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
Java1.

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.

Рисунок 1.1 – Структура типов данных

Рисунок 1.1 – Структура типов данных

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.

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

Наименование	Размер, байт	Диапазон значений
byte	1	0…255
sbyte	1	–128 … 127
Char1	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 …
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.

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 которой1 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 бло-
ка =<значение идентификатора N>.

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.

Водной 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 классахпотомках. 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 воз вращает 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 осно ванию 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	double
Sign(X)	знак числа X: -1, если число меньше нуля; 0, если число равно нулю; 1, если число больше нуля	int
`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	`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 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;
```

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

```
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
```