/последнего элемента, равного `value`, и возвращает его позицию. Если элемент не найден, то возвращается -1. Если класс `T` создан пользователем, то в нем должен быть переопределён метод `Equals`. Например:
	`class MyClass`
	`{`
	`  public int value;`
	`  public MyClass(int NewValue) {value = NewValue;}`
	`  public override bool Equals(object obj)`
	`  {`
	`    if (obj is MyClass)`
	`      return (obj as MyClass).value == value;`
	`    else`
	`      return false;`
	`  }`
	`}`
	
	`MyClass[] Mas = new MyClass[6];`
	`Mas[0] = new MyClass(4);`
	`Mas[1] = new MyClass(2);`
	`Mas[2] = new MyClass(6);`
	`Mas[3] = new MyClass(4);`
	`Mas[4] = new MyClass(2);`
	`Mas[5] = new MyClass(7);`
	`MyClass Find = new MyClass(2);`
	`int i1 = Array.IndexOf(Mas, Find);  // i1 = 1`
	`int i2 = Array.LastIndexOf(Mas, Find); // i2 = 4`
	`Find = new MyClass(3);`
	`int i3 = Array.IndexOf(Mas, Find);  // i3 = -1`

Продолжение таблицы 7.1

Наименование	Описание

FindIndex<T> (T[] array, Predicate<T> match) **FindLastIndex<T>** (T[] array, Predicate<T> match)	Производят поиск в массиве `array` первого/последнего элемента, удовлетворяющего условию `match`, и возвращает его позицию. Если элемент не найден, то возвращается -1. Для осуществления поиска должен быть реализован метод сравнения. Например:
	`class MyClass`
	`{`
	`  public int value;`
	`  public MyClass(int NewValue) {value = NewValue;}`
	`  public static int V;`
	`  public static bool Upper(MyClass Find)`
	`  {`
	`  return Find.value > V;`
	`  }`
	`}`
	
	`MyClass[] Mas = new MyClass[6];`
	`Mas[0] = new MyClass(4);`
	`Mas[1] = new MyClass(2);`
	`Mas[2] = new MyClass(6);`
	`Mas[3] = new MyClass(4);`
	`Mas[4] = new MyClass(2);`
	`Mas[5] = new MyClass(7);`
	`MyClass.V = 5;`
	`int i1 = Array.FindIndex(Mas, MyClass.Upper);`
	`// i1 = 2;`
	`int i2 = Array.FindLastIndex(Mas, MyClass.Upper);`
	`// i2 = 5;`
	`MyClass.V = 8;`
	`int i3 = Array.FindIndex(Mas, MyClass.Upper);`
	`// i3 = -1`

Find<T> (T[] array, Predicate<T> match) **FindLast<T>** (T[] array, Predicate<T> match) **FindAll<T>** (T[] array, Predicate<T> match)	Методы `Find` и `FindLast` производят поиск в массиве `array` первого/последнего элемента, удовлетворяющего условию `match`, и возвращает его. Если элемент не найден, то возвращается значение по умолчанию для типа `T`. Метод `FindAll` производит поиск в массиве `array` всех элементов, удовлетворяющего условию `match`, и возвращает их в виде массива. Если элементы не найдены, то возвращается пустой массив. Для осуществления поиска должен быть реализован метод сравнения.

Продолжение таблицы 7.1

Наименование	Описание

Reverse
(Array array
[,int index,
int length])
	Изменяет порядок элементов во всем одномерном массиве (или `length` элементов начиная с позиции `index`) на обратный. Например:
	`int[] Mas = { 1, 2, 3, 4 };`
	`Array.Reverse(Mas);  // Mas = {4, 3, 2, 1}`
	`Array.Reverse(Mas, 1, 2); // Mas = {4, 2, 3, 1}`

Sort<T>
(T[] array)
	Сортирует массив `array`, используя для сортировки метод `CompareTo()` интерфейса `IComparable<T>` элемента массива. Если класс `T` создан пользователем, то в нем должен быть реализован данный интерфейс. Например:
	`class MyClass : IComparable<MyClass>`
	`{`
	`  public int value;`
	`  public MyClass(int NewValue) {value = NewValue;}`
	`  public int CompareTo(MyClass other)`
	`  {`
	`  return value - other.value;`
	`  }`
	`}`
	
	`MyClass[] Mas = new MyClass[6];`
	`Mas[0] = new MyClass(4);`
	`Mas[1] = new MyClass(2);`
	`Mas[2] = new MyClass(6);`
	`Mas[3] = new MyClass(4);`
	`Mas[4] = new MyClass(2);`
	`Mas[5] = new MyClass(7);`
	`Array.Sort(Mas);`
	`// Mas = {2, 2, 4, 4, 6, 7}`

Продолжение таблицы 7.1

Наименование	Описание

Sort<T> (T[] array, IComparer<T> comparer)	Сортирует массив array, используя для сортировки объект comparer некоторого класса, реализующего интерфейс IComparer<T>. Например:
	`class MyClass`
	`{`
	`  public int value;`
	`  public MyClass(int NewValue) {value = NewValue;}`
	`}`
	
	`class ReverseCompare : IComparer<MyClass>`
	`{`
	`  public int Compare(MyClass x, MyClass y)`
	`  {`
	`  return y.value - x.value;`
	`  }`
	`}`
	
	`MyClass[] Mas = new MyClass[6];`
	`Mas[0] = new MyClass(4);`
	`Mas[1] = new MyClass(2);`
	`Mas[2] = new MyClass(6);`
	`Mas[3] = new MyClass(4);`
	`Mas[4] = new MyClass(2);`
	`Mas[5] = new MyClass(7);`
	`ReverseCompare Comparer = new ReverseCompare();`
	`Array.Sort(Mas, Comparer);`
	`// Mas = {7, 6, 4, 4, 2, 2}`

7.4.2 Класс List<T>

Представляет строго типизированный список объектов, доступных по индексу. Поддерживает методы для поиска по списку, выполнения сортировки и других операций со списками. Некоторые члены класса приведены в таблице 7.2.

Таблица 7.2 – Некоторые члены класса List<T>

Наименование	Описание
Capacity	Возвращает или задаёт общее число элементов, которые может вместить внутренняя структура данных без изменения размера.

Продолжение таблицы 7.2

Наименование	Описание

Count	Возвращает количество элементов, уже находящихся в списке. Например:
	`List<int> l = new List<int>();`
	`int i1 = l.Count; // i1 = 0;`
	`l.Add(5);`
	`int i2 = l.Count; // i1 = 1;`

[int index**]**	Возвращает или задаёт элемент по указанному индексу. Например:
	`List<int> l = new List<int>();`
	`l.Add(5);`
	`int i = l[0]; // i = 5`

Add
(T item)
	Добавляет элемент в конец списка. Например:
	`List<int> l = new List<int>(); // Список пуст`
	`l.Add(5); // В списке один элемент`
	`l.Add(2); // В списке два элемента`
	`l.Add(4); // В списке три элемента`

Insert
(int index,
T item)
	Вставляет элемент `item` в позицию `index`. Значение `index` должно быть в диапазоне от 0 до `Count`. Например:
	`List<int> l = new List<int>();`
	`l.Add(5);  // l = {5}`
	`l.Add(2);  // l = {5, 2}`
	`l.Insert(1,7); // l = {5, 7, 2}`

RemoveAt
(int index)
	Удаляет элемент из позиции `index`. Значение `index` должно быть в диапазоне от 0 до `Count-1`. Например:
	`List<int> l = new List<int>();`
	`l.Add(5);  // l = {5}`
	`l.Add(2);  // l = {5, 2}`
	`l.Add(7);  // l = {5, 2, 7}`
	`l.RemoveAt(1); // l = {5, 7}`

Clear()	Удаляет все элементы из списка. Например
	`List<int> l = new List<int>(); // Список пуст`
	`l.Add(5);  // В списке один элемент`
	`l.Add(2);  // В списке два элемента`
	`l.Clear(); // Список снова пуст`

Remove
(T item)
	Удаляет первое вхождение элемента `item`. Возвращает `true`, если элемент был удалён (иначе `false`). Например:
	`List<int> l = new List<int>();`
	`l.Add(2);  // l = {2}`
	`l.Add(3);  // l = {2, 3}`
	`l.Add(2);  // l = {2, 3, 2}`
	`l.Remove(2); // l = {3, 2}`

Продолжение таблицы 7.2

Наименование	Описание

RemoveAll (Predicate<T> match)	Удаляет все элементы, удовлетворяющие условию match. Для осуществления удаления должен быть реализован метод сравнения. Возвращает количество удалённых элементов. Например:
	`class MyClass`
	`{`
	`  public int value;`
	`  public MyClass(int NewValue) {value = NewValue;}`
	`  public static int V;`
	`  public static bool Upper(MyClass Find)`
	`  {`
	`  return Find.value > V;`
	`  }`
	`}`
	`List<MyClass> l = new List<MyClass>();`
	`l.Add(new MyClass(4));`
	`l.Add(new MyClass(2));`
	`l.Add(new MyClass(6));`
	`l.Add(new MyClass(2));`
	`l.Add(new MyClass(7));`
	`MyClass.V = 4;`
	`l.RemoveAll(MyClass.Upper); // l = {4, 2, 2}`

RemoveRange (int index, int count)	Удаляет count элементов из списка начиная с позиции index. Значения index и index+count должны быть в диапазоне от 0 до Count. Например:
	`List<int> l = new List<int>();`
	`l.Add(5);  // l = {5}`
	`l.Add(2);  // l = {5, 2}`
	`l.Add(7);  // l = {5, 2, 7}`
	`l.Add(3);  // l = {5, 2, 7, 3}`
	`l.RemoveRange(1, 2); // l = {5, 3}`

Также имеются методы IndexOf, LastIndexOf, FindIndex, FindLastIndex, Find, FindLast, FindAll, Sort, Reverse, аналогичные методам класса Array.	

7.4.3 Класс LinkedList<T>

Представляет собой двусвязный список, каждый узел которого является экземпляром класса LinkedListNode<T> и обладает рядом свойств, некоторые из которых приведены в таблице 7.3.

Таблица 7.3 – Некоторые свойства класса `LinkedListNode<T>`

Наименование	Описание
`List`	Позволяет получить доступ к списку класса `LinkedList<T>`, которому принадлежит узел.
`Next`	Позволяет получить следующий (относительно данного) элемент списка класса `LinkedListNode<T>`. Если данный элемент является последним, то возвращается `null`.
`Previous`	Позволяет получить предыдущий (относительно данного) элемент списка класса `LinkedListNode<T>`. Если данный элемент является первым, то возвращается `null`.
`Value`	Возвращает значение, содержащееся в узле, класса `T`.

Некоторые члены класса `LinkedList<T>` приведены в таблице 7.4.

Таблица 7.4 – Некоторые члены класса `LinkedList<T>`

Наименование	Описание
`Count`	Возвращает количество узлов в списке
`First`	Возвращает первый узел списка класса `LinkedListNode<T>`, или `null`, если список пуст.
`Last`	Возвращает последний узел списка класса `LinkedListNode<T>`, или `null`, если список пуст.

`AddLast`
`(T value)`
	Добавляет новый узел со значением `value` в конец списка и возвращает ссылку на него класса `LinkedListNode<T>`. Например:
	`LinkedList<int> l = new LinkedList<int>();`
	`l.AddLast(5);  // l = {5}`
	`l.AddLast(8);  // l = {5, 8}`

`AddFirst`
`(T value)`
	Добавляет новый узел со значением `value` в начало списка и возвращает ссылку на него класса `LinkedListNode<T>`. Например:
	`LinkedList<int> l = new LinkedList<int>();`
	`l.AddFirst(5);  // l = {5}`
	`l.AddFirst(8);  // l = {8, 5}`

`AddAfter`
`(LinkedListNode<T> node,`
`T value)`
	Добавляет новый узел со значением `value` после узла `node` и возвращает ссылку на него класса `LinkedListNode<T>`. Например:
	`LinkedList<int> l = new LinkedList<int>();`
	`l.AddLast(5);  // l = {5}`
	`l.AddLast(8);  // l = {5, 8}`
	`LinkedListNode<int> node = l.First; // node = {5}`
	`l.AddAfter(node, 9);  // l = {5, 9, 8}`

Продолжение таблицы 7.4

Наименование	Описание

AddBefore (LinkedListNode<T> node, T value)	Добавляет новый узел со значением value перед узлом node и возвращает ссылку на него класса LinkedListNode<T>. Например:
	`LinkedList<int> l = new LinkedList<int>();`
	`l.AddLast(5);  // l = {5}`
	`l.AddLast(8);  // l = {5, 8}`
	`LinkedListNode<int> node = l.First; // node = {5}`
	`l.AddBefore(node, 9); // l = {9, 5, 8}`

Clear()	Удаляет все узлы из списка

RemoveFirst()	Удаляет узел в начале списка. Например:
	`LinkedList<int> l = new LinkedList<int>();`
	`l.AddLast(5);  // l = {5}`
	`l.AddLast(8);  // l = {5, 8}`
	`l.RemoveFirst();  // l = {8}`

RemoveLast()	Удаляет узел в конце списка. Например:
	`LinkedList<int> l = new LinkedList<int>();`
	`l.AddLast(5);  // l = {5}`
	`l.AddLast(8);  // l = {5, 8}`
	`l.RemoveLast();  // l = {5}`

Remove (T value)	Удаляет первый узел, имеющий значение value. Если удаление произведено, то возвращается true (иначе false). Например:
	`LinkedList<int> l = new LinkedList<int>();`
	`l.AddLast(5);  // l = {5}`
	`l.AddLast(8);  // l = {5, 8}`
	`l.AddLast(9);  // l = {5, 8, 9}`
	`l.Remove(8);  // l = {5, 9}`

Find (T value)	Возвращает первый узел (класса LinkedListNode<T>), имеющий значение value. Если узел не найден, то возвращается null. Например:
	`LinkedList<int> l = new LinkedList<int>();`
	`l.AddLast(5);  // l = {5}`
	`l.AddLast(8);  // l = {5, 8}`
	`l.AddLast(9);  // l = {5, 8, 9}`
	`l.AddLast(8);  // l = {5, 8, 9, 8}`
	`LinkedListNode<int> node = l.Find(8);`
	`// node = 8 (второй узел)`
	`node = l.Find(6);  // node = null`

Продолжение таблицы 7.4

Наименование	Описание

FindLast
(T value)
	Возвращает последний узел (класса LinkedListNode<T>), имеющий значение value. Если узел не найден, то возвращается null. Например:
	
	`LinkedList<int> l = new LinkedList<int>();`
	`l.AddLast(5);  // l = {5}`
	`l.AddLast(8);  // l = {5, 8}`
	`l.AddLast(9);  // l = {5, 8, 9}`
	`l.AddLast(8);  // l = {5, 8, 9, 8}`
	`LinkedListNode<int> node = l.FindLast(8);`
	`// node = 8 (четвёртый узел)`
	`node = l.FindLast(6); // node = null`

7.4.4 Класс Queue<T>

Представляет собой класс очереди, работающей по принципу FIFO («первый пришёл – первый ушёл»). Некоторые члены класса приведены в таблице 7.5.

Таблица 7.5 – Некоторые члены класса Queue<T>

Наименование	Описание
Count	Возвращает количество элементов в очереди

Enqueue
(T item)
	Добавляет в конец очереди новый элемент со значением item. Например:
	
	`Queue<int> q = new Queue<int>();`
	`q.Enqueue(5);  // q = {5}`
	`q.Enqueue(8);  // q = {5, 8} (начало слева)`

Dequeue()	Забирает из начала очереди элемент класса T и возвращает его. Если очередь пуста, то при выполнении операции возникнет исключение. Например:
	
	`Queue<int> q = new Queue<int>();`
	`q.Enqueue(5);  // q = {5}`
	`q.Enqueue(8);  // q = {5, 8} (начало слева)`
	`int i = q.Dequeue(); // i = 5, q = {8}`

Peek()	Получает из начала очереди элемент класса T но не забирает его из очереди. Если очередь пуста, то при выполнении операции возникнет исключение. Например:
	
	`Queue<int> q = new Queue<int>();`
	`q.Enqueue(5);  // q = {5}`
	`q.Enqueue(8);  // q = {5, 8} (начало слева)`
	`int i = q.Peek();  // i = 5, q = {5, 8}`

Clear()	Удаляет все элементы из очереди

7.4.5 Класс Stack<T>

Представляет собой класс очереди, работающей по принципу LIFO («последний пришёл – первый ушёл»), называемую также стеком. Некоторые члены класса приведены в таблице 7.6.

Таблица 7.6 – Некоторые члены класса `Stack<T>`

Наименование	Описание
`Count`	Возвращает количество элементов в стеке

`Push`
`(T item)`
	Добавляет в вершину стека новый элемент со значением `item`. Например:
	
	`Stack<int> st = new Stack<int>();`
	`st.Push(5);  // st = {5}`
	`st.Push(8);  // st = {8, 5} (верх слева)`

`Pop``()`	Забирает из вершины стека элемент класса `T` и возвращает его. Если стек пуст, то при выполнении операции возникнет исключение. Например:
	
	`Stack<int> st = new Stack<int>();`
	`st.Push(5);  // st = {5}`
	`st.Push(8);  // st = {8, 5} (верх слева)`
	`int i = st.Pop();  // i = 8, st = {5}`

`Peek``()`	Получает из вершины стека элемент класса `T` но не забирает его из стека. Если стек пуст, то при выполнении операции возникнет исключение. Например:
	
	`Stack<int> st = new Stack<int>();`
	`st.Push(5);  // st = {5}`
	`st.Push(8);  // st = {8, 5} (верх слева)`
	`int i = st.Peek();  // i = 8, st = {8, 5}`

`Clear``()`	Удаляет все элементы из стека

7.4.6 Классы SortedSet<T> и HashSet<T>

Представляют собой множества объектов. Множество может содержать только уникальные объекты.

Отличие между классами заключается в том, что класс `SortedSet<T>` упорядоченный, а класс `HashSet<T>` – неупорядоченный, но высокопроизводительный.

Если класс `T` пользовательский, то в нем должен быть реализован интерфейс `IComparable<T>`. Также во многих случаях требуется перегрузка метода `Equals`.

Некоторые члены классов приведены в таблице 7.7.

Таблица 7.7 – Некоторые члены классов `SortedSet<T>` и `HashSet<T>`[1]

Наименование	Описание
`Count`	Возвращает количество элементов в множестве
`Max`	Возвращает объект класса `T`, имеющий максимальное значение. Если множество пустое, то возвращается значение по умолчанию для класса `T`.
`Min`	Возвращает объект класса `T`, имеющий минимальное значение. Если множество пустое, то возвращается значение по умолчанию для класса `T`.

`Add`
`(T item)`
	Добавляет в множество новый элемент со значением `item` и возвращает результат добавления типа `bool` (`true`, если добавление произведено, и `false` в противном случае). Например:
	
	`SortedSet<int> s = new SortedSet<int>();`
	`bool b = s.Add(8); // b = true,  s = {8}`
	`b = s.Add(5);  // b = true,  s = {5, 8}`
	`b = s.Add(8);  // b = false, s = {5, 8}`

`Clear``()`	Удаляет все элементы из множества

`Contains`
`(T item)`
	Проверяет, входит ли элемент `item` во множество. Возвращает значение типа `bool`. Например:
	
	`SortedSet<int> s = new SortedSet<int>();`
	`s.Add(5);  // s = {5}`
	`s.Add(2);  // s = {2, 5}`
	`s.Add(7);  // s = {2, 5, 7}`
	`bool b = s.Contains(7);  // b = true`
	`b = s.Contains(8);  // b = false`

`Remove`
`(T item)`
	Удаляет из множества элемент со значением `item` и возвращает результат удаления типа `bool` (`true`, если удаление произведено, и `false` в противном случае). Например:
	
	`SortedSet<int> s = new SortedSet<int>();`
	`bool b = s.Add(8); // b = true,  s = {8}`
	`b = s.Add(5);  // b = true,  s = {5, 8}`
	`b = s.Remove(8);  // b = true,  s = {5}`
	`b = s.Remove(9);  // b = false, s = {5}`

[1] Примеры приводятся для класса `SortedSet<T>`

Продолжение таблицы 7.7

Наименование	Описание

RemoveWhere (Predicate<T> match)	Удаляет из множества элементы, удовлетворяющие условию match и возвращает количество удалённых элементов. Требуется реализация метода, проверяющего требуемое условие. Например:
	`class MyClass : IComparable<MyClass>`
	`{`
	`  public int value;`
	`  public MyClass(int NewValue)`
	`  { value = NewValue; }`
	`  public static int V;`
	`  public static bool Upper(MyClass Find)`
	`  { return Find.value > V; }`
	`  public int CompareTo(MyClass other)`
	`  { return value - other.value; }`
	`}`
	
	`SortedSet<MyClass> s = new SortedSet<MyClass>();`
	`s.Add(new MyClass(5));  // s = {5}`
	`s.Add(new MyClass(2));  // s = {2, 5}`
	`s.Add(new MyClass(7));  // s = {2, 5, 7}`
	`s.Add(new MyClass(3));  // s = {2, 3, 5, 7}`
	`MyClass.V = 4;`
	`int i = s.RemoveWhere(MyClass.Upper);`
	`// i = 2, s = {2, 3}`
	`i = s.RemoveWhere(MyClass.Upper);`
	`// i = 0, s = {2, 3}`

IntersectWith (IEnumerable<T> other)	Изменяет множество так, чтобы оно содержало только элементы, входящие в коллекцию other. Например:
	`SortedSet<int> s1 = new SortedSet<int>();`
	`s1.Add(5);`
	`s1.Add(2);`
	`s1.Add(7);`
	`s1.Add(3);  // s1 = {2, 3, 5, 7}`
	`SortedSet<int> s2 = new SortedSet<int>();`
	`s2.Add(4);`
	`s2.Add(2);`
	`s2.Add(7);  // s2 = {2, 4, 7}`
	`s1.IntersectWith(s2);  // s1 = {2, 7}`

Продолжение таблицы 7.7

Наименование	Описание

UnionWith (IEnumerable<T> other)	Изменяет множество так, чтобы оно содержало элементы, входящие как в множество, так и в коллекцию other. Например:
	`SortedSet<int> s1 = new SortedSet<int>();`
	`s1.Add(5);`
	`s1.Add(2);`
	`s1.Add(7);  // s1 = {2, 5, 7}`
	`SortedSet<int> s2 = new SortedSet<int>();`
	`s2.Add(4);`
	`s2.Add(9);`
	`s2.Add(7);  // s2 = {4, 7, 9}`
	`s1.UnionWith(s2);  // s1 = {2, 4, 5, 7, 9}`

ExceptWith (IEnumerable<T> other)	Изменяет множество так, чтобы оно содержало элементы, не входящие в коллекцию other. Например:
	`SortedSet<int> s1 = new SortedSet<int>();`
	`s1.Add(5);`
	`s1.Add(2);`
	`s1.Add(7);  // s1 = {2, 5, 7}`
	`SortedSet<int> s2 = new SortedSet<int>();`
	`s2.Add(4);`
	`s2.Add(7);  // s2 = {4, 7}`
	`s1.ExceptWith(s2);  // s1 = {2, 5}`

IsSubsetOf (IEnumerable<T> other)	Проверяет, является ли множество подмножеством коллекции other. Например:
	`SortedSet<int> s1 = new SortedSet<int>();`
	`s1.Add(5);`
	`s1.Add(2);  // s1 = {2, 5}`
	`SortedSet<int> s2 = new SortedSet<int>();`
	`s2.Add(2);`
	`s2.Add(7);`
	`s2.Add(5);  // s2 = {2, 5, 7}`
	`SortedSet<int> s3 = new SortedSet<int>();`
	`s2.Add(2);`
	`s2.Add(7);`
	`s2.Add(4);  // s3 = {2, 4, 7}`
	`bool b = s1.IsSubsetOf(s2);  // b = true`
	`b = s1.IsSubsetOf(s3);  // b = false`

Продолжение таблицы 7.7

Наименование	Описание

IsSupersetOf (IEnumerable<T> other)	Проверяет, является ли множество надмножеством коллекции other. Например:
	`SortedSet<int> s1 = new SortedSet<int>();`
	`s1.Add(5);`
	`s1.Add(2);`
	`s1.Add(7);  // s1 = {2, 5, 7}`
	`SortedSet<int> s2 = new SortedSet<int>();`
	`s2.Add(2);`
	`s2.Add(5);  // s2 = {2, 5}`
	`SortedSet<int> s3 = new SortedSet<int>();`
	`s3.Add(2);`
	`s3.Add(4);  // s3 = {2, 4}`
	`bool b = s1.IsSupersetOf(s2); // b = true`
	`b = s1.IsSupersetOf(s3);  // b = false`

SetEquals (IEnumerable<T> other)	Проверяет, содержат ли множество и коллекция other только одинаковые элементы. Например:
	`SortedSet<int> s1 = new SortedSet<int>();`
	`s1.Add(5);`
	`s1.Add(2);  // s1 = {2, 5}`
	`SortedSet<int> s2 = new SortedSet<int>();`
	`s2.Add(2);`
	`s2.Add(5);  // s2 = {2, 5}`
	`SortedSet<int> s3 = new SortedSet<int>();`
	`s3.Add(2);  // s3 = {2}`
	`SortedSet<int> s4 = new SortedSet<int>();`
	`s4.Add(2);`
	`s4.Add(5);`
	`s4.Add(7);  // s4 = {2, 5, 7}`
	`bool b = s1.SetEquals(s2);  // b = true`
	`b = s1.SetEquals(s3);  // b = false`
	`b = s1.SetEquals(s4);  // b = false`

7.4.7 Классы Dictionary<TKey, TValue> и SortedDictionary<TKey, TValue>

Представляют собой словари пар «ключей-значений». Ключи в словаре должны быть уникальными и не могут содержать значение null.

Если ключ представляет собой пользовательский тип, то требуется выполнить одно из двух действий:

- создать класс, реализующий интерфейс IEqualityComparer<T> / IComparer<T> и использовать объект этого класса при создании словаря для обеспечения сравнения ключей;
- реализовать перегрузку методов Equals и GetHashCode.

Если значение представляет собой пользовательский тип, то требуется перегрузка методов `Equals` и `GetHashCode`.

Порядок хранения пар в классе `Dictionary<TKey, TValue>` не определён, в классе `SortedDictionary<TKey, TValue>` – отсортированы по ключу.

Некоторые члены классов приведены в таблице 7.8.

Таблица 7.8 – Некоторые члены классов `Dictionary<TKey, TValue>` и `SortedDictionary<TKey, TValue>`

Наименование	Описание
`Count`	Возвращает количество пар в словаре

`[``TKey key`**`]`**	Возвращает или задаёт значение, связанное с ключом `key`. Если в словаре ещё нет ключа, которому присваивается значение, то производится добавление пары «ключ-значение». Если в словаре нет ключа, значение которого запрашивается, то возникает исключение. Например:
	
	`Dictionary<int, string> d =`
	`  new Dictionary<int, string>();`
	`d[1] = "понедельник"; // d.Count = 1`
	`d[2] = "вторник";     // d.Count = 2`
	`d[1] = "среда";       // d.Count = 2`
	`string s = d[3];      // Исключение`

`Add`
`(TKey key,`
`TValue value)`
	Добавляет новую пару «ключ-значение» с ключом `key` и значением `value` в словарь. Если пара с таким ключом уже имеется, то возникает исключение. Например:
	
	`Dictionary<int, string> d =`
	`  new Dictionary<int, string>();`
	`d.Add(1, "понедельник");`
	`d.Add(2, "вторник");`
	`d.Add(1, "среда");    // Исключение`

`Remove`
`(TKey key)`
	Удаляет из словаря пару с ключом `key`. Возвращает значение типа `bool` показывающее, было ли проведено удаление. Например:
	
	`Dictionary<int, string> d =`
	`  new Dictionary<int, string>();`
	`d.Add(1, "понедельник");`
	`d.Add(2, "вторник");`
	`bool b = d.Remove(1); // b = true,  d.Count = 1`
	`b = d.Remove(1);      // b = false, d.Count = 1`

`Clear``()`	Удаляет из словаря все пары.

Продолжение таблицы 7.8

Наименование	Описание

ContainsKey (TKey key)	Определяет, есть ли в словаре пара с ключом key, и возвращает значение типа bool. Например:
	
	`Dictionary<int, string> d =`
	`  new Dictionary<int, string>();`
	`d.Add(1, "понедельник");`
	`d.Add(2, "вторник");`
	`bool b = d.ContainsKey(1); // b = true`
	`b = d.ContainsKey(3);  // b = false`

ContainsValue (TValue value)	Определяет, есть ли в словаре пара с значением value, и возвращает значение типа bool. Например:
	
	`Dictionary<int, string> d =`
	`  new Dictionary<int, string>();`
	`d.Add(1, "понедельник");`
	`d.Add(2, "вторник");`
	`bool b = d.ContainsValue("вторник"); // b = true`
	`b = d.ContainsValue("среда");  // b = false`

TryGetValue (TKey key, out TValue value)	Получает значение value для ключа key, если такой ключ существует, или значение по умолчанию, если ключ не существует. Возвращает значение типа bool показывающее, был ли найден ключ. Например:
	
	`Dictionary<int, string> d =`
	`  new Dictionary<int, string>();`
	`d.Add(1, "понедельник");`
	`d.Add(2, "вторник");`
	`string s; // s = null`
	`bool b = d.TryGetValue(1,out s);`
	`// b = true,  s = "понедельник"`
	`b = d.TryGetValue(3, out s);`
	`// b = false, s = null`

8 Работа с файлами

Для работы с файлами C# предоставляет набор классов, обеспечивающих, как операции манипулирования файлами (например, копирование, удаление, открытие и т.д.), так и операции работы с данными, содержащимися в файле (чтение, запись).

8.1 Класс File

Класс `File` предоставляет статические методы для создания, копирования, удаления, перемещения и открытия файлов, а также помогает при создании объектов `FileStream`.

Класс расположен в пространстве имён `System.IO`.

По умолчанию всем пользователям предоставляется полный доступ к новым файлам с правом на чтение и запись. Однако для настройки некоторых методов класса могут использоваться перечисления, указанный в таблице 8.1.

Таблица 8.1 – Перечисления, используемые для настройки методов класса `File`

Наименование	Назначение	Значения
`FileMode`	Определяет режим работы с файлом	• `CreateNew` – создание нового файла. Если файл существует, то вызывается исключение; • `Create` – создание нового файла. Если файл существует, то он перезаписывается; • `Open` – открытие файла. Если файл не существует, то вызывается исключение; • `OpenOrCreate` – открытие файла. Если файл не существует, то создаётся новый файл; • `Truncate` – открытие файла и удаление из него всех данных; • `Append` – открытие файла на дозапись.
`FileAccess`	Определяет доступные операции для файла	• `Read` – открытие файла для чтения; • `Write` – открытие файла для записи; • `ReadWrite` – открытие файла для чтения и записи.
`FileShare`	Определяет тип совместного доступа к файлу	• `None` – запрет доступа; • `Read` – разрешает доступ к файлу для чтения; • `Write` – разрешает доступ к файлу для записи; • `ReadWrite` – разрешает доступ к файлу для чтения и для записи; • `Delete` – разрешает доступ к файлу для удаления.

Некоторые члены класса `File` приведены в таблице 8.2.

Таблица 8.2 – Некоторые члены класса `File`

Наименование	Описание
Copy (string sourceFileName, string destFileName)	Копирует файл `sourceFileName` в `destFileName`
Delete (string path)	Удаляет файл `path`
Exists (string path)	Возвращает `true`, если файл `path` существует, и `false` в противном случае
Move (string sourceFileName, string destFileName)	Перемещает или переименовывает файл `sourceFileName` в `destFileName`
Replace (string sourceFileName, string destinationFileName, string destinationBackupFileName)	Заменяет содержимое файла `destinationFileName` на содержимое файла `sourceFileName` создавая резервную копию `destinationBackupFileName`. Если копию создавать не требуется, то вместо третьего параметра указывается `null`
Open (string path, FileMode mode [,FileAccess access [,FileShare share]])	Открывает файл `path` с требуемыми параметрами `mode`, `access`, `share`. Возвращает объект класса `FileStream` для работы с файлом
Create (string path)	Создает новый или перезаписывает существующий файл `path`. Возвращает объект класса `FileStream` для работы с файлом
CreateText (string path)	Создает новый или перезаписывает существующий файл `path` в кодировке UTF-8. Возвращает объект класса `StreamWriter` для записи в файл
OpenRead (string path)	Открывает файл `path` для чтения. Возвращает объект класса `FileStream` для чтения из файла
OpenWrite (string path)	Открывает файл `path` для записи. Возвращает объект класса `FileStream` для записи в файл
OpenText (string path)	Открывает файл `path` для чтения. Возвращает объект класса `StreamReader`, позволяющий читать данные из файла с текстом в кодировке UTF-8
AppendText (string path)	Открывает файл `path` для дозаписи. Возвращает объект класса `StreamWriter`, позволяющий добавлять в существующий файл текст в кодировке UTF-8

Продолжение таблицы 8.2

Наименование	Описание
ReadAllText (string path)	Возвращает строку, содержащую все строки файла path
ReadAllLines (string path)	Возвращает массив строк, содержащихсяв файле path
ReadAllBytes (string path)	Возвращает массив элементов класса byte, содержащихсяв файле path
WriteAllText (string path, string contents)	Создает новый файл path, записывает в него указанную строку contents и затем закрывает файл. Если файл уже существует, он будет перезаписан
WriteAllLines (string path, string**[]** contents)	Создает новый файл path, записывает в него массив строк contents и затем закрывает файл. Если файл уже существует, он будет перезаписан
WriteAllBytes (string path, byte**[]** bytes)	Создает новый файл path, записывает в него массив bytes и затем закрывает файл. Если файл уже существует, он будет перезаписан

Выполнение операций осуществляется через класс, например:

```
File.Copy(@"C:\Temp\1.txt",@"C:\Temp\2.txt");
File.Delete(@"C:\Temp\1.txt");
```

8.2 Работа с файлами как с потоками

Рассматривая выше класс File можно обратить внимание, что методы открытия или создания файла возвращают объекты других классов через которые осуществляется работа с данными, находящимися в файле. Все эти объекты представляют собой потоки, работающие по единому принципу и рассматривающие файл как последовательность однотипных элементов, которыми можно манипулировать. Каждый поток знает свою длину и текущую позицию (в элементах потока), а также обеспечивает операции чтения и записи элементов в поток, закрытие потока. При каждой операции или записи текущая позиция в потоке меняется.

При работе с потоками может использоваться ключевое слово using, обеспечивающее по окончании работы с потоком его закрытие. Его использование имеет следующую формальную структуру:

```
using (<описание переменной и создание потока>)
{
  <использование переменной для работы с потоком>
}
```

Далее будут рассмотрены некоторые классы, обеспечивающие работу с файлами через потоки.

8.2.1 Класс FileStream

Класс `FileStream` применяется для операций чтения и записи в файл, открытия и закрытия файлов в файловой системе. При этом, любой файл рассматривается как последовательность байт, т.е. не учитывается, содержит ли файл текст в некоторой кодировке или двоичный код.

Класс расположен в пространстве имён `System.IO`.

Некоторые методы класса `FileStream` приведены в таблице 8.3.

Таблица 8.3 – Некоторые методы класса `FileStream`

Наименование	Описание
`FileStream` `(string path, FileMode mode [,FileAccess access [,FileShare share]])`	Создание потока на основе файла `path` с указанными параметрами `mode`, `access`, `share`
`Close``()`	Закрывает существующий поток
`Read` `(byte[] array, int offset, int count)`	Считывает из потока `count` или оставшееся количество байт, и размещает их в `array` начиная с позиции `offset`. Возвращает реально считанное количество байт
`ReadByte``()`	Считывает один байт из потока и преобразует его к типу `int`. Если считывание не произошло, то возвращается -1
`Write` `(byte[] array, int offset, int count)`	Записывает в поток `count` байт. Байты берутся из `array` начиная с позиции `offset`. Если значения `count` или `offset` указаны неверно (т.е. произойдет выход за границы массива), то запись не производится
`WriteByte` `(byte value)`	Записывает один байт `value` в поток
`Seek` `(long offset, SeekOrigin origin)`	Перемещает текущую позицию в потоке на `offset` байт относительно опорной точки, заданной в `origin`. Опорная точка может быть: • `SeekOrigin.Begin` – начало потока; • `SeekOrigin.End` – конец потока; • `SeekOrigin.Current` – текущее положение в потоке. Возвращается текущая позиция в потоке
`SetLength` `(long value)`	Устанавливает длину потока

Продолжение таблицы 8.3

Наименование	Описание
Length	Возвращает длину потока в байтах
Position	Получение или изменение текущей позиции в потоке
Name	Возвращает имя, переданное в конструктор потока (имя файла)

Пример: запись в файл чисел от 100 до 109 и последующее их чтение. Для преобразования между простыми типами и массивом байт используются методы класса `BitConverter`.

```
string path = "data.dat";
using (FileStream fs = File.Create(path))
{
  for (int i=100; i<110; i++)
  {
    byte[] info = BitConverter.GetBytes(i);
    fs.Write(info, 0, info.Length);
  }
}

using (FileStream fs = File.OpenRead(path))
{
  byte[] b = new byte[sizeof(int)];
  listBox1.Items.Clear();
  while (fs.Read(b, 0, b.Length) > 0)
  {
    int i = BitConverter.ToInt32(b,0);
    listBox1.Items.Add(i);
  }
}
```

8.2.2 Класс StreamReader

Класс `StreamReader` позволяет создавать объект, считывающий символы из потока в определённой кодировке, и применяемый для чтения строк из стандартного текстового файла.

Класс расположен в пространстве имён `System.IO`.

В таблице 8.4 приведены некоторые члены данного класса.

Таблица 8.4 – Некоторые члены класса `StreamReader`

Наименование	Описание
StreamReader (Stream stream [,Encoding encoding])	Создает экземпляр класса из указанного потока `stream`. Используется кодировка по умолчанию или `encoding`

Продолжение таблицы 8.4

Наименование	Описание
StreamReader (string path [,Encoding encoding])	Создает экземпляр класса из указанного имени файла path. Используется кодировка по умолчанию или encoding
Close()	Закрывает существующий поток
Peek()	Возвращает преобразованное к int значение следующего символа, или -1, если символов больше нет. **Позиция в потоке не меняется**
Read()	Считывает один символ из потока и преобразует его к типу int. Если считывание не произошло, то возвращается -1
Read (char[] buffer, int index, int count)	Считывает из потока count или оставшееся количество символов, и размещает их в buffer начиная с позиции index. Возвращает реально считанное количество символов
ReadBlock (char[] buffer, int index, int count)	Считывает из потока count или оставшееся количество символов, и размещает их в buffer начиная с позиции index. Возвращает реально считанное количество символов
ReadLine()	Считывает из потока следующую счтроку и возвращает ее. Если достигнут конец файла, то возвращается null
ReadToEnd()	Считывает из потока оставшиеся символы (начиная с текущей позиции) и возвращает их в виде строки. Если текущая позиция находится в конце файла, то возвращается пустая строка
EndOfStream	Возвращает true, если текущая позиция находится в конце файла, или false в противном случае
CurrentEncoding	Возвращает текущую кодировку в виде объекта класса Encoding

Пример: чтение информации о людях (человек характеризуется ФИО, серией и номер паспорта). Для хранения информации об одном человеке создан класс.

```
class TPeople
{
  public string FIO, Series;
  public int Number;
  public override string ToString()
  {
    return FIO;
  }
}

using (StreamReader sr = new StreamReader(File.Open("data.txt",
       FileMode.Open)))
```

```
{
  while (!sr.EndOfStream)
  {
    TPeople People = new TPeople();
    People.FIO = sr.ReadLine();
    People.Series = sr.ReadLine();
    People.Number = Convert.ToInt32(sr.ReadLine());
    listBox1.Items.Add(People);
  }
}
```

8.2.3 Класс StreamWriter

Класс `StreamWriter` позволяет создавать объект, записывающий символы в поток в определённой кодировке, и применяемый для записи строк в стандартный текстовый файл.

Класс расположен в пространстве имён `System.IO`.

В таблице 8.5 приведены некоторые члены данного класса.

Таблица 8.5 – Некоторые члены класса `StreamWriter`

Наименование	Описание
`StreamWriter` `(Stream stream [,Encoding encoding])`	Создает экземпляр класса и связывает его с потоком `stream`, в который будут выводится данные. Используется кодировка по умолчанию или `encoding`
`StreamWriter` `(string path [,Encoding encoding])`	Создает экземпляр класса и связывает его с файлом `path`, в который будут выводится данные. Используется кодировка по умолчанию или `encoding`
`Close``()`	Закрывает существующий поток
`Write``(<данные>)` **`WriteLine`**`(<данные>)`	Записывает в поток `<данные>`, которые могут быть переменными, константами, результатами выражений и т.д. Также поддерживается набор параметров, используемых методом `String.Format()`. `WriteLine` отличается от `Write` выводом символов конца строки и может быть использована без параметров
`Encoding`	Возвращает текущую кодировку в виде объекта класса `Encoding`

Пример: запись информации о людях (человек характеризуется ФИО, серией и номер паспорта). Для хранения информации об одном человеке создан класс.

```
class TPeople
{
  public string FIO, Series;
  public int Number;
```

```
  public override string ToString()
  {
    return FIO;
  }
}

TPeople[] Peoples = ???;
using (StreamWriter sw = new StreamWriter(File.Open("data.txt",
       FileMode.Create)))
{
  foreach (TPeople People in Peoples)
  {
    sw.WriteLine(People.FIO);
    sw.WriteLine(People.Series);
    sw.WriteLine(People.Number);
  }
}
```

8.2.4 Класс BinaryReader

Класс `BinaryReader` позволяет создавать объект, считывающий из потока простые типы данных как двоичные значения в определённой кодировке.

Класс расположен в пространстве имён `System.IO`.

В таблице 8.6 приведены некоторые члены данного класса.

Таблица 8.6 – Некоторые члены класса `BinaryReader`

Наименование	Описание
`BinaryReader` `(Stream input [,Encoding encoding])`	Создает экземпляр класса из указанного потока `input`. Используется кодировка по умолчанию или `encoding`
`Close``()`	Закрывает существующий поток
`PeekChar``()`	Возвращает преобразованное к `int` значение следующего символа, или -1, если символов больше нет. **Позиция в потоке не меняется**
`Read``()`	Считывает один символ из потока и преобразует его к типу `int`. Если считывание не произошло, то возвращается -1
`Read` `(char[] buffer, int index, int count)`	Считывает из потока `count` или оставшееся количество символов, и размещает их в `buffer` начиная с позиции `index`. Возвращает реально считанное количество символов

Продолжение таблицы 8.6

Наименование	Описание

Read<тип>()	Считывает знвачение типа <тип> и возвращает его, например:
	`using (BinaryReader br = ???)`
	`{`
	`  int i = br.ReadInt32();`
	`  string s = br.ReadString();`
	`  double d = br.ReadDouble();`
	`}`

ReadBytes
(int count)
	Считывает из потока count байт и помещает их в массив байт, например:
	`using (BinaryReader br = ???)`
	`{`
	`  byte[] b = br.ReadBytes(5);`
	`}`

ReadChars
(int count)
	Считывает из потока count символов и помещает их в массив символов, например:
	`using (BinaryReader br = ???)`
	`{`
	`  char[] c = br.ReadChars(5);`
	`}`

Пример: чтение информации о людях (человек характеризуется ФИО, серией и номер паспорта). Для хранения информации об одном человеке создан класс.

```
class TPeople
{
  public string FIO, Series;
  public int Number;
  public override string ToString()
  {
    return FIO;
  }
}

using (BinaryReader br = new BinaryReader(File.Open("data.txt",
       FileMode.Open)))
{
  while (br.PeekChar() != -1)
  {
    TPeople People = new TPeople();
    People.FIO = br.ReadString();
    People.Series = br.ReadString();
    People.Number = br.ReadInt32();
    listBox1.Items.Add(People);
  }
}
```

8.2.5 Класс BinaryWriter

Класс `BinaryWriter` позволяет создавать объект, записывающий в поток простые типы данных как двоичные значения в определенной кодировке.

Класс расположен в пространстве имен `System.IO`.

В таблице 8.7 приведены некоторые члены данного класса.

Таблица 8.7 – Некоторые члены класса `BinaryWriter`

Наименование	Описание
`BinaryWriter` `(Stream output [,Encoding encoding])`	Создает экземпляр класса и связывает его с потоком `output`, в который будут выводится данные. Используется кодировка по умолчанию или `encoding`
`Close``()`	Закрывает существующий поток
`Write``(<данные>)`	Записывает в поток `<данные>`, которые могут быть переменными, константами, результатами выражений и т.д.
`Write` `(byte[] buffer, int index, int count)`	Записывает в поток `count` байт из иассива `buffer` начиная с позиции `index`
`Write` `(char[] chars, int index, int count)`	Записывает в поток `count` символов из иассива `chars` начиная с позиции `index`
`Seek` `(long offset, SeekOrigin origin)`	Перемещает текущую позицию в потоке на `offset` байт относительно опорной точки, заданной в `origin`. Опорная точка может быть: • `SeekOrigin.Begin` – начало потока; • `SeekOrigin.End` – конец потока; • `SeekOrigin.Current` – текущее положение в потоке. Возвращается текущая позиция в потоке

Пример: запись информации о людях (человек характеризуется ФИО, серией и номер паспорта). Для хранения информации об одном человеке создан класс.

```
class TPeople
{
  public string FIO, Series;
  public int Number;
  public override string ToString()
  {
    return FIO;
  }
}
```

```
TPeople[] Peoples = ???;
using (BinaryWriter bw = new BinaryWriter(File.Open("data.txt",
       FileMode.Create)))
{
  foreach (TPeople People in Peoples)
  {
    bw.Write(People.FIO);
    bw.Write(People.Series);
    bw.Write(People.Number);
  }
}
```