



По соглашению обобщенные типы и методы с *единственным* параметром типа обычно именуют его как `T`, если назначение параметра очевидно. В случае *нескольких* параметров типа каждый такой параметр имеет более описательное имя (с префиксом `T`).

Операция `typeof` и несвязанные обобщенные типы

Во время выполнения открытых обобщенных типов не существует: они закрываются на этапе компиляции. Однако во время выполнения возможно существование *несвязанного* (*unbound*) обобщенного типа — исключительно как объекта `Type`. Единственным способом указания несвязанного обобщенного типа в C# является применение операции `typeof`:

```
class A<T> {}  
class A<T1,T2> {}  
...  
Type a1 = typeof (A<>);           // Несвязанный тип (обратите внимание  
                                     // на отсутствие аргументов типа)  
Type a2 = typeof (A<,>);          // При указании нескольких аргументов  
                                     // типа используются запятые
```

Открытые обобщенные типы применяются в сочетании с API-интерфейсом рефлексии (см. главу 18).

Операцию `typeof` можно также использовать для указания закрытого типа:

```
Type a3 = typeof (A<int,int>);
```

или открытого типа (который закроется во время выполнения):

```
class B<T> { void X() { Type t = typeof (T); } }
```

Стандартное значение для параметра обобщенного типа

Чтобы получить стандартное значение для параметра обобщенного типа, можно применить ключевое слово `default`. Стандартным значением для ссылочного типа является `null`, а для типа значения — результат побитового обнуления полей в этом типе:

```
static void Zap<T> (T[] array)  
{  
    for (int i = 0; i < array.Length; i++)  
        array[i] = default(T);  
}
```

Начиная с версии C# 7.1, аргумент типа можно опускать в случаях, когда компилятор способен вывести его. Последнюю строку кода можно было бы заменить такой строкой:

```
array[i] = default;
```

Ограничения обобщений

По умолчанию параметр типа может быть замещен любым типом. Чтобы требовать более специфичные аргументы типа, к параметру типа можно применить *ограничения*. Ниже перечислены возможные ограничения:

```
where T : базовый-класс      // Ограничение базового класса
where T : интерфейс          // Ограничение интерфейса
where T : class               // Ограничение ссылочного типа
where T : class? // (См. раздел "Ссылочные типы, допускающие null" в главе 4)
where T : struct              // Ограничение типа значения (исключает типы,
                             // допускающие null)
where T : unmanaged           // Ограничение неуправляемого кода
where T : new()                // Ограничение конструктора без параметров
where U : T                   // Неприкрытое ограничение типа
where T : notnull              // Тип значения, не допускающий null, или
                             // ссылочный тип, не допускающий null (начиная с версии C# 8)
```

В следующем примере `GenericClass<T, U>` требует, чтобы тип `T` был производным от класса `SomeClass` (либо идентичен ему) и реализовывал интерфейс `Interface1`, а тип `U` предоставлял конструктор без параметров:

```
class     SomeClass {}
interface Interface1 {}

class GenericClass<T,U> where T : SomeClass, Interface1
                           where U : new()
{...}
```

Ограничения можно применять везде, где определены параметры типа, как в методах, так и в определениях типов.



Ограничение обеспечивает ограниченность; тем не менее, основной целью ограничений параметров типа является разрешение действий, которые в противном случае были бы запрещены.

Например, ограничение `T: Foo` позволяет обрабатывать экземпляры `T` как `Foo`, а ограничение `T: new()` разрешает создавать новые экземпляры `T`.

Ограничение базового класса указывает, что параметр типа должен быть подклассом заданного класса (или совпадать с ним); *ограничение интерфейса* указывает, что параметр типа должен реализовывать этот интерфейс. Такие ограничения позволяют экземплярам параметра типа быть неявно преобразуемыми в указанный класс или интерфейс. Например, пусть необходимо написать обобщенный метод `Max`, который возвращает большее из двух значений. Мы можем задействовать обобщенный интерфейс `IComparable<T>`, определенный в пространстве имен `System`:

```
public interface IComparable<T> // Упрощенная версия интерфейса
{
    int CompareTo (T other);
}
```

Метод CompareTo возвращает положительное число, если this больше other. Применяя данный интерфейс в качестве ограничения, мы можем написать метод Max следующим образом (чтобы не отвлекать внимания, проверка на null опущена):

```
static T Max <T> (T a, T b) where T : IComparable<T>
{
    return a.CompareTo (b) > 0 ? a : b;
}
```

Метод Max может принимать аргументы любого типа, реализующего интерфейс IComparable<T> (что включает большинство встроенных типов, таких как int и string):

```
int z = Max (5, 10);                                // 10
string last = Max ("ant", "zoo");                  // zoo
```



Начиная с версии C# 11, ограничение интерфейса также позволяет получать доступ к статическим виртуальным/абстрактным членам этого интерфейса (см. раздел “Статические виртуальные/абстрактные члены интерфейсов” в главе 1). Например, если интерфейс IFoo определяет статический абстрактный метод по имени Bar, тогда ограничение T:IFoo делает допустимым вызов T.Bar(). В разделе “Статический полиморфизм” главы 4 мы продолжим данную тему.

Ограничение class и *ограничение struct* указывают, что T должен быть ссылочным типом или типом значения (не допускающим null). Хорошим примером ограничения struct является структура System.Nullable<T> (мы обсудим этот тип в разделе “Типы значений, допускающие null” главы 4):

```
struct Nullable<T> where T : struct {...}
```

Ограничение неуправляемого кода (появившееся в версии C# 7.3) представляет собой более строгую версию ограничения struct: тип T обязан быть простым типом значения или структурой, которая (рекурсивно) свободна от любых ссылочных типов.

Ограничение конструктора без параметров требует, чтобы тип T имел открытый конструктор без параметров и позволял вызывать операцию new на T:

```
static void Initialize<T> (T[] array) where T : new()
{
    for (int i = 0; i < array.Length; i++)
        array[i] = new T();
}
```

Неприкрытое ограничение типа требует, чтобы один параметр типа был производным от другого параметра типа (или совпадал с ним). В следующем примере метод FilteredStack возвращает другой экземпляр Stack, содержащий только подмножество элементов, в которых параметр типа U является параметром типа T:

```
class Stack<T>
{
    Stack<U> FilteredStack<U>() where U : T {...}
}
```

Создание подклассов для обобщенных типов

Подклассы для обобщенного класса можно создавать точно так же, как в случае необобщенного класса. Подкласс может оставлять параметры типа базового класса открытыми, как показано в следующем примере:

```
class Stack<T>          {...}
class SpecialStack<T> : Stack<T>  {...}
```

Либо же подкласс может закрывать параметры обобщенного типа посредством конкретного типа:

```
class IntStack : Stack<int>      {...}
```

Подкласс может также вводить новые аргументы типа:

```
class List<T>          {...}
class KeyedList<T, TKey> : List<T> {...}
```



Формально *все* аргументы типа в подтипе являются свежими: можно сказать, что подтип закрывает и затем повторно открывает аргументы базового типа. Это значит, что подкласс может назначать аргументам типа новые (и потенциально более осмысленные) имена, когда повторно их открывает:

```
class List<T> {...}
class KeyedList<TElement, TKey> : List<TElement> {...}
```

Самоссылающиеся объявления обобщений

При закрытии аргумента типа тип может указывать *самого себя* в качестве конкретного типа:

```
public interface IEquatable<T> { bool Equals (T obj); }
public class Balloon : IEquatable<Balloon>
{
    public string Color { get; set; }
    public int CC { get; set; }

    public bool Equals (Balloon b)
    {
        if (b == null) return false;
        return b.Color == Color && b.CC == CC;
    }
}
```

Следующий код также допустим:

```
class Foo<T> where T : IComparable<T> { ... }
class Bar<T> where T : Bar<T> { ... }
```

Статические данные

Статические данные уникальны для каждого закрытого типа:

```
Console.WriteLine (++Bob<int>.Count);           // 1
Console.WriteLine (++Bob<int>.Count);           // 2
Console.WriteLine (++Bob<string>.Count);         // 1
Console.WriteLine (++Bob<object>.Count);         // 1

class Bob<T> { public static int Count; }
```

Параметры типа и преобразования

Операция приведения в C# может выполнять преобразования нескольких видов, включая:

- числовое преобразование;
- ссылочное преобразование;
- упаковывающее/распаковывающее преобразование;
- специальное преобразование (через перегрузку операций; см. главу 4).

Решение о том, какой вид преобразования произойдет, принимается на этапе компиляции, базируясь на известных типах операндов. Это создает интересный сценарий с параметрами обобщенного типа, т.к. точные типы операндов на этапе компиляции не известны. Если возникает неоднозначность, тогда компилятор генерирует сообщение об ошибке.

Наиболее распространенный сценарий связан с выполнением ссылочного преобразования:

```
StringBuilder Foo<T> (T arg)
{
    if (arg is StringBuilder)
        return (StringBuilder) arg;           // Не скомпилируется
    ...
}
```

Без знания фактического типа T компилятор предполагает, что вы намереваетесь выполнить *специальное преобразование*. Простейшим решением будет использование взамен операции as, которая не дает неоднозначности, т.к. не позволяет осуществлять специальные преобразования:

```
StringBuilder Foo<T> (T arg)
{
    StringBuilder sb = arg as StringBuilder;
    if (sb != null) return sb;
    ...
}
```

Более общее решение предусматривает приведение сначала к object. Такой подход работает, поскольку предполагается, что преобразования в/из object должны быть не специальными, а ссылочными или упаковывающими/распаковывающими. В данном случае StringBuilder является ссылочным типом, поэтому должно происходить ссылочное преобразование:

```
return (StringBuilder) (object) arg;
```

Распаковывающие преобразования также способны привносить неоднозначность. Показанное ниже преобразование может быть распаковывающим, числовым или специальным:

```
int Foo<T> (T x) => (int) x; // Ошибка на этапе компиляции
```

И снова решение заключается в том, чтобы сначала выполнить приведение к `object`, а затем к `int` (которое в данном случае однозначно сигнализирует о распаковывающем преобразовании):

```
int Foo<T> (T x) => (int) (object) x;
```

Ковариантность

Исходя из предположения, что тип А допускает преобразование в В, тип X имеет ковариантный параметр типа, если `X<A>` поддается преобразованию в `X`.



Согласно понятию ковариантности (и контравариантности) в языке C# формулировка “поддается преобразованию” означает возможность преобразования через *неявное ссылочное преобразование*, такое как *A является подклассом B* или *A реализует B*. Сюда не входят числовые преобразования, упаковывающие преобразования и специальные преобразования.

Например, тип `IFoo<T>` имеет ковариантный тип T, если справедливо следующее:

```
IFoo<string> s = ...;  
IFoo<object> b = s;
```

Интерфейсы допускают ковариантные параметры типа (как это делают делегаты; см. главу 4), но классы — нет. Массивы также разрешают ковариантность (массив `A[]` может быть преобразован в `B[]`, если для A имеется ссылочное преобразование в B) и обсуждаются здесь ради сравнения.



Ковариантность и контравариантность (или просто “вариантность”) — сложные концепции. Мотивация, лежащая в основе введения и расширения вариантиности в C#, заключалась в том, чтобы позволить обобщенным интерфейсам и обобщенным типам (в частности, определенным в .NET, таким как `IEnumerable<T>`) работать более предсказуемым образом. Даже не понимая все детали ковариантности и контравариантности, вы можете извлечь из них выгоду.

Вариантность не является автоматической

Чтобы обеспечить статическую безопасность типов, параметры типа не определяются как вариантные автоматически. Рассмотрим приведенный ниже код:

```
class Animal {}  
class Bear : Animal {}  
class Camel : Animal {}
```

```
public class Stack<T> // Простая реализация стека
{
    int position;
    T[] data = new T[100];
    public void Push (T obj) => data[position++] = obj;
    public T Pop() => data[--position];
}
```

Следующий код не скомпилируется:

```
Stack<Bear> bears = new Stack<Bear>();
Stack<Animal> animals = bears; // Ошибка на этапе компиляции
```

Это ограничение предотвращает возможность возникновения ошибки во время выполнения из-за такого кода:

```
animals.Push (new Camel()); // Попытка добавить объект Camel в bears
```

Тем не менее, отсутствие ковариантности может послужить препятствием повторному использованию. Предположим, что требуется написать метод Wash (мыть) для стека объектов, представляющих животных:

```
public class ZooCleaner
{
    public static void Wash (Stack<Animal> animals) {...}
}
```

Вызов метода Wash со стеком объектов представляющих медведей (bear), приведет к генерации ошибки на этапе компиляции. Один из обходных путей предполагает переопределение метода Wash с ограничением:

```
class ZooCleaner
{
    public static void Wash<T> (Stack<T> animals) where T : Animal { ... }
```

Теперь метод Wash можно вызывать следующим образом:

```
Stack<Bear> bears = new Stack<Bear>();
ZooCleaner.Wash (bears);
```

Другое решение состоит в том, чтобы обеспечить реализацию классом Stack<T> интерфейса с ковариантным параметром типа, как вскоре будет показано.

Массивы

По историческим причинам типы массивов поддерживают ковариантность. Это значит, что массив B[] может быть приведен к A[], если B является подклассом A (и оба они являются ссылочными типами). Например:

```
Bear[] bears = new Bear[3];
Animal[] animals = bears; // Нормально
```

Недостаток такой возможности повторного использования заключается в том, что присваивание элементов может потерпеть неудачу во время выполнения:

```
animals[0] = new Camel(); // Ошибка во время выполнения
```

Объявление ковариантного параметра типа

Параметры типа в интерфейсах и делегатах могут быть объявлены как ковариантные путем их пометки с помощью модификатора `out`. Данный модификатор гарантирует, что в отличие от массивов ковариантные параметры типа будут полностью безопасными в отношении типов.

Мы можем проиллюстрировать сказанное на классе `Stack`, обеспечив реализацию им следующего интерфейса:

```
public interface IPoppable<out T> { T Pop(); }
```

Модификатор `out` для `T` указывает, что тип `T` применяется только в *выходных позициях* (например, в возвращаемых типах для методов). Модификатор `out` помечает параметр типа как *ковариантный* и разрешает написание такого кода:

```
var bears = new Stack<Bear>();
bears.Push (new Bear());
//bears реализует IPoppable<Bear>. Мы можем преобразовать bears в IPoppable<Animal>:
IPoppable<Animal> animals = bears; // Допустимо
Animal a = animals.Pop();
```

Преобразование `bears` в `animals` разрешено компилятором в силу того, что параметр типа является ковариантным. Преобразование безопасно в отношении типов, т.к. сценарий, от которого компилятор пытается уклониться (помещение объекта `Camel` в стек), возникнуть не может, поскольку нет способа передать `Camel` в интерфейс, где `T` может встречаться только в *выходных позициях*.



Ковариантность (и контравариантность) в интерфейсах — это то, что обычно *потребляется*: необходимость в *написании* вариантовых интерфейсов возникает реже.



Любопытно, что параметры метода, помеченные как `out`, не подходят для ковариантности из-за ограничения в среде CLR.

Возможность ковариантного приведения можно задействовать для решения описанной ранее проблемы повторного использования:

```
public class ZooCleaner
{
    public static void Wash (IPoppable<Animal> animals) { ... }
}
```



Интерфейсы `IEnumerable<T>` и `IEnumerator<T>`, описанные в главе 7, имеют ковариантный параметр типа `T`, что позволяет приводить `IEnumerable<string>`, например, к `IEnumerable<object>`.

Компилятор генерирует ошибку, если ковариантный параметр типа применяется во *входной* позиции (скажем, в параметре метода или в записываемом свойстве).



Ковариантность (и контравариантность) работают только для элементов со *ссылочными* — не *упаковывающими* — преобразованиями. (Это применимо как к вариантности параметров типа, так и к вариантности массивов.) Таким образом, если имеется метод, который принимает параметр типа `IPoppable<object>`, то его можно вызывать с `IPoppable<string>`, но не с `IPoppable<int>`.

Контравариантность

Как было показано ранее, если предположить, что A разрешает неявное ссылочное преобразование в B, то тип X имеет ковариантный параметр типа, когда `X<A>` допускает ссылочное преобразование в `X`. Контравариантность существует в случае, если возможно преобразование в обратном направлении — из `X` в `X<A>`. Это поддерживается, когда параметр типа встречается только во *входных* позициях, и обозначается с помощью модификатора `in`. Продолжая предыдущий пример, пусть класс `Stack<T>` реализует следующий интерфейс:

```
public interface IPushable<in T> { void Push (T obj); }
```

Теперь вполне законно поступать так:

```
IPushable<Animal> animals = new Stack<Animal>();  
IPushable<Bear> bears = animals; // Допустимо  
bears.Push (new Bear());
```

Ни один из членов `IPushable` не содержит тип `T` в *выходной* позиции, а потому никаких проблем с приведением `animals` к `bears` не возникает (например, данный интерфейс не поддерживает метод `Pop`).



Класс `Stack<T>` может реализовывать оба интерфейса, `IPushable<T>` и `IPoppable<T>`, несмотря на то, что тип `T` в указанных двух интерфейсах имеет противоположные модификаторы вариантности! Причина в том, что вариантность должна использоваться через интерфейс, а не через класс; следовательно, перед выполнением вариантного преобразования его потребуется пропустить сквозь призму либо `IPoppable`, либо `IPushable`. В результате вы будете ограничены только операциями, которые допускаются соответствующими правилами вариантности.

Это также показывает, почему *классам* не позволено иметь вариантные параметры типа: конкретные реализации обычно требуют про текания данных в обоих направлениях.

Для другого примера необходим следующий интерфейс, который определен в пространстве имен `System`:

```
public interface IComparer<in T>  
{  
    // Возвращает значение, отражающее относительный порядок a и b  
    int Compare (T a, T b);  
}
```

Поскольку интерфейс имеет контравариантный параметр типа *T*, мы можем применять *IComparer<object>* для сравнения двух строк:

```
var objectComparer = Comparer<object>.Default;
// objectComparer реализует IComparer<object>
IComparer<string> stringComparer = objectComparer;
int result = stringComparer.Compare ("Brett", "Jemaine");
```

Зеркально отражая ковариантность, компилятор сообщит об ошибке, если вы попытаетесь использовать контравариантный параметр типа в выходной позиции (скажем, в качестве возвращаемого значения или в читаемом свойстве).

Сравнение обобщений C# и шаблонов C++

Обобщения C# в использовании похожи на шаблоны C++, но работают они совершенно по-другому. В обоих случаях должен осуществляться синтез между поставщиком и потребителем, при котором типы-заполнители заполняются потребителем. Однако в ситуации с обобщениями C# типы поставщика (т.е. открытые типы вроде *List<T>*) могут быть скомпилированы в библиотеку (такую как *mscorlib.dll*). Дело в том, что собственно синтез между поставщиком и потребителем, который создает закрытые типы, в действительности не происходит вплоть до времени выполнения. Для шаблонов C++ такой синтез производится на этапе компиляции. Это значит, что в C++ развертывать библиотеки шаблонов как сборки .dll не получится — они существуют только в виде исходного кода. Вдобавок также затрудняется динамическое инспектирование параметризованных типов, не говоря уже об их создании на лету.

Чтобы лучше понять, почему сказанное справедливо, давайте взглянем на метод *Max* в C#:

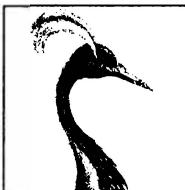
```
static T Max <T> (T a, T b) where T : IComparable<T>
=> a.CompareTo (b) > 0 ? a : b;
```

Почему бы ни реализовать метод *Max* следующим образом:

```
static T Max <T> (T a, T b)
=> (a > b ? a : b); // Ошибка на этапе компиляции
```

Причина в том, что метод *Max* должен быть скомпилирован один раз, но работать для всех возможных значений *T*. Компиляция не может пройти успешно ввиду отсутствия единого смысла операции *>* для всех значений *T* — на самом деле операция *>* может быть доступна далеко не в каждом типе *T*. Для сравнения ниже показан код того же метода *Max*, написанный с применением шаблонов C++. Такой код будет компилироваться отдельно для каждого значения *T*, пользуясь семантикой *>* для конкретного типа *T* и приводя к ошибке на этапе компиляции, если отдельный тип *T* не поддерживает операцию *>*:

```
template <class T> T Max (T a, T b)
{
    return a > b ? a : b;
}
```

Дополнительные средства языка C#

В настоящей главе будут раскрыты более сложные аспекты языка C#, которые основаны на концепциях, исследованных в главах 2 и 3. Первые четыре раздела необходимо читать последовательно, а остальные — в произвольном порядке.

Делегаты

Делегат — это объект, которому известно, как вызывать некий метод.

Тип делегата определяет вид метода, который могут вызывать экземпляры делегата. В частности он определяет *возвращаемый тип* и *типы параметров* метода. Ниже показано определение типа делегата по имени *Transformer*:

```
delegate int Transformer (int x);
```

Делегат *Transformer* совместим с любым методом, который имеет возвращаемый тип *int* и принимает единственный параметр *int*, вроде следующего:

```
int Square (int x) { return x * x; }
```

или более скжато:

```
int Square (int x) => x * x;
```

Присваивание метода переменной делегата создает экземпляр делегата:

```
Transformer t = Square;
```

Экземпляр делегата можно вызывать тем же способом, что и метод:

```
int answer = t(3); // answer получает значение 9
```

Вот завершенный пример:

```
Transformer t = Square; // Создание экземпляра делегата
int result = t(3); // Вызов делегата
Console.WriteLine (result); // 9

int Square (int x) => x * x;
delegate int Transformer (int x); // Объявление типа делегата
```

Экземпляр делегата действует в вызывающем компоненте буквально как посредник: вызывающий компонент обращается к делегату, после чего делегат вызывает целевой метод. Такая косвенность отвязывает вызывающий компонент от целевого метода.

Оператор:

```
Transformer t = Square;
```

является сокращением следующего оператора:

```
Transformer t = new Transformer (Square);
```



Формально когда мы ссылаемся на `Square` без скобок или аргументов, то указываем *группу методов*. Если метод перегружен, тогда компилятор C# выберет корректную перегруженную версию на основе сигнатуры делегата, которому `Square` присваивается.

Выражение:

```
t (3)
```

представляет собой сокращение такого вызова:

```
t.Invoke(3)
```



Делегат похож на *обратный вызов* — общий термин, который охватывает конструкции вроде указателей на функции С.

Написание подключаемых методов с помощью делегатов

Метод присваивается переменной делегата во время выполнения, что удобно при написании подключаемых методов. В следующем примере присутствует служебный метод по имени `Transform`, который применяет трансформацию к каждому элементу в целочисленном массиве. Метод `Transform` имеет параметр делегата, предназначенный для указания подключаемой трансформации:

```
int[] values = { 1, 2, 3 };
Transform (values, Square); // Привязаться к методу Square
foreach (int i in values)
    Console.Write (i + " "); // 1 4 9
void Transform (int[] values, Transformer t)
{
    for (int i = 0; i < values.Length; i++)
        values[i] = t (values[i]);
}
int Square (int x) => x * x;
int Cube (int x) => x * x * x;
delegate int Transformer (int x);
```

Мы можем изменить трансформацию, просто поменяв `Square` на `Cube` во второй строке кода.

Наш метод `Transform` является *функцией более высокого порядка*, потому что получает в качестве аргумента функцию. (Метод, который *возвращает* делегат, тоже будет функцией более высокого порядка.)

Целевые методы экземпляра и статические целевые методы

Целевой метод делегата может быть локальным, статическим или методом экземпляра. Ниже демонстрируется использование статического целевого метода:

```
Transformer t = Test.Square;
Console.WriteLine (t(10));                                // 100
class Test { public static int Square (int x) => x * x; }
delegate int Transformer (int x);
```

А так применяется целевой метод экземпляра:

```
Test test = new Test();
Transformer t = test.Square;
Console.WriteLine (t(10));                                // 100
class Test { public int Square (int x) => x * x; }
delegate int Transformer (int x);
```

Когда объекту делегата присваивается метод экземпляра, объект делегата поддерживает ссылку не только на данный метод, но также на экземпляр, которому метод принадлежит. Этот экземпляр представляет свойство Target класса System.Delegate (которое будет иметь значение null для делегата, ссылающегося на статический метод). Вот пример:

```
MyReporter r = new MyReporter();
r.Prefix = "%Complete: ";
ProgressReporter p = r.ReportProgress;
p(99);                                                 // %Complete: 99
Console.WriteLine (p.Target == r);                      // True
Console.WriteLine (p.Method);                           // void ReportProgress(Int32)
r.Prefix = "";
p(99);                                                 // 99
public delegate void ProgressReporter (int percentComplete);
class MyReporter
{
    public string Prefix = "";
    public void ReportProgress (int percentComplete)
        => Console.WriteLine (Prefix + percentComplete);
}
```

Поскольку экземпляр сохраняется в свойстве Target делегата, его время жизни расширяется (по крайней мере) до времени жизни самого делегата.

Групповые делегаты

Все экземпляры делегатов обладают возможностью *группового вызова* (multicast), т.е. экземпляр делегата может ссылаться не только на одиничный целевой метод, но также и на список целевых методов. Экземпляры делегатов комбинируются с помощью операций + и +=:

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

Последняя строка функционально эквивалентна следующей строке:

```
d = d + SomeMethod2;
```

Обращение к d теперь приведет к вызову методов SomeMethod1 и SomeMethod2. Делегаты вызываются в порядке, в котором они добавлялись.

Операции - и -- удаляют правый операнд делегата из левого операнда делегата. Например:

```
d -= SomeMethod1;
```

Обращение к d теперь приведет к вызову только метода SomeMethod2.

Использование операции + или += с переменной делегата, имеющей значение null, допустимо и эквивалентно присваиванию этой переменной нового значения:

```
SomeDelegate d = null;
```

```
d += SomeMethod1; //Когда d равно null, эквивалентно оператору d = SomeMethod1;
```

Аналогичным образом применение операции -= к переменной делегата с единственным целевым методом эквивалентно присваиванию этой переменной значения null.



Делегаты являются неизменяемыми, так что в случае использования операции += или -= фактически создается новый экземпляр делегата и присваивается существующей переменной.

Если групповой делегат имеет возвращаемый тип, отличающийся от void, тогда вызывающий компонент получает возвращаемое значение из последнего вызванного метода. Предшествующие методы по-прежнему вызываются, но их возвращаемые значения отбрасываются. В большинстве сценариев применения групповые делегаты имеют возвращаемые типы void, так что описанная тонкая ситуация не возникает.



Все типы делегатов неявно порождены от класса System.MulticastDelegate, который унаследован от System.Delegate. Операции +, -, += и -=, выполняемые над делегатом, транслируются в вызовы статических методов Combine и Remove класса System.Delegate.

Пример группового делегата

Предположим, что вы написали метод, выполнение которого занимает длительное время. Такой метод мог бы регулярно сообщать о ходе работ вызывающему компоненту, обращаясь к делегату. В следующем примере метод HardWork имеет параметр делегата ProgressReporter, который вызывается для отражения хода работ:

```
public delegate void ProgressReporter (int percentComplete);
public class Util
{
```

```

public static void HardWork (ProgressReporter p)
{
    for (int i = 0; i < 10; i++)
    {
        p (i * 10);                                // Вызвать делегат
        System.Threading.Thread.Sleep (100); // Эмулировать длительную работу
    }
}
}

```

Для мониторинга хода работ метод Main создает экземпляр группового делегата p, так что ход работ отслеживается двумя независимыми методами:

```

ProgressReporter p = WriteProgressToConsole;
p += WriteProgressToFile;
Util.HardWork (p);

void WriteProgressToConsole (int percentComplete)
    => Console.WriteLine (percentComplete);

void WriteProgressToFile (int percentComplete)
    => System.IO.File.WriteAllText ("progress.txt",
                                    percentComplete.ToString ());

```

Обобщенные типы делегатов

Тип делегата может содержать параметры обобщенного типа:

```
public delegate T Transformer<T> (T arg);
```

Располагая таким определением, можно написать обобщенный служебный метод Transform, который работает с любым типом:

```

int[] values = { 1, 2, 3 };
Util.Transform (values, Square); // Привязаться к методу Square
foreach (int i in values)
    Console.Write (i + " ");
                                // 1   4   9

int Square (int x) => x * x;

public class Util
{
    public static void Transform<T> (T[] values, Transformer<T> t)
    {
        for (int i = 0; i < values.Length; i++)
            values[i] = t (values[i]);
    }
}

```

Делегаты Func и Action

Благодаря обобщенным делегатам становится возможным написание небольшого набора типов делегатов, которые настолько универсальны, что способны работать с методами, имеющими любой возвращаемый тип и любое (разумное) количество аргументов. Такими делегатами являются Func и Action, определенные в пространстве имен System (модификаторы in и out указывают *вариантность*, которая вскоре будет раскрыта в контексте делегатов):

```

delegate TResult Func <out TResult>();  

delegate TResult Func <in T, out TResult> (T arg);  

delegate TResult Func <in T1, in T2, out TResult> (T1 arg1, T2 arg2);  

... и так далее вплоть до T16  

delegate void Action ();  

delegate void Action <in T> (T arg);  

delegate void Action <in T1, in T2> (T1 arg1, T2 arg2);  

... и так далее вплоть до T16

```

Показанные делегаты исключительно универсальны. Делегат Transformer в предыдущем примере может быть заменен делегатом Func, который принимает один аргумент типа T и возвращает значение того же самого типа:

```

public static void Transform<T> (T[] values, Func<T, T> transformer)  

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

```

Делегаты Func и Action не покрывают лишь практические сценарии, связанные с параметрами ref/out и параметрами указателей.



Когда язык C# только появился, делегаты Func и Action отсутствовали (поскольку не было обобщений). Именно по указанной исторической причине в большей части .NET используются специальные типы делегатов, а не Func и Action.

Сравнение делегатов и интерфейсов

Задачу, которую можно решить с помощью делегата, вполне реально решить также посредством интерфейса. Мы можем переписать исходный пример, применяя вместо делегата интерфейс ITransformer:

```

int[] values = { 1, 2, 3 };
Util.TransformAll (values, new Squarer ());
foreach (int i in values)
    Console.WriteLine (i);
public interface ITransformer
{
    int Transform (int x);
}
public class Util
{
    public static void TransformAll (int[] values, ITransformer t)
    {
        for (int i = 0; i < values.Length; i++)
            values[i] = t.Transform (values[i]);
    }
}
class Squarer : ITransformer
{
    public int Transform (int x) => x * x;
}

```

Решение на основе делегатов может оказаться более удачным, чем решение на основе интерфейсов, если соблюдено одно или более следующих условий:

- в интерфейсе определен только один метод;
- требуется возможность группового вызова;
- подписчик нуждается в многократной реализации интерфейса.

В примере с `ITransformer` групповой вызов не нужен. Однако в интерфейсе определен только один метод. Более того, подписчику может потребоваться многократная реализация интерфейса `ITransformer` для поддержки разнообразных трансформаций наподобие возведения в квадрат или куб. В случае интерфейсов нам придется писать отдельный тип для каждой трансформации, т.к. класс может реализовывать `ITransformer` только один раз. В результате получается довольно громоздкий код:

```
int[] values = { 1, 2, 3 };
Util.TransformAll (values, new Cuber());
foreach (int i in values)
    Console.WriteLine (i);

class Squarer : ITransformer
{
    public int Transform (int x) => x * x;
}

class Cuber : ITransformer
{
    public int Transform (int x) => x * x * x;
}
```

Совместимость делегатов

Совместимость типов

Все типы делегатов несовместимы друг с другом, даже если они имеют одинаковые сигнатуры:

```
D1 d1 = Method1;
D2 d2 = d1;                                // Ошибка на этапе компиляции

void Method1() { }
delegate void D1();
delegate void D2();
```



Однако следующий код разрешен:

```
D2 d2 = new D2 (d1);
```

Экземпляры делегатов считаются равными, если они имеют одинаковые целевые методы:

```
D d1 = Method1;
D d2 = Method1;
Console.WriteLine (d1 == d2);      // True

void Method1() { }
delegate void D();
```

Групповые делегаты считаются равными, если они ссылаются на те же самые методы в одинаковом порядке.

Совместимость параметров

При вызове метода можно предоставлять аргументы, которые относятся к более специфичным типам, нежели те, что определены для параметров данного метода. Это обычное полиморфное поведение. По той же причине делегат может иметь более специфичные типы параметров, чем его целевой метод. Это называется *контравариантностью*. Вот пример:

```
StringAction sa = new StringAction (ActOnObject);
sa ("hello");
void ActOnObject (object o) => Console.WriteLine (o); // hello
delegate void StringAction (string s);
```

(Как и с вариантностью параметров типа, делегаты являются вариантными только для ссылочных преобразований.)

Делегат просто вызывает метод от имени кого-то другого. В таком случае `StringAction` вызывается с аргументом типа `string`. Когда аргумент затем передается целевому методу, он неявно приводится вверх к типу `object`.



Стандартный шаблон событий спроектирован для того, чтобы помочь задействовать контравариантность через использование общего базового класса `EventArgs`. Например, можно иметь единственный метод, который вызывается двумя разными делегатами с передачей одному объекта `MouseEventArgs`, а другому — `KeyEventArgs`.

Совместимость возвращаемых типов

В результате вызова метода можно получить обратно тип, который является более специфическим, чем запрошенный. Так выглядит обыкновенное полиморфное поведение. По той же самой причине целевой метод делегата может возвращать более специфический тип, чем описанный самим делегатом. Это называется *ковариантностью*:

```
ObjectRetriever o = new ObjectRetriever (RetrieveString);
object result = o();
Console.WriteLine (result); // hello
string RetrieveString() => "hello";
delegate object ObjectRetriever();
```

Делегат `ObjectRetriever` ожидает получить обратно `object`, но может быть получен также и подкласс `object`, потому что возвращаемые типы делегатов *ковариантны*.

Вариантность параметров типа обобщенного делегата

В главе 3 было показано, что обобщенные интерфейсы поддерживают ковариантные и контравариантные параметры типа. Та же самая возможность существует и для делегатов.

При определении обобщенного типа делегата рекомендуется поступать следующим образом:

- помечать параметр типа, применяемый только для возвращаемого значения, как ковариантный (*out*);
- помечать любой параметр типа, используемый только для параметров, как контравариантный (*in*).

В результате преобразования смогут работать естественным образом, соблюдая отношения наследования между типами.

Показанный ниже делегат (определенный в пространстве имен *System*) имеет ковариантный параметр *TResult*:

```
delegate void Action<in T> (T arg);
```

позволяя записывать так:

```
Func<string> x = ...;
Func<object> y = x;
```

Следующий делегат (определенный в пространстве имен *System*) имеет контравариантный параметр *T*:

```
delegate void Action<in T> (T arg);
```

делая возможным такой код:

```
Action<object> x = ...;
Action<string> y = x;
```

События

Во время применения делегатов обычно возникают две независимые роли: *ретранслятор* (*broadcaster*) и *подписчик* (*subscriber*).

Ретранслятор — это тип, который содержит поле делегата. Ретранслятор решает, когда делать передачу, вызывая делегат.

Подписчики — это целевые методы-получатели. Подписчик решает, когда начинать и останавливать прослушивание, используя операции *+=* и *-=* на делегате ретранслятора. Подписчик ничего не знает о других подписчиках и не вмешивается в их работу.

События являются языковым средством, которое формализует описанный шаблон. Конструкция *event* открывает только подмножество возможностей делегата, требуемое для модели “ретранслятор/подписчик”. Основное назначение событий заключается в *предотвращении влияния подписчиков друг на друга*.

Простейший способ объявления события предусматривает помещение ключевого слова *event* перед членом делегата:

```
// Определение делегата
public delegate void PriceChangedHandler (decimal oldPrice, decimal newPrice);
public class Broadcaster
{
    // Объявление события
    public event PriceChangedHandler PriceChanged;
}
```

Код внутри типа `Broadcaster` имеет полный доступ к члену `PriceChanged` и может трактовать его как делегат. Код за пределами `Broadcaster` может только выполнять операции `+=` и `-=` над событием `PriceChanged`.

Внутренняя работа событий

При объявлении показанного ниже события “за кулисами” происходят три действия:

```
public class Broadcaster
{
    public event PriceChangedHandler PriceChanged;
}
```

Во-первых, компилятор транслирует объявление события в примерно такой код:

```
PriceChangedHandler priceChanged; // закрытый делегат
public event PriceChangedHandler PriceChanged
{
    add { priceChanged += value; }
    remove { priceChanged -= value; }
}
```

Ключевыми словами `add` и `remove` обозначаются явные средства доступа к событию, которые работают аналогично средствам доступа к свойству. Позже мы покажем, как их реализовать.

Во-вторых, компилятор ищет *внутри* класса `Broadcaster` ссылки на `PriceChanged`, в которых выполняются операции, отличающиеся от `+=` или `-=`, и переадресует их на лежащее в основе поле делегата `priceChanged`.

В-третьих, компилятор транслирует операции `+=` и `-=`, примененные к событию, в вызовы средств доступа `add` и `remove` события. Интересно, что это делает поведение операций `+=` и `-=` уникальным в случае применения к событиям: в отличие от других сценариев они не являются просто сокращением для операций `+` и `-`, за которыми следует операция присваивания.

Рассмотрим следующий пример. Класс `Stock` запускает свое событие `PriceChanged` каждый раз, когда изменяется свойство `Price` данного класса:

```
public delegate void PriceChangedHandler (decimal oldPrice,
                                         decimal newPrice);

public class Stock
{
    string symbol;
    decimal price;

    public Stock (string symbol) => this.symbol = symbol;
    public event PriceChangedHandler PriceChanged;
    public decimal Price
    {
```

```

get => price;
set
{
    if (price == value) return;           // Выйти, если ничего не изменилось
    decimal oldPrice = price;
    price = value;

    if (PriceChanged != null)           // Если список вызова не пуст,
        PriceChanged (oldPrice, price); // тогда запустить событие
}
}
}

```

Если в приведенном примере убрать ключевое слово `event`, чтобы `PriceChanged` превратилось в обычное поле делегата, то результаты окажутся теми же самыми. Но класс `Stock` станет менее надежным до такой степени, что подписчики смогут предпринимать следующие действия, влияя друг на друга.

- Заменять других подписчиков, переустанавливая `PriceChanged` (вместо использования операции `+=`).
- Очищать всех подписчиков (устанавливая `PriceChanged` в `null`).
- Выполнять групповую рассылку другим подписчикам путем вызова делегата.

Стандартный шаблон событий

Почти во всех сценариях, для которых определяются события в библиотеках .NET, их определение придерживается стандартного шаблона, предназначенного для обеспечения согласованности между библиотекой и пользовательским кодом. В основе стандартного шаблона событий лежит `System.EventArgs` — предопределенный класс .NET, имеющий только статическое поле `Empty`. Базовый класс `EventArgs` позволяет передавать информацию событию. В рассматриваемом примере `Stock` мы создаем подкласс `EventArgs` для передачи старого и нового значений цены, когда инициируется событие `PriceChanged`:

```

public class PriceChangedEventArgs : System.EventArgs
{
    public readonly decimal LastPrice;
    public readonly decimal NewPrice;

    public PriceChangedEventArgs (decimal lastPrice, decimal newPrice)
    {
        LastPrice = lastPrice;
        NewPrice = newPrice;
    }
}

```

Ради многократного использования подкласс `EventArgs` именован в соответствии с содержащейся в нем информацией (а не с событием, для которого он будет применяться). Обычно он открывает доступ к данным как к свойствам или полям, предназначенным только для чтения.

Имея подкласс EventArgs, далее потребуется выбрать или определить делегат для события согласно следующим трем правилам.

- Он обязан иметь возвращаемый тип void.
- Он должен принимать два аргумента: первый — object и второй — подкласс EventArgs. Первый аргумент указывает ретранслятор события, а второй аргумент содержит дополнительную информацию для передачи событию.
- Его имя должно заканчиваться на EventHandler.

В .NET определен обобщенный делегат по имени System.EventHandler<>, который соответствует этому:

```
public delegate void EventHandler<TEventArgs>
    (object source, TEventArgs e) where TEventArgs : EventArgs;
```



До появления в языке обобщений (до выхода версии C# 2.0) взамен необходимо было записывать специальный делегат следующего вида:

```
public delegate void PriceChangedHandler
    (object sender, PriceChangedEventArgs e);
```

По историческим причинам большинство событий внутри библиотек .NET используют делегаты, определенные подобным образом.

Следующий шаг — определение события выбранного типа делегата. В приведенном ниже коде применяется обобщенный делегат EventHandler:

```
public class Stock
{
    ...
    public event EventHandler<PriceChangedEventArgs> PriceChanged;
}
```

В заключение шаблон требует написания защищенного виртуального метода, который запускает событие. Имя такого метода должно совпадать с именем события, предваренным словом On, и он должен принимать единственный аргумент EventArgs:

```
public class Stock
{
    ...
    public event EventHandler<PriceChangedEventArgs> PriceChanged;
    protected virtual void OnPriceChanged (PriceChangedEventArgs e)
    {
        if (PriceChanged != null) PriceChanged (this, e);
    }
}
```



Чтобы надежно работать в многопоточных сценариях (см. главу 14), делегат необходимо присваивать временной переменной перед его проверкой и вызовом:

```
var temp = PriceChanged;  
if (temp != null) temp (this, e);
```

Мы можем получить ту же самую функциональность и без переменной temp с помощью null-условной операции:

```
PriceChanged?.Invoke (this, e);
```

По причине безопасности к потокам и лаконичности это наилучший и общепринятый способ вызова событий.

В результате мы имеем центральную точку, из которой подклассы могут вызывать или переопределять событие (предполагая, что класс не запечатан).

Ниже приведен полный код примера:

```
using System;  
  
Stock stock = new Stock ("THPW");  
stock.Price = 27.10M;  
// Зарегистрировать с событием PriceChanged  
stock.PriceChanged += stock_PriceChanged;  
stock.Price = 31.59M;  
void stock_PriceChanged (object sender, PriceChangedEventArgs e)  
{  
    // Предупреждение об увеличении биржевого курса на 10%  
    if ((e.NewPrice - e.LastPrice) / e.LastPrice > 0.1M)  
        Console.WriteLine ("Alert, 10% stock price increase!");  
}  
public class PriceChangedEventArgs : EventArgs  
{  
    public readonly decimal LastPrice;  
    public readonly decimal NewPrice;  
    public PriceChangedEventArgs (decimal lastPrice, decimal newPrice)  
    {  
        LastPrice = lastPrice; NewPrice = newPrice;  
    }  
}  
public class Stock  
{  
    string symbol;  
    decimal price;  
    public Stock (string symbol) => this.symbol = symbol;  
    public event EventHandler<PriceChangedEventArgs> PriceChanged;  
    protected virtual void OnPriceChanged (PriceChangedEventArgs e)  
    {  
        PriceChanged?.Invoke (this, e);  
    }  
    public decimal Price  
    {  
        get => price;
```

```

    set
    {
        if (price == value) return;
        decimal oldPrice = price;
        price = value;
        OnPriceChanged (new PriceChangedEventArgs (oldPrice, price));
    }
}
}
}

```

Когда событие не несет в себе дополнительной информации, можно использовать предопределенный необобщенный делегат `EventHandler`. Мы перепишем код класса `Stock` так, чтобы событие `PriceChanged` инициировалось после изменения цены, причем какая-либо информация о событии не требуется — необходимо сообщить лишь о самом факте его возникновения. Мы также будем применять свойство `EventArgs.Empty`, чтобы избежать ненужного создания экземпляра `EventArgs`:

```

public class Stock
{
    string symbol;
    decimal price;
    public Stock (string symbol) { this.symbol = symbol; }
    public event EventHandler PriceChanged;
    protected virtual void OnPriceChanged (EventArgs e)
    {
        PriceChanged?.Invoke (this, e);
    }
    public decimal Price
    {
        get { return price; }
        set
        {
            if (price == value) return;
            price = value;
            OnPriceChanged (EventArgs.Empty);
        }
    }
}

```

Средства доступа к событию

Средства доступа к событию представляют собой реализации его операций `+=` и `-=`. По умолчанию средства доступа реализуются неявно компилятором. Взгляните на следующее объявление события:

```
public event EventHandler PriceChanged;
```

Компилятор преобразует его в перечисленные ниже компоненты:

- в закрытое поле делегата;
- в пару открытых функций доступа к событию (`add_PriceChanged` и `remove_PriceChanged`), реализации которых переадресуют операции `+=` и `-=` закрытому полю делегата.

Контроль над процессом преобразования можно взять на себя, определив явные средства доступа. Вот как выглядит ручная реализация события PriceChanged из предыдущего примера:

```
private EventHandler priceChanged;           // Объявить закрытый делегат
public event EventHandler PriceChanged
{
    add { priceChanged += value; }
    remove { priceChanged -= value; }
}
```

Приведенный пример функционально идентичен стандартной реализации средств доступа C# (за исключением того, что C# также обеспечивает безопасность в отношении потоков во время обновления делегата через свободный от блокировок алгоритм сравнения и обмена; см. <http://albahari.com/threading>). Определяя средства доступа к событию самостоятельно, мы указываем компилятору C# на то, что генерировать стандартное поле и логику средств доступа не требуется.

- С помощью явных средств доступа к событию можно применять более сложные стратегии хранения и обращения к лежащему в основе делегату. Ниже описаны три сценария, в которых это полезно.
- Когда средства доступа к событию просто поручают другому классу групповую передачу события.
- Когда класс открывает доступ ко многим событиям, для которых большую часть времени существует очень мало подписчиков, как в случае элемента управления Windows. В таких ситуациях лучше хранить экземпляры делегатов подписчиков в словаре, т.к. со словарем связаны меньшие накладные расходы по хранению, чем с десятками ссылок null на поля делегатов.
- Когда явно реализуется интерфейс, в котором объявлено событие.

Рассмотрим пример, иллюстрирующий последний сценарий:

```
public interface IFoo { event EventHandler Ev; }

class Foo : IFoo
{
    private EventHandler ev;

    event EventHandler IFoo.Ev
    {
        add { ev += value; }
        remove { ev -= value; }
    }
}
```



Части add и remove события транслируются в методы add_XXX и remove_XXX.

Модификаторы событий

Как и методы, события могут быть виртуальными, переопределенными, абстрактными или запечатанными. События также могут быть статическими:

```
public class Foo
{
    public static event EventHandler<EventArgs> StaticEvent;
    public virtual event EventHandler<EventArgs> VirtualEvent;
}
```

Лямбда-выражения

Лямбда-выражение — это неименованный метод, записанный вместо экземпляра делегата. Компилятор немедленно преобразует лямбда-выражение в одну из следующих двух конструкций.

- Экземпляр делегата.
- Дерево выражения, которое имеет тип `Expression<TDelegate>` и представляет код внутри лямбда-выражения в виде объектной модели, поддерживающей обход. Дерево выражения позволяет интерпретировать лямбда-выражение позже во время выполнения (см. раздел “Построение выражений запросов” в главе 8).

В показанном ниже примере лямбда-выражением является `x => x * x`:

```
Transformer sqr = x => x * x;
Console.WriteLine (sqr(3));      // 9
delegate int Transformer (int i);
```



Внутренне компилятор преобразует лямбда-выражение данного типа в закрытый метод, телом которого будет код выражения.

Лямбда-выражение имеет следующую форму:

(параметры) => выражение-или-блок-операторов

Для удобства круглые скобки можно опускать, но только в ситуации, когда есть в точности один параметр выводимого типа.

В нашем примере присутствует единственный параметр `x`, а выражением является `x * x`:

```
x => x * x;
```

Каждый параметр лямбда-выражения соответствует параметру делегата, а тип выражения (которым может быть `void`) — возвращаемому типу этого делегата. В данном примере `x` соответствует параметру `i`, а выражение `x * x` — возвращаемому типу `int` и потому оно совместимо с делегатом `Transformer`:

```
delegate int Transformer (int i);
```

Код лямбда-выражения может быть блоком операторов, а не выражением. Мы можем переписать пример следующим образом:

```
x => { return x * x; };
```

Лямбда-выражения чаще всего применяются с делегатами Func и Action, так что приведенное ранее выражение вы будете регулярно встречать в такой форме:

```
Func<int,int> sqr = x => x * x;
```

Ниже показан пример выражения, которое принимает два параметра:

```
Func<string,string,int> totalLength = (s1, s2) => s1.Length + s2.Length;  
int total = totalLength ("hello", "world"); // total равно 10
```

Если использовать параметры не нужно, тогда их можно *отбрасывать* с помощью символа подчеркивания (начиная с версии C# 9):

```
Func<string,string,int> totalLength = (_,_) => ...
```

Вот пример выражения, которое вообще не принимает аргументов:

```
Func<string> greeter = () => "Hello, world";
```

Начиная с версии C# 10, компилятор разрешает неявную типизацию с помощью лямбда-выражений, которую можно распознать посредством делегатов Func и Action, поэтому данный оператор можно сократить следующим образом:

```
var greeter = () => "Hello, world";
```

Явное указание типов параметров и возвращаемого типа лямбда-выражения

Компилятор обычно способен выводить типы параметров лямбда-выражения из контекста. Когда это не так, вы должны явно указывать тип для каждого параметра. Взгляните на следующие два метода:

```
void Foo<T> (T x) {}  
void Bar<T> (Action<T> a) {}
```

Приведенный далее код не скомпилируется, потому что компилятор не сможет вывести тип x:

```
Bar (x => Foo (x)); // Какой тип имеет x?
```

Исправить ситуацию можно явным указанием типа x следующим образом:

```
Bar (<int x> => Foo (x));
```

Рассматриваемый пример довольно прост и может быть исправлен еще двумя способами:

```
Bar<int> (x => Foo (x)); // Указать параметр типа для Bar  
Bar<int> (Foo); // Как и выше, но использовать группу методов
```

В следующем примере иллюстрируется еще одно использование явных типов параметров (начиная с версии C# 10):

```
var sqr = (int x) => x * x;
```

Компилятор выводит для sqr тип Func<int,int>. (Без указания int неявная типизация завершится неудачей: компилятору известно, что типом sqr должен быть Func<T,T>, но не известно, каким должен быть тип T.)

Начиная с версии C# 10, также можно указывать возвращаемый тип лямбда-выражения:

```
var sqr = int (int x) => x;
```

Указание возвращаемого типа может повысить эффективность компилятора при использовании сложных вложенных лямбда-выражений.

Стандартные параметры лямбда-выражений (C# 12)

Точно так же, как обычные методы могут иметь необязательные параметры:

```
void Print (string message = "") => Console.WriteLine (message);
```

то же самое можно сказать и о лямбда-выражениях:

```
var print = (string message = "") => Console.WriteLine (message);
print ("Hello"); print ();
```

Данное средство полезно при работе с такими библиотеками, как ASP.NET Minimal API.

Захватывание внешних переменных

Лямбда-выражение может ссылаться на любые переменные, которые доступны там, где оно определено. Такие переменные называются *внешними* и могут включать локальные переменные, параметры и поля:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
Console.WriteLine (multiplier (3));           // 6
```

Внешние переменные, на которые ссылается лямбда-выражение, называются *захваченными переменными*. Лямбда-выражение, захватывающее переменные, называется *замыканием*.



Переменные могут захватываться также анонимными методами и локальными методами. Во всех случаях по отношению к захваченным переменным действуют те же самые правила.

Захваченные переменные вычисляются, когда делегат фактически *вызывается*, а не когда переменные были *захвачены*:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
factor = 10;
Console.WriteLine (multiplier (3));           // 30
```

Лямбда-выражения сами могут обновлять захваченные переменные:

```
int seed = 0;
Func<int> natural = () => seed++;
Console.WriteLine (natural());                // 0
Console.WriteLine (natural());                // 1
Console.WriteLine (seed);                     // 2
```

Захваченные переменные имеют свое время жизни, расширенное до времени жизни делегата. В следующем примере локальная переменная seed обычно покидала бы область видимости после того, как выполнение Natural завершено. Но поскольку переменная seed была захвачена, время жизни этой переменной расширяется до времени жизни захватившего ее делегата, т.е. natural:

```
static Func<int> Natural()
{
    int seed = 0;
    return () => seed++;                                // Возвращает замыкание
}

static void Main()
{
    Func<int> natural = Natural();
    Console.WriteLine (natural());                      // 0
    Console.WriteLine (natural());                      // 1
}
```

Локальная переменная, созданная внутри лямбда-выражения, будет уникальной для каждого вызова экземпляра делегата. Если мы переделаем предыдущий пример, чтобы создавать seed внутри лямбда-выражения, то получим разные результаты (что в данном случае нежелательно):

```
static Func<int> Natural()
{
    return() => { int seed = 0; return seed++; };
}

static void Main()
{
    Func<int> natural = Natural();
    Console.WriteLine (natural());                      // 0
    Console.WriteLine (natural());                      // 0
}
```



Внутренне захватывание реализуется “займствованием” захваченных переменных и их помещением в поля закрытого класса. При вызове метода создается экземпляр этого класса и привязывается на время жизни к экземпляру делегата.

Статические лямбда-выражения

При захватывании локальных переменных, параметров, полей экземпляра или ссылки this компилятору может потребоваться генерировать закрытый класс для хранения ссылки на захваченные данные и создать его экземпляр. Это влечет за собой небольшие накладные расходы в плане производительности, т.к. память должна выделяться и впоследствии освобождаться. В ситуациях, когда производительность критична, одна из стратегий микрооптимизации предусматривает минимизацию нагрузки на сборщик мусора за счет обеспечения того, что “горячие” по производительности пути кода нуждаются в небольшом количестве выделений памяти или вообще без них обходятся.

Начиная с версии C# 9, вы можете гарантировать, что лямбда-выражение, локальная функция или анонимный метод не захватывает состояние, применив ключевое слово `static`. Это может быть полезно в сценариях микрооптимизации для предотвращения выделений памяти. Например, вот как применить модификатор `static` к лямбда-выражению:

```
Func<int, int> multiplier = static n => n * 2;
```

Если позже попытаться модифицировать лямбда-выражение, чтобы оно захватывало локальную переменную, то компилятор сообщит об ошибке:

```
int factor = 2;
Func<int, int> multiplier = static n => n * factor; // Не скомпилируется
```



Результатом вычисления самого лямбда-выражения является экземпляр делегата, который требует выделения памяти. Однако если лямбда-выражение не захватывает переменные, тогда компилятор будет многократно использовать единственный кешированный экземпляр в течение всего времени жизни приложения, так что на практике какие-либо затраты отсутствуют.

Указанную возможность допускается также применять с локальными методами. В следующем примере метод `Multiply` не имеет доступа к переменной `factor`:

```
void Foo()
{
    int factor = 123;
    static int Multiply (int x) => x * 2; // Статический локальный метод
}
```

Конечно, метод `Multiply` по-прежнему может явно выделять память, вызывая `new`. Прием защищает от потенциального скрытого выделения. Применение `static` здесь возможно также в качестве инструмента документирования, указывая на сниженный уровень связности.

Статические лямбда-выражения все же могут иметь доступ к статическим переменным и константам (поскольку они не требуют замыкания).



Ключевое слово `static` действует просто как проверка; оно не оказывает никакого влияния на код IL, производимый компилятором. Без ключевого слова `static` компилятор не генерирует замыкание, если в нем нет необходимости (и даже тогда он предпринимает уловки с целью снижения затрат).

Захватывание итерационных переменных

Когда захватывается итерационная переменная в цикле `for`, она трактуется компилятором C# так, как если бы была объявлена *вне* цикла. Таким образом, на каждой итерации захватывается *та же самая* переменная. Приведенный ниже код выводит 333, а не 012:

```
Action[] actions = new Action[3];
for (int i = 0; i < 3; i++)
    actions [i] = () => Console.WriteLine (i);
foreach (Action a in actions) a(); // 333
```

Каждое замыкание (выделенное полужирным) захватывает одну и ту же переменную *i*. (Это действительно имеет смысл, когда считать, что *i* является переменной, значение которой сохраняется между итерациями цикла; при желании *i* можно даже явно изменять внутри тела цикла.) Последствие заключается в том, что при более позднем вызове каждый делегат видит значение *i* на момент *вызыва*, т.е. 3. Чтобы лучше проиллюстрировать сказанное, развернем цикл *for*:

```
Action[] actions = new Action[3];
int i = 0;
actions[0] = () => Console.WriteLine (i);
i = 1;
actions[1] = () => Console.WriteLine (i);
i = 2;
actions[2] = () => Console.WriteLine (i);
i = 3;
foreach (Action a in actions) a(); // 333
```

Если требуется вывести на экран 012, то решение состоит в том, чтобы присвоить итерационную переменную какой-то локальной переменной с областью видимости *внутри* цикла:

```
Action[] actions = new Action[3];
for (int i = 0; i < 3; i++)
{
    int loopScopedi = i;
    actions [i] = () => Console.WriteLine (loopScopedi);
}
foreach (Action a in actions) a(); // 012
```

Из-за того, что во время любой итерации переменная *loopScopedi* создается заново, каждое замыкание захватывает *отличающуюся* переменную.



До выхода версии C# 5.0 циклы *foreach* работали аналогично. Итогом была значительная путаница: в отличие от цикла *for* итерационная переменная в цикле *foreach* неизменяема, и можно было бы ожидать, что она трактуется как локальная по отношению к телу цикла. Хорошая новость заключается в том, что теперь проблема устранена, и вы можете благополучно захватывать итерационную переменную цикла *foreach* безо всяких неожиданностей.

Сравнение лямбда-выражений и локальных методов

Функциональность локальных методов (см. раздел “Локальные методы” в главе 3) частично совпадает с функциональностью лямбда-выражений. Локальные методы обладают следующими тремя преимуществами:

- они могут быть рекурсивными (вызывать сами себя) без применения неусложных трюков;
- они лишены беспорядка, связанного с указанием типа делегата;
- они требуют чуть меньших накладных расходов.

Локальные методы более эффективны, потому что избегают косвенности делегата (за которую приходится платить дополнительными циклами центрального процессора и выделением памяти). Они также могут получать доступ к локальным переменным содержащего метода, не заставляя компилятор “захватывать” захваченные переменные и помещать их в скрытый класс.

Тем не менее, во многих случаях делегат *необходим*, чаще всего при вызове функции более высокого порядка, т.е. метода с параметром типа делегата:

```
public void Foo (Func<int, bool> predicate) { ... }
```

(Дополнительные сведения ищите в главе 8.) В ситуациях подобного рода, так или иначе, необходим делегат, и они представляют собой в точности те случаи, когда лямбда-выражения обычно короче и яснее.

АНОНИМНЫЕ МЕТОДЫ

Анонимные методы — это функциональная возможность, появившаяся в C# 2.0, которая по большей части относится к лямбда-выражениям, введенным в версии C# 3.0. Анонимный метод похож на лямбда-выражение, но в нем отсутствуют:

- неявно типизированные параметры;
- синтаксис выражений (анонимный метод должен всегда быть блоком операторов);
- возможность компиляции в дерево выражения путем присваивания объекту типа Expression<T>.

В анонимном методе используется ключевое слово `delegate`, за которым следует (необязательное) объявление параметра и тело метода. Например:

```
Transformer sqr = delegate (int x) {return x * x;};
Console.WriteLine (sqr(3));                                // 9
delegate int Transformer (int i);
```

Первая строка семантически эквивалентна приведенному ниже лямбда-выражению:

```
Transformer sqr = (int x) => {return x * x};
```

Или просто:

```
Transformer sqr = x => x * x;
```

Анонимные методы захватывают внешние переменные в точности как лямбда-выражения и могут предваряться ключевым словом `static`, которое заставляет их вести себя подобно статическим лямбда-выражениям.



Уникальной особенностью анонимных методов является возможность полностью опускать объявление параметра — даже если делегат его ожидает. Это может быть удобно при объявлении событий со стандартным пустым обработчиком:

```
public event EventHandler Clicked = delegate { };
```

В итоге устраняется необходимость проверки на равенство `null` перед запуском события. Приведенный далее код также будет допустимым:

```
// Обратите внимание, что параметры не указаны:  
Clicked += delegate { Console.WriteLine ("clicked"); };
```

Операторы `try` и исключения

Оператор `try` указывает блок кода, предназначенный для обработки ошибок или очистки. За блоком `try` должен следовать один или большее количество блоков `catch` и/или блок `finally` либо то и другое. Блок `catch` выполняется, когда происходит ошибка в блоке `try`. Блок `finally` выполняется после выполнения блока `try` (или блока `catch`, если он присутствует), обеспечивая очистку независимо от того, возникла ошибка или нет.

Блок `catch` имеет доступ к объекту `Exception`, который содержит информацию об ошибке. Блок `catch` применяется либо для компенсации последствий ошибки, либо для *повторной генерации* исключения. Исключение генерируется повторно, если требуется просто зарегистрировать сам факт возникновения проблемы в журнале или если необходимо сгенерировать исключение нового типа более высокого уровня.

Блок `finally` добавляет к программе детерминизм: среда CLR старается выполнять его всегда. Он полезен для проведения задач очистки вроде закрытия сетевых подключений.

Оператор `try` выглядит следующим образом:

```
try  
{  
    ... // Во время выполнения этого блока может возникнуть исключение  
}  
catch (ExceptionA ex)  
{  
    ... // Обработка исключения типа ExceptionA  
}  
catch (ExceptionB ex)  
{  
    ... // Обработка исключения типа ExceptionB  
}  
finally  
{  
    ... // Код очистки  
}
```

Взгляните на показанную ниже программу:

```
int y = Calc (0);
Console.WriteLine (y);
int Calc (int x) => 10 / x;
```

Поскольку *x* имеет нулевое значение, исполняющая среда генерирует исключение `DivideByZeroException` и программа прекращает работу. Чтобы предотвратить такое поведение, мы перехватываем исключение:

```
try
{
    int y = Calc (0);
    Console.WriteLine (y);
}
catch (DivideByZeroException ex)
{
    Console.WriteLine ("x cannot be zero"); // x не может иметь нулевое значение
}
Console.WriteLine ("program completed"); // программа завершена
int Calc (int x) => 10 / x;
```

Вот результирующий вывод:

```
x cannot be zero
program completed
```



Приведенный простой пример предназначен только для демонстрации обработки исключений. На практике вместо реализации такого сценария лучше перед вызовом `Calc` явно проверять делитель на равенство нулю.

Проверка с целью предотвращения ошибок предпочтительнее реализации блоков `try/catch`, т.к. обработка исключений является относительно затратной в плане ресурсов, требуя немало процессорного времени.

Когда исключение возникает внутри оператора `try`, среда CLR выполняет следующую проверку.

Имеет ли оператор try совместимые блоки catch?

Если да, то управление переходит к совместимому блоку `catch`, далее к блоку `finally` (при его наличии) и затем продолжается нормальное выполнение.

Если нет, то управление переходит прямо к блоку `finally` (при его наличии), после чего среда CLR ищет в стеке вызовов другие блоки `try`; в случае их нахождения она повторяет проверку.

Если ни одна из функций в стеке вызовов не взяла на себя ответственность за исключение, тогда программа прекращает работу.

Конструкция `catch`

Конструкция `catch` указывает тип исключения, подлежащего перехвату. Типом может быть либо класс `System.Exception`, либо какой-то подкласс `System.Exception`.

Указание типа `System.Exception` приводит к перехвату всех возможных ошибок. Это удобно в следующих ситуациях:

- программа потенциально способна восстанавливаться независимо от конкретного типа исключения;
- планируется повторная генерация исключения (возможно, после его регистрации в журнале);
- обработчик ошибок является последним средством перед тем, как программа прекратит работу.

Однако более обычной является ситуация, когда перехватываются исключения специфических типов, чтобы не иметь дела с исключениями, на которые обработчик не был рассчитан (скажем, `OutOfMemoryException`).

Перехватывать исключения нескольких типов можно с помощью множества конструкций `catch` (и снова данный пример проще реализовать посредством явной проверки аргументов, а не за счет обработки исключений):

```
class Test
{
    static void Main (string[] args)
    {
        try
        {
            byte b = byte.Parse (args[0]);
            Console.WriteLine (b);
        }
        catch (IndexOutOfRangeException)
        {
            // Должен быть предоставлен хотя бы один аргумент
            Console.WriteLine ("Please provide at least one argument");
        }
        catch (FormatException)
        {
            // Аргумент должен быть числом
            Console.WriteLine ("That's not a number!");
        }
        catch (OverflowException)
        {
            // Возникло переполнение
            Console.WriteLine ("You've given me more than a byte!");
        }
    }
}
```

Для заданного исключения выполняется только одна конструкция `catch`. Если вы хотите предусмотреть сетку безопасности для перехвата общих исключений (вроде `System.Exception`), то должны размещать более специфические обработчики *первыми*.

Исключение может быть перехвачено без указания переменной, если доступ к свойствам исключения не нужен:

```
catch (OverflowException) // Переменная не указана
{
    ...
}
```

Кроме того, можно опускать и переменную, и тип (тогда будут перехватываться все исключения):

```
catch { ... }
```

Фильтры исключений

В конструкции `catch` можно указывать *фильтр исключений* с помощью конструкции `when`:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{
    ...
}
```

Если в приведенном примере генерируется исключение `WebException`, тогда вычисляется булевское выражение, находящееся после ключевого слова `when`. Если результатом оказывается `false`, то данный блок `catch` игнорируется и просматриваются любые последующие конструкции `catch`. Благодаря фильтрам исключений может появиться смысл в повторном перехвате исключения того же самого типа:

```
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{ ... }
catch (WebException ex) when (ex.Status == WebExceptionStatus.SendFailure)
{ ... }
```

Булевское выражение в конструкции `when` может иметь побочные эффекты, например, вызывать метод, который фиксирует в журнале сведения об исключении для целей диагностики.

Блок `finally`

Блок `finally` выполняется всегда — независимо от того, возникало ли исключение, и полностью ли был выполнен блок `try`. Блоки `finally` обычно используются для размещения кода очистки.

Блок `finally` выполняется в любом из следующих случаев:

- после завершения блока `catch` (или генерации нового исключения);
- после завершения блока `try` (или генерации исключения, для которого не был предусмотрен блок `catch`);
- после того, как поток управления покидает блок `try` из-за наличия оператора перехода (например, `return` или `goto`).

Единственное, что может воспрепятствовать выполнению блока `finally` — бесконечный цикл или неожиданное завершение процесса.

Блок `finally` содействует повышению детерминизма программы. В показанном ниже примере открываемый файл *всегда* закрывается независимо от перечисленных далее обстоятельств:

- блок `try` завершается нормально;
- происходит преждевременный возврат из-за того, что файл пуст (`EndOfStream`);
- во время чтения файла возникает исключение `IOException`:

```
void ReadFile()
{
    StreamReader reader = null; // Из пространства имен System.IO
    try
    {
        reader = File.OpenText ("file.txt");
        if (reader.EndOfStream) return;
        Console.WriteLine (reader.ReadToEnd());
    }
    finally
    {
        if (reader != null) reader.Dispose();
    }
}
```

Здесь мы закрываем файл с помощью вызова `Dispose` на `StreamReader`. Вызов `Dispose` на объекте внутри блока `finally` представляет собой стандартное соглашение, которое явно поддерживается в C# посредством оператора `using`.

Оператор `using`

Многие классы инкапсулируют неуправляемые ресурсы, такие как файловые и графические дескрипторы или подключения к базам данных. Классы подобного рода реализуют интерфейс `System.IDisposable`, в котором определен единственный метод без параметров по имени `Dispose`, предназначенный для очистки этих ресурсов. Оператор `using` предлагает элегантный синтаксис для вызова `Dispose` на объекте реализации `IDisposable` внутри блока `finally`.

Таким образом, код:

```
using (StreamReader reader = File.OpenText ("file.txt"))
{
    ...
}
```

в точности эквивалентен следующему коду:

```
{
    StreamReader reader = File.OpenText ("file.txt");
    try
    {
        ...
    }
    finally
    {
        if (reader != null)
            ((IDisposable)reader).Dispose();
    }
}
```

Объявления using

Если опустить круглые скобки и блок операторов, следующий за оператором `using` (C# 8+), то он становится *объявлением using*. В результате ресурс освобождается, когда поток управления выходит за пределы *включающего* блока операторов:

```
if (File.Exists ("file.txt"))
{
    using var reader = File.OpenText ("file.txt");
    Console.WriteLine (reader.ReadLine ());
    ...
}
```

В данном случае память для `reader` будет освобождена, как только поток управления окажется за пределами блока оператора `if`.

Генерация исключений

Исключения могут генерироваться либо исполняющей средой, либо пользовательским кодом. В приведенном далее примере метод `Display` генерирует исключение `System.ArgumentNullException`:

```
try { Display (null); }
catch (ArgumentNullException ex)
{
    Console.WriteLine ("Caught the exception"); // Исключение перехвачено
}

void Display (string name)
{
    if (name == null)
        throw new ArgumentNullException (nameof (name));
    Console.WriteLine (name);
}
```



Поскольку проверка аргумента на предмет `null` и генерация исключения `ArgumentNullException` — довольно распространенный шаблон кода, в .NET 6 для него предусмотрено сокращение:

```
void Display (string name)
{
    ArgumentNullException.ThrowIfNull (name);
    Console.WriteLine (name);
}
```

Обратите внимание, что указывать имя параметра не пришлось. Причина объясняется в разделе “Атрибут CallerArgumentExpression” далее в главе.

Выражения throw

Конструкция `throw` может появляться и как выражение в функциях, сжатых до выражения:

```
public string Foo () => throw new NotImplementedException ();
```

Выражение `throw` может также находиться внутри тернарной условной операции:

```
string ProperCase (string value) =>
    value == null ? throw new ArgumentException ("value") :
    value == "" ? "":
    char.ToUpper (value[0]) + value.Substring (1);
```

Повторная генерация исключения

Исключение можно перехватывать и генерировать повторно:

```
try { ... }
catch (Exception ex)
{
    // Записать в журнал информацию об ошибке
    ...
    throw; // Повторно сгенерировать то же самое исключение
}
```



Если вместо `throw` указать `throw ex`, то пример сохранит работоспособность, но свойство `StackTrace` исключения больше не будет отражать исходную ошибку.

Повторная генерация в подобной манере дает возможность зарегистрировать в журнале информацию об ошибке, не подавляя ее. Она также позволяет отказаться от обработки исключения, если обстоятельства сложились не так, как ожидалось. Еще один распространенный сценарий предусматривает повторную генерацию исключения более специфического типа:

```
try
{
    ... // Получить значение DateTime из данных XML-элемента
}
catch (FormatException ex)
{
    throw new XmlException ("Invalid DateTime", ex); //Недопустимый формат DateTime
}
```

Обратите внимание, что при создании экземпляра `XmlException` мы передаем конструктору во втором аргументе исходное исключение `ex`. Этот аргумент заполняет свойство `InnerException` нового экземпляра исключения и содействует отладке. Практически все типы исключений предлагают аналогичный конструктор.

Повторная генерация *менее* специфичного исключения может осуществляться при пересечении границы доверия, чтобы не допустить утечки технической информации потенциальному взломщику.

Основные свойства класса `System.Exception`

Ниже описаны наиболее важные свойства класса `System.Exception`.

`StackTrace`

Строка, представляющая все методы, которые были вызваны, начиная с источника исключения и заканчивая блоком `catch`.

Message

Строка с описанием ошибки.

InnerException

Внутреннее исключение (если есть), которое привело к генерации внешнего исключения. Может иметь еще одно свойство InnerException.



Все исключения в C# происходят во время выполнения — в языке C# отсутствует эквивалент проверяемых исключений этапа компиляции, которые есть в Java.

Общие типы исключений

Перечисленные ниже типы исключений широко используются в CLR и библиотеках .NET. Их можно генерировать либо применять в качестве базовых классов для порождения специальных типов исключений.

System.ArgumentException

Генерируется, когда функция вызывается с недопустимым аргументом. Как правило, указывает на наличие ошибки в программе.

System.ArgumentNullException

Подкласс ArgumentException, который генерируется, когда аргумент функции (непредсказуемо) равен null.

System.ArgumentOutOfRangeException

Подкласс ArgumentException, который генерируется, когда (обычно числовой) аргумент имеет слишком большое или слишком малое значение. Например, такое исключение возникает при передаче отрицательного числа в функцию, которая принимает только положительные значения.

System.InvalidOperationException

Генерируется, когда состояние объекта оказывается неподходящим для успешного выполнения метода независимо от любых заданных значений аргументов. В качестве примеров можно назвать чтение неоткрытого файла или получение следующего элемента из перечислителя в случае, если лежащий в основе список был изменен на середине выполнения итерации.

System.NotSupportedException

Генерируется для указания на то, что специфическая функциональность не поддерживается. Хорошим примером может служить вызов метода Add на коллекции, для которой свойство IsReadOnly возвращает true.

System.NotImplementedException

Генерируется для указания на то, что функция пока еще не реализована.

System.ObjectDisposedException

Генерируется, когда объект, на котором вызывается функция, был освобожден.

Еще одним часто встречающимся типом исключения является `NullReferenceException`. Среда CLR генерирует такое исключение, когда вы пытаетесь получить доступ к члену объекта, значение которого равно `null` (указывая на ошибку в коде). Исключение `NullReferenceException` можно генерировать напрямую (в тестовых целях) следующим образом:

```
throw null;
```

Шаблон методов TryXXX

При написании метода в ситуации, когда что-то пошло не так, у вас есть выбор — возвратить код неудачи некоторого вида либо сгенерировать исключение. В общем случае исключение следует генерировать, если ошибка находится за рамками нормального рабочего потока или ожидается, что непосредственно вызывающий код неспособен справиться с ошибкой. Тем не менее, иногда лучше предложить потребителю оба варианта. Примером может служить тип `int`, в котором определены две версии метода `Parse`, отвечающего за разбор:

```
public int Parse (string input);
public bool TryParse (string input, out int returnValue);
```

Если разбор заканчивается неудачей, то метод `Parse` генерирует исключение, а `TryParse` возвращает значение `false`.

Такой шаблон можно реализовать, обеспечив вызов метода `TryXXX` внутри метода `XXX`:

```
public возвращаемый-тип XXX (входной-тип input)
{
    возвращаемый-тип returnValue;
    if (!TryXXX (input, out returnValue))
        throw new YYException (...);
    return returnValue;
}
```

Альтернативы исключениям

Как и метод `int.TryParse`, функция может сообщать о неудаче путем возвращения в вызывающую функцию кода ошибки через возвращаемый тип или параметр. Хотя такой подход хорошо работает с простыми и предсказуемыми отказами, он становится громоздким при необходимости охвата необычных или непредсказуемых ошибок, засоряя сигнатуры методов и привнося ненужную сложность и беспорядок.

Кроме того, его нельзя распространить на функции, не являющиеся методами, такие как операции (например, деление) или свойства. В качестве альтернативы информация об ошибке может храниться в общем местоположении, в котором ее способны видеть все функции из стека вызовов (скажем, можно иметь статический метод, сохраняющий текущий признак ошибки для потока). Тем не менее, это требует от каждой функции участия в шаблоне распространения ошибок, который мало того, что громоздкий, но по иронии судьбы сам подвержен ошибкам.

Перечисление и итераторы

Перечисление

Перечислитель — это допускающий только чтение односторонний курсор по *последовательности* значений. Компилятор C# трактует тип как перечислитель, если он обладает одной из следующих характеристик:

- имеет открытый метод без параметров по имени `MoveNext` и свойство по имени `Current`;
- реализует интерфейс `System.Collections.Generic.IEnumerator<T>`;
- реализует интерфейс `System.Collections.IEnumerator`.

Оператор `foreach` выполняет итерацию по *перечислимому* объекту. Перечислимый объект является логическим представлением последовательности. Это не собственно курсор, а объект, который производит курсор на себе самом. Компилятор C# трактует тип как перечислимый, если он обладает любой из следующих характеристик (проверка осуществляется в указанном порядке):

- имеет открытый метод без параметров по имени `GetEnumerator`, который возвращает перечислитель;
- реализует интерфейс `System.Collections.Generic.IEnumerable<T>`;
- реализует интерфейс `System.Collections.IEnumerable`;

(начиная с версии C# 9) может быть привязан к *расширяющему методу* `GetEnumerator`, который возвращает перечислитель (см. раздел “Расширяющие методы” далее в главе).

Шаблон перечисления выглядит так:

```
class Enumerator // Обычно реализует интерфейс IEnumator или IEnumator<T>
{
    public ТипПеременнойПеречислителя Current { get {...} }
    public bool MoveNext() {...}
}

class Enumerable // Обычно реализует интерфейс IEnumarable или IEnumarable<T>
{
    public Enumerator GetEnumerator() {...}
}
```

Ниже показан высокоуровневый способ выполнения итерации по символам в слове “beer” с использованием оператора `foreach`:

```
foreach (char c in "beer")
    Console.WriteLine (c);
```

А вот низкоуровневый метод проведения итерации по символам в слове “beer” без применения оператора `foreach`:

```
using (var enumerator = "beer".GetEnumerator())
    while (enumerator.MoveNext())
    {
        var element = enumerator.Current;
        Console.WriteLine (element);
    }
```

Если перечислитель реализует интерфейс `IDisposable`, тогда оператор `foreach` также действует как оператор `using`, неявно освобождая объект перечислителя.

Интерфейсы перечисления более подробно рассматриваются в главе 7.

Инициализаторы и выражения коллекций

Перечислимый объект можно создать и заполнить за один шаг с помощью инициализатора коллекции:

```
using System.Collections.Generic;
var list = new List<int> {1, 2, 3};
```

Начиная с версии C# 12, это выражение можно сократить за счет использования выражения коллекции (обратите внимание на квадратные скобки):

```
using System.Collections.Generic;
List<int> list = [1, 2, 3];
```



Выражения коллекции имеют *целевую типизацию*, т.е. тип выражения коллекции `[1, 2, 3]` зависит от типа, которому оно присваивается (в данном случае `List<int>`). В следующем примере целевыми типами являются `int[]` и `Span<int>` (которые будут рассматриваться в главе 23):

```
int[] array = [1, 2, 3];
Span<int> span = [1, 2, 3];
```

Целевая типизация означает, что тип можно опускать в других сценариях, где компилятор способен его определить, например, при вызове методов:

```
Foo ([1, 2, 3]);
void Foo (List<int> numbers) { ... }
```

Компилятор транслирует это в следующий код:

```
using System.Collections.Generic;
List<int> list = new List<int>();
list.Add (1);
list.Add (2);
list.Add (3);
```

Здесь требуется, чтобы перечислимый объект реализовывал интерфейс `System.Collections.IEnumerable` и потому имел метод `Add`, который принимает соответствующее количество параметров для вызова. (Благодаря выражениям коллекций компилятор также поддерживает другие шаблоны, которые позволяют создавать коллекции, доступные только для чтения.)

Похожим образом можно инициализировать словари (см. раздел “Словари” в главе 7):

```
var dict = new Dictionary<int, string>()
{
    { 5, "five" },
    { 10, "ten" }
};
```

Или более компактно:

```
var dict = new Dictionary<int, string>()
{
    [3] = "three",
    [10] = "ten"
};
```

Последний прием допустим не только со словарями, но и с любым типом, для которого существует индексатор.

Итераторы

В то время как оператор `foreach` является *потребителем* перечислителя, итератор выступает в качестве *поставщика* перечислителя. В приведенном ниже примере итератор используется для возвращения последовательности чисел Фибоначчи (в которой каждое число является суммой двух предыдущих чисел):

```
using System;
using System.Collections.Generic;

foreach (int fib in Fibs(6))
    Console.Write (fib + " ");
}

IEnumerable<int> Fibs (int fibCount)
{
    for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
    {
        yield return prevFib;
        int newFib = prevFib+curFib;
        prevFib = curFib;
        curFib = newFib;
    }
}
```

Выход выглядит так:

```
1 1 2 3 5 8
```

Оператор `return` выражает: “Вот значение, которое должно быть возвращено из этого метода”, а оператор `yield return` сообщает: “Вот следующий элемент, который должен быть выдан этим перечислителем”. С каждым оператором `yield` управление возвращается вызывающему компоненту, но состояние вызываемого метода сохраняется, так что данный метод может продолжить свое выполнение, как только вызывающий компонент перечислит следующий элемент. Жизненный цикл такого состояния ограничен перечислителем, поэтому состояние может быть освобождено, когда вызывающий компонент завершит перечисление.



Компилятор преобразует методы итератора в закрытые классы, которые реализуют интерфейсы `IEnumerable<T>` и/или `IEnumerator<T>`. Логика внутри блока итератора “инвертируется” и сращивается с методом `MoveNext` и свойством `Current` класса перечислителя, которые сгенерированы компилятором. Это значит,

что при вызове метода итератора всего лишь создается экземпляр сгенерированного компилятором класса; никакой написанный вами код на самом деле не выполняется! Ваш код запускается только когда начинается перечисление по результирующей последовательности, обычно с помощью оператора `foreach`.

Итераторы могут быть локальными методами (см. раздел “Локальные методы” в главе 3).

Семантика итератора

Итератор представляет собой метод, свойство или индексатор, который содержит один или большее количество операторов `yield`. Итератор должен возвращать один из следующих четырех интерфейсов (иначе компилятор сообщит об ошибке):

```
// Перечислимые интерфейсы
System.Collections.IEnumerable
System.Collections.Generic.IEnumerable<T>

// Интерфейсы перечислителя
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

Семантика итератора отличается в зависимости от того, что он возвращает — реализацию *перечислимого* интерфейса или реализацию интерфейса *перечислителя* (за более подробным описанием обращайтесь в главу 7).

Разрешено применять *несколько операторов yield*:

```
foreach (string s in Foo())
{
    Console.WriteLine(s);           // Выводит "One", "Two", "Three"
    I Enumerable<string> Foo()
    {
        yield return "One";
        yield return "Two";
        yield return "Three";
    }
}
```

Оператор `yield break`

Наличие оператора `return` в блоке итератора не допускается — вместо него должен использоваться оператор `yield break` для указания на то, что блок итератора должен быть завершен преждевременно, не возвращая больше элементов. Для его демонстрации мы можем модифицировать метод `Foo`:

```
I Enumerable<string> Foo (bool breakEarly)
{
    yield return "One";
    yield return "Two";
    if (breakEarly)
        yield break;
    yield return "Three";
}
```

Итераторы и блоки `try/catch/finally`

Оператор `yield return` не может присутствовать в блоке `try`, который имеет конструкцию `catch`:

```
IEnumerable<string> Foo()
{
    try { yield return "One"; }           // Не допускается
    catch { ... }
}
```

Также оператор `yield return` нельзя применять внутри блока `catch` или `finally`. Указанные ограничения объясняются тем фактом, что компилятор должен транслировать итераторы в обычные классы с членами `MoveNext`, `Current` и `Dispose`, а трансляция блоков обработки исключений может привести к чрезмерной сложности.

Однако оператор `yield` можно использовать в блоке `try`, который имеет (только) блок `finally`:

```
IEnumerable<string> Foo()
{
    try { yield return "One"; }           // Нормально
    finally { ... }
}
```

Код в блоке `finally` выполняется, когда потребляемый перечислитель достигает конца последовательности или освобождается. Оператор `foreach` неявно освобождает перечислитель, если произошло преждевременное завершение, обеспечивая безопасный способ применения перечислителей. При явной работе с перечислителем частой ловушкой оказывается преждевременное прекращение перечисления без освобождения перечислителя, т.е. в обход блока `finally`. Во избежание подобного риска код, явно использующий итератор, можно поместить внутрь оператора `using`:

```
string firstElement = null;
var sequence = Foo();
using (var enumerator = sequence.GetEnumerator())
{
    if (enumerator.MoveNext())
        firstElement = enumerator.Current;
```

Компоновка последовательностей

Итераторы в высшей степени комponуемы. Мы можем расширить наш пример с числами Фибоначчи, выводя только четные числа Фибоначчи:

```
using System;
using System.Collections.Generic;
foreach (int fib in EvenNumbersOnly (Fibs(6)))
    Console.WriteLine (fib);
IEnumerable<int> Fibs (int fibCount)
{
    for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
    {
```

```

        yield return prevFib;
        int newFib = prevFib+curFib;
        prevFib = curFib;
        curFib = newFib;
    }
}

IEnumerable<int> EvenNumbersOnly (IEnumerable<int> sequence)
{
    foreach (int x in sequence)
        if ((x % 2) == 0)
            yield return x;
}

```

Каждый элемент не вычисляется вплоть до последнего момента — когда он запрашивается операцией MoveNext. На рис. 4.1 проиллюстрированы запросы, а также вывод данных с течением времени.

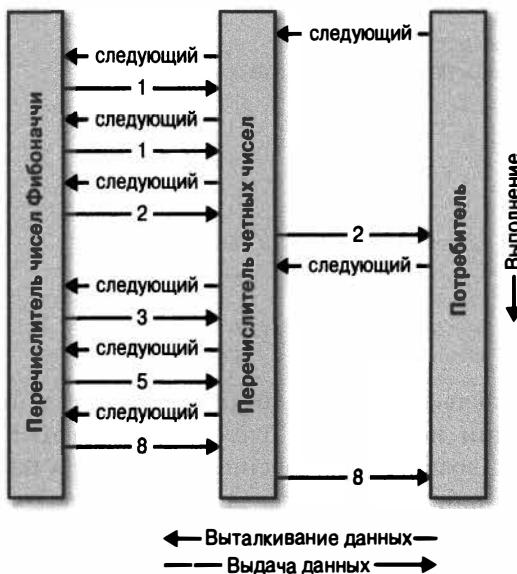


Рис. 4.1. Пример компоновки последовательностей

Возможность компоновки, поддерживаемая шаблоном итератора, жизненно необходима при построении запросов LINQ; мы подробно обсудим данную тему в главе 8.

Типы значений, допускающие null

Ссылочные типы могут представлять несуществующее значение с помощью ссылки null. Тем не менее, типы значений не обладают способностью представлять значения null обычным образом:

```

string s = null; // Нормально, ссылочный тип
int i = null; // Ошибка на этапе компиляции, тип значения не может быть null

```

Чтобы представить null с помощью типа значения, нужно применять специальную конструкцию, которая называется *типом, допускающим null*. Тип, допускающий null, обозначается как тип значения, за которым следует символ ?:

```
int? i = null; // Нормально; тип, допускающий null
Console.WriteLine (i == null); // True
```

Структура Nullable<T>

Тип T? транслируется в System.Nullable<T> — легковесную неизменяющую структуру, которая имеет только два поля, предназначенные для представления значения (Value) и признака наличия значения (HasValue). По существу структура System.Nullable<T> очень проста:

```
public struct Nullable<T> where T : struct
{
    public T Value {get;}
    public bool HasValue {get;}
    public T GetValueOrDefault();
    public T GetValueOrDefault (T defaultValue);
    ...
}
```

Код:

```
int? i = null;
Console.WriteLine (i == null); // True
```

транслируется в:

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine (!i.HasValue); // True
```

Попытка извлечь значение Value, когда в поле HasValue содержится false, приводит к генерации исключения InvalidOperationException. Метод GetValueOrDefault возвращает значение Value, если HasValue равно true, и результат new T() или заданное стандартное значение в противном случае.

Стандартным значением T? является null.

Неявные и явные преобразования с участием типов, допускающих null

Преобразование из T в T? будет неявным, в то время как из T? в T — явным:

```
int? x = 5; // неявное
int y = (int)x; // явное
```

Явное приведение полностью эквивалентно обращению к свойству Value объекта типа, допускающего null. Следовательно, если HasValue равно false, тогда генерируется исключение InvalidOperationException.

Упаковка и распаковка значений типов, допускающих null

Когда T? упаковывается, упакованное значение в куче содержит T, а не T?. Такая оптимизация возможна из-за того, что упакованное значение относится к ссылочному типу, который уже способен выражать null.

В C# также разрешено распаковывать типы, допускающие null, с помощью операции as. Если приведение не удается, тогда результатом будет null:

```
object o = "string";
int? x = o as int?;
Console.WriteLine (x.HasValue);    // False
```

Подъем операций

В структуре Nullable<T> не определены такие операции, как <, > или даже ==. Несмотря на это, следующий код успешно компилируется и выполняется:

```
int? x = 5;
int? y = 10;
bool b = x < y;                // true
```

Код работает благодаря тому, что компилятор заимствует, или “поднимает”, операцию “меньше чем” у лежащего в основе типа значения. Семантически предыдущее выражение сравнения транслируется так:

```
bool b = (x.HasValue && y.HasValue) ? (x.Value < y.Value) : false;
```

Другими словами, если x и y имеют значения, то сравнение производится посредством операции “меньше чем” типа int; в противном случае результатом будет false.

Подъем операций означает возможность неявного использования операций из типа T для T?. Вы можете определить операции для типа T?, чтобы представить специализированное поведение в отношении null, но в подавляющем большинстве случаев лучше полагаться на автоматическое применение компилятором систематической логики работы со значением null. Ниже показано несколько примеров:

```
int? x = 5;
int? y = null;

// Примеры использования операции эквивалентности
Console.WriteLine (x == y);        // False
Console.WriteLine (x == null);     // False
Console.WriteLine (x == 5);        // True
Console.WriteLine (y == null);     // True
Console.WriteLine (y == 5);        // False
Console.WriteLine (y != 5);        // True

// Примеры применения операций отношения
Console.WriteLine (x < 6);        // True
Console.WriteLine (y < 6);        // False
Console.WriteLine (y > 6);        // False

// Примеры использования всех других операций
Console.WriteLine (x + 5);        // 10
Console.WriteLine (x + y);        // null (выводит пустую строку)
```

Компилятор представляет логику в отношении null по-разному в зависимости от категории операции. Правила объясняются в последующих разделах.

Операции эквивалентности (== и !=)

Поднятые операции эквивалентности обрабатывают значения null точно так же, как поступают ссылочные типы. Это означает, что два значения null равны:

```
Console.WriteLine (      null ==      null);      // True
Console.WriteLine ((bool?)null == (bool?)null);    // True
```

Более того:

- если в точности один операнд имеет значение null, то операнды не равны;
- если оба операнда отличаются от null, то сравниваются их свойства Value.

Операции отношения (<, <=, >=, >)

Работа операций отношения основана на принципе, согласно которому сравнение operandов null не имеет смысла. Это означает, что сравнение null либо с null, либо со значением, отличающимся от null, дает в результате false:

```
bool b = x < y; // Транслируется в:
bool b = (x.HasValue && y.HasValue)
        ? (x.Value < y.Value)
        : false;
// b равно false (предполагая, что x равно 5, а y - null)
```

Остальные операции (+, -, *, /, %, &, |, ^, <<, >>, +, ++, --, !, ~)

Остальные операции возвращают null, когда любой из operandов равен null. Такой шаблон должен быть хорошо знаком пользователям языка SQL:

```
int? c = x + y; // Транслируется в:
int? c = (x.HasValue && y.HasValue)
        ? (int?) (x.Value + y.Value)
        : null;
// c равно null (предполагая, что x равно 5, а y - null)
```

Исключением будет ситуация, когда операции & и | применяются к bool?, что мы вскоре обсудим.

Смешивание типов, допускающих и не допускающих null

Типы, допускающие и не допускающие null, можно смешивать (прием работает, поскольку существует неявное преобразование из T в T?):

```
int? a = null;
int b = 2;
int? c = a + b; // c равно null - эквивалентно a + (int?)b
```

Тип `bool?` и операции `&` и `|`

Когда предоставляются operandы типа `bool?`, операции `&` и `|` трактуют `null` как *неизвестное значение*. Таким образом, `null | true` дает `true` по следующим причинам:

- если неизвестное значение равно `false`, то результатом будет `true`;
- если неизвестное значение равно `true`, то результатом будет `true`.

Аналогичным образом `null & false` дает `false`. Подобное поведение должно быть знакомо пользователям языка SQL. Ниже приведены другие комбинации:

```
bool? n = null;
bool? f = false;
bool? t = true;
Console.WriteLine (n | n);      // (null)
Console.WriteLine (n | f);      // (null)
Console.WriteLine (n | t);      // True
Console.WriteLine (n & n);      // (null)
Console.WriteLine (n & f);      // False
Console.WriteLine (n & t);      // (null)
```

Типы значений, допускающие `null`, и операции для работы с `null`

Типы значений, допускающие `null`, особенно хорошо работают с операцией `??` (см. раздел “Операция объединения с `null`” в главе 2), как иллюстрируется в следующем примере:

```
int? x = null;
int y = x ?? 5;                      // y равно 5
int? a = null, b = 1, c = 2;
Console.WriteLine (a ?? b ?? c); // 1(первое значение, отличающееся от null)
```

Использование операции `??` эквивалентно вызову метода `GetValueOrDefault` с явным стандартным значением за исключением того, что выражение для стандартного значения никогда не вычисляется, если переменная не равна `null`.

Типы значений, допускающие `null`, также удобно применять с `null`-условной операцией (см. раздел “`null`-условная операция” в главе 2). В показанном далее примере переменная `length` получает значение `null`:

```
System.Text.StringBuilder sb = null;
int? length = sb?.ToString().Length;
```

Скомбинировав такой код с операцией объединения с `null`, переменной `length` можно присвоить значение 0 вместо `null`:

```
int length = sb?.ToString().Length ?? 0;    // length получает значение 0,
                                            // если sb равно null
```

Сценарии использования типов значений, допускающих null

Один из самых распространенных сценариев использования типов значений, допускающих null — представление неизвестных значений. Он часто встречается в программировании для баз данных, когда класс отображается на таблицу со столбцами, допускающими значение null. Если такие столбцы хранят строковые значения (например, столбец EmailAddress в таблице Customer), то никаких проблем не возникает, потому что string в среде CLR является ссылочным типом, который может быть null. Однако большинство других типов столбцов SQL отображаются на типы структур CLR, что делает типы значений, допускающие null, очень удобными при отображении типов SQL на типы CLR:

```
// Отображается на таблицу Customer в базе данных
public class Customer
{
    ...
    public decimal? AccountBalance;
}
```

Тип, допускающий null, может также применяться для представления поддерживаемого поля в так называемом *свойстве окружения* (ambient property). Когда свойство окружения равно null, оно возвращает значение своего родителя:

```
public class Row
{
    ...
    Grid parent;
    Color? color;

    public Color Color
    {
        get { return color ?? parent.Color; }
        set { color = value == parent.Color ? (Color?)null : value; }
    }
}
```

Альтернативы типам значений, допускающим null

До того, как типы значений, допускающие null, стали частью языка C# (т.е. до версии C# 2.0), существовало много стратегий для работы с такими типами, и по историческим причинам их примеры по-прежнему можно встретить в библиотеках .NET. Одна из таких стратегий предусматривала назначение определенного значения, отличающегося от null, в качестве “значения null”; пример можно найти в классах строк и массивов. Метод string.IndexOf возвращает “магическое” значение -1, когда символ в строке не обнаружен:

```
int i = "Pink".IndexOf('b');
Console.WriteLine(i);                                // -1
```

Тем не менее, метод `Array.IndexOf` возвращает `-1`, только если индекс ограничен нулем. Более общий рецепт заключается в том, что `IndexOf` возвращает значение на единицу меньше нижней границы массива. В следующем примере `IndexOf` возвращает `0`, если элемент не найден:

```
// Создать массив, нижняя граница которого равна 1, а не 0:  
Array a = Array.CreateInstance (typeof (string),  
                               new int[] {2}, new int[] {1});  
a.SetValue ("a", 1);  
a.SetValue ("b", 2);  
Console.WriteLine (Array.IndexOf (a, "c")); // 0
```

Выбор “магического” значения сопряжен с проблемами по нескольким причинам.

- Такой подход приводит к тому, что каждый тип значения имеет разное представление `null`. По контрасту с этим типы, допускающие `null`, предлагаю один общий шаблон, который работает для всех типов значений.
- Приемлемого значения может и не быть. В предыдущем примере всегда использовать `-1` не получится. То же самое справедливо для ранее приведенного примера, представляющего неизвестный баланс счета (`AccountBalance`).
- Если забыть о проверке на равенство “магическому” значению, то появится некорректное значение, которое может оказаться незамеченным вплоть до стадии выполнения — когда обнаружится неожиданное поведение. С другой стороны, если забыть о проверке `HasValue` на равенство `null`, тогда немедленно генерируется исключение `InvalidOperationException`.
- Способность значения быть `null` не отражена в *типе*. Типы сообщают о целях программы, позволяя компилятору проверять корректность и применять согласованный набор правил.

Ссылочные типы, допускающие `null`

В то время как *типы значений, допускающие null*, наделяют типы значений способностью принимать значение `null`, *ссылочные типы, допускающие null* (C# 8+), делают противоположное. Когда включены, они привносят в ссылочные типы определенную долю *возможности не быть null*, цель которой — избежать возникновения ошибок `NullReferenceException`.

Ссылочные типы, допускающие `null`, вводят уровень безопасности, обеспечиваемый исключительно компилятором в форме предупреждений, когда он обнаруживает код, который подвержен риску генерации `NullReferenceException`.

Чтобы включить ссылочные типы, допускающие `null`, потребуется добавить элемент `Nullable` в файл проекта `.csproj` (при желании их включения для всего проекта):

```
<PropertyGroup>  
  <Nullable>enable</Nullable>  
</PropertyGroup>
```

и/или использовать следующие директивы в коде там, где они должны вступить в силу:

```
#nullable enable      // Включает ссылочные типы, допускающие null,  
                      // начиная с этой точки  
#nullable disable    // Отключает ссылочные типы, допускающие null,  
                      // начиная с этой точки  
#nullable restore    // Переустанавливает ссылочные типы, допускающие null,  
                      // согласно настройке проекта
```

После включения компилятор по умолчанию считает, что ссылочный тип не может быть `null`: если вы хотите, чтобы ссылочный тип принимал значения `null` без генерации компилятором предупреждения, тогда должны применять суффикс `? для обозначения ссылочного типа, допускающего null`. В показанном ниже примере `s1` не допускает значение `null`, тогда как `s2` допускает:

```
#nullable enable      // Включить ссылочные типы, допускающие null  
string s1 = null;     // Компилятор генерирует предупреждение!  
string? s2 = null;  // Нормально: s2 - ссылочный тип, допускающий null
```



Поскольку ссылочные типы, допускающие `null`, являются конструкциями этапа компиляции, во время выполнения между `string` и `string?` нет никакой разницы. Напротив, типы значений, допускающие `null`, привносят в систему типов кое-что конкретное, а именно — структуру `Nullable<T>`.

Следующий код также приводит к генерации предупреждения из-за отсутствия инициализации `x`:

```
class Foo { string x; }
```

Предупреждение исчезнет, если вы инициализируете `x` посредством инициализатора поля или с помощью кода в конструкторе.

null-терпимая операция

Компилятор также выдаст предупреждение при разыменовании ссылочного типа, допускающего `null`, если посчитает, что может возникнуть исключение `NullReferenceException`. В показанном далее коде доступ к свойству `Length` строки вызывает генерацию предупреждения:

```
void Foo (string? s) => Console.Write (s.Length);
```

Избавиться от предупреждения можно за счет использования `null-терпимой операции (!)`:

```
void Foo (string? s) => Console.Write (s!.Length);
```

Применение `null-терпимой операции` в этом примере опасно тем, что в конечном итоге мы можем получить то же самое исключение `NullReferenceException`, которое в первую очередь пытались избежать. Вот как можно было бы исправить ситуацию:

```
void Foo (string? s)
{
    if (s != null) Console.WriteLine (s.Length);
}
```

Обратите внимание, что теперь `null`-терпимая операция не нужна. На самом деле компилятор выполняет *статический анализ потока управления* и достаточно интеллектуален, чтобы сделать вывод (по крайней мере, в простых случаях) о том, что разыменование безопасно и шансов на возникновение исключения `NullReferenceException` нет.

Способность компилятора обнаруживать и предупреждать не является “пленепробиваемой”, к тому же существуют пределы в плане того, что возможнохватить. Например, компилятор не может знать, заполнены ли элементы массива, а потому следующий код не приводит к выдаче предупреждения:

```
var strings = new string[10];
Console.WriteLine (strings[0].Length);
```

Разъединение контекстов с заметками и с предупреждениями

Включение ссылочных типов, допускающих `null`, через директиву `#nullable enable` (или посредством настройки проекта `<Nullable>enable </Nullable>`) выполняет два действия.

- Включает контекст с заметками о допустимости значения `null`, который сообщает компилятору о необходимости трактовки всех объявлений переменных ссылочных типов как не допускающих `null`, если только они не снабжены суффиксом `?`.
- Включает контекст с предупреждениями о допустимости значения `null`, который сообщает компилятору о необходимости выдавать предупреждения при обнаружении кода, где есть риск генерации исключения `NullReferenceException`.

Иногда полезно разъединять эти две концепции и включать только контекст с заметками или (что менее полезно) только контекст с предупреждениями:

```
#nullable enable annotations      // Включить контекст с заметками
// ИЛИ:
#nullable enable warnings        // Включить контекст с предупреждениями
```

(Тот же самый трюк работает с `#nullable disable` и `#nullable restore`.) Вы также можете достичь цели через файл проекта:

```
<Nullable>annotations</Nullable>
<!-- ИЛИ -->
<Nullable>warnings</Nullable>
```

Включение только контекста с заметками для отдельного класса или сборки может стать хорошим первым шагом к вводу ссылочных типов, допускающих `null`, в унаследованную кодовую базу. За счет корректной пометки открытых членов класс или сборка сможет действовать как “добропорядочный гражданин” в отношении других классов или сборок, так что у них появится возмож-

ность в полной мере использовать ссылочные типы, допускающие null, без необходимости иметь дело с предупреждениями в классе или сборке.

Трактовка предупреждений о допустимости значения null как ошибок

В проектах, реализуемых с нуля, имеет смысл с самого начала включать контекст, допускающий null. Возможно, вы захотите принять дополнительный шаг в плане трактовки предупреждений о допустимости значения null как ошибок, чтобы ваш проект не мог успешно скомпилироваться до тех пор, пока не будут устранены все предупреждения, касающиеся допустимости значения null:

```
<PropertyGroup>
  <Nullable>enable</Nullable>
  <WarningsAsErrors>CS8600;CS8602;CS8603</WarningsAsErrors>
</PropertyGroup>
```

Расширяющие методы

Расширяющие методы позволяют расширять существующий тип новыми методами, не изменяя определение исходного типа. Расширяющий метод — это статический метод статического класса, в котором к первому параметру применен модификатор `this`. Типом первого параметра должен быть тип, который расширяется:

```
public static class StringHelper
{
    public static bool IsCapitalized (this string s)
    {
        if (string.IsNullOrEmpty(s)) return false;
        return char.ToUpper (s[0]);
    }
}
```

Расширяющий метод `IsCapitalized` может вызываться так, как если бы он был методом экземпляра класса `string`:

```
Console.WriteLine ("Perth".IsCapitalized());
```

Вызов расширяющего метода при компиляции транслируется в обычный вызов статического метода:

```
Console.WriteLine (StringHelper.IsCapitalized ("Perth"));
```

Такая трансляция работает следующим образом:

```
arg0.Method (arg1, arg2, ...);           // Вызов расширяющего метода
StaticClass.Method (arg0, arg1, arg2, ...); // Вызов статического метода
```

Интерфейсы тоже можно расширять:

```
public static T First<T> (this I Enumerable<T> sequence)
{
    foreach (T element in sequence)
        return element;
```

```
        throw new InvalidOperationException ("No elements!"); //Элементы отсутствуют
    }
    ...
Console.WriteLine ("Seattle".First());                                // S
```

Цепочки расширяющих методов

Как и методы экземпляра, расширяющие методы предлагают аккуратный способ для связывания вызовов функций в цепочки. Взгляните на следующие две функции:

```
public static class StringHelper
{
    public static string Pluralize (this string s) {...}
    public static string Capitalize (this string s) {...}
}
```

Строковые переменные `x` и `y` эквивалентны и получают значение "Sausages", но `x` использует расширяющие методы, тогда как `y` — статические:

```
string x = "sausage".Pluralize().Capitalize();
string y = StringHelper.Capitalize (StringHelper.Pluralize ("sausage"));
```

Неоднозначность и распознавание

Пространства имен

Расширяющий метод не может быть доступен до тех пор, пока его пространство имен не окажется в области видимости, обычно за счет импорта посредством оператора `using`. Взгляните на приведенный далее расширяющий метод `IsCapitalized`:

```
using System;
namespace Utils
{
    public static class StringHelper
    {
        public static bool IsCapitalized (this string s)
        {
            if (string.IsNullOrEmpty(s)) return false;
            return char.ToUpper (s[0]);
        }
    }
}
```

Для применения метода `IsCapitalized` в показанном ниже приложении должно импортироваться пространство имен `Utils`, иначе на этапе компиляции возникнет ошибка:

```
namespace MyApp
{
    using Utils;
    class Test
    {
        static void Main() => Console.WriteLine ("Perth".IsCapitalized());
    }
}
```

Расширяющий метод или метод экземпляра

Любой совместимый метод экземпляра всегда будет иметь преимущество над расширяющим методом. В следующем примере предпочтение всегда будет отдаваться методу Foo класса Test — даже если осуществляется вызов с аргументом x типа int:

```
class Test
{
    public void Foo (object x) { } // Этот метод всегда имеет преимущество
}
static class Extensions
{
    public static void Foo (this Test t, int x) { }
}
```

Единственный способ обратиться к расширяющему методу в такой ситуации — воспользоваться нормальным статическим синтаксисом; другими словами, Extensions.Foo(...).

Расширяющий метод или другой расширяющий метод

Если два расширяющих метода имеют одинаковые сигнатуры, тогда расширяющий метод должен вызываться как обычный статический метод, чтобы устранить неоднозначность при вызове. Однако если один расширяющий метод имеет более специфичные аргументы, то предпочтение будет отдаваться ему.

В целях иллюстрации рассмотрим два класса:

```
static class StringHelper
{
    public static bool IsCapitalized (this string s) {...}
}
static class ObjectHelper
{
    public static bool IsCapitalized (this object s) {...}
}
```

В следующем коде вызывается метод IsCapitalized класса StringHelper:

```
bool test1 = "Perth".IsCapitalized();
```

Классы и структуры считаются более специфичными, чем интерфейсы.

Понижение уровня расширяющего метода

Интересный сценарий может возникнуть, когда в Microsoft добавят в библиотеку времени выполнения .NET расширяющий метод, который конфликтует с расширяющим методом в какой-то существующей сторонней библиотеке. Как автор сторонней библиотеки, вы можете принять решение “отозвать” свой метод расширения, но не удалять его и не нарушать двоичную совместимость с существующими потребителями.

К счастью, этого легко добиться, просто удалив ключевое слово this из определения расширяющего метода. В итоге расширяющий метод понижается до обычного статического метода. Изящество такого решения заключается в том, что любая сборка, скомпилированная с использованием старой библиоте-

ки, продолжит работать (и привязываться к методу, как и ранее). Причина в том, что на этапе компиляции вызовы расширяющих методов преобразуются в вызовы статических методов.

Потребители будут затронуты понижением уровня только в случае перекомпиляции своего кода, и тогда вызовы прежнего расширяющего метода будут привязаны к версии Microsoft (в случае импортирования соответствующего пространства имен). Если потребитель по-прежнему пожелает обратиться к вашему методу, тогда он может вызвать его как статический метод.

Анонимные типы

Анонимный тип — это простой класс, созданный на лету с целью хранения набора значений. Для создания анонимного типа применяется ключевое слово new с инициализатором объекта, указывающим свойства и значения, которые будет содержать тип; например:

```
var dude = new { Name = "Bob", Age = 23 };
```

Компилятор транслирует данный оператор в (приблизительно) такой код:

```
internal class AnonymousGeneratedTypeName
{
    private string name; // Фактическое имя поля несущественно
    private int age; // Фактическое имя поля несущественно
    public AnonymousGeneratedTypeName (string name, int age)
    {
        this.name = name; this.age = age;
    }

    public string Name => name;
    public int Age => age;

    // Методы Equals и GetHashCode переопределены (см. главу 6).
    // Метод ToString также переопределен.

}

...
var dude = new AnonymousGeneratedTypeName ("Bob", 23);
```

При ссылке на анонимный тип должно использоваться ключевое слово var, т.к. имя этого типа не известно.

Имя свойства анонимного типа может быть выведено из выражения, которое само по себе является идентификатором (или заканчивается им); таким образом:

```
int Age = 23;
var dude = new { Name = "Bob", Age, Age.ToString().Length };
```

эквивалентно:

```
var dude = new { Name = "Bob", Age = Age, Length = Age.ToString().Length };
```

Два экземпляра анонимного типа, объявленные внутри одной сборки, будут иметь один и тот же лежащий в основе тип, если их элементы идентично именованы и типизированы:

```
var a1 = new { X = 2, Y = 4 };
var a2 = new { X = 2, Y = 4 };
Console.WriteLine (a1.GetType() == a2.GetType()); // True
```

Кроме того, метод `Equals` переопределен, чтобы выполнять *структурное сравнение эквивалентности* (сравнение данных):

```
Console.WriteLine (a1.Equals (a2)); // True
```

С другой стороны операция сравнения эквивалентности (`==`) выполняет ссылочное сравнение:

```
Console.WriteLine (a1 == a2); // False
```

Можно создавать массивы анонимных типов, как показано ниже:

```
var dudes = new []
{
    new { Name = "Bob", Age = 30 },
    new { Name = "Tom", Age = 40 }
};
```

Метод не может (удобно) возвращать объект анонимного типа, поскольку писать метод с возвращаемым типом `var` не допускается:

```
var Foo() => new { Name = "Bob", Age = 30 }; // Незаконно!
```

(В последующих разделах будут описаны записи и кортежи, которые предлагают альтернативные подходы для возвращения нескольких значений из метода.)

Анонимные типы являются неизменяемыми, поэтому их экземпляры нельзя модифицировать после создания. Тем не менее, начиная с версии C# 10, можно использовать ключевое слово `with` для создания копии с вариациями (*неразрушающее изменение*):

```
var a1 = new { A = 1, B = 2, C = 3, D = 4, E = 5 };
var a2 = a1 with { E = 10 };
Console.WriteLine (a2); // { A = 1, B = 2, C = 3, D = 4, E = 10 }
```

Анонимные типы особенно удобны при написании запросов LINQ (глава 8).

Кортежи

Подобно анонимным типам кортежи предлагают простой способ хранения набора значений. Главный замысел кортежей — безопасно возвращать множество значений из метода, не прибегая к параметрам `out` (то, что невозможно делать с помощью анонимных типов). Однако тогда были введены записи, предлагающие краткий типизированный подход, который рассматривается в следующем разделе.



Кортежи поддерживают почти все, что обеспечивают анонимные типы, и обладают потенциальным преимуществом, будучи типами значений, но их основной недостаток связан со стиранием типов во время выполнения вместе с именованными элементами (как вскоре будет показано).

Создать лiteralный кортеж проще всего, указав в круглых скобках список желаемых значений. В результате создается кортеж с *неименованными* элементами, на которые можно ссылаться как на Item1, Item2 и т.д.:

```
var bob = ("Bob", 23); // Позволить компилятору вывести типы элементов  
Console.WriteLine(bob.Item1); // Bob  
Console.WriteLine(bob.Item2); // 23
```

Кортежи являются типами значений с изменяемыми (допускающими чтение/запись) элементами:

```
var joe = bob; // joe - *копия* bob  
joe.Item1 = "Joe"; // Изменить значение элемента Item1 в joe с Bob на Joe  
Console.WriteLine(bob); // (Bob, 23)  
Console.WriteLine(joe); // (Joe, 23)
```

В отличие от анонимных типов *тип кортежа* можно указывать явно. Нужно лишь перечислить типы элементов в круглых скобках:

```
(string,int) bob = ("Bob", 23); // Ключевое слово var для кортежей  
// не обязательно!
```

Это означает, что кортеж можно удобно возвращать из метода:

```
(string,int) person = GetPerson(); // При желании взамен можно было бы  
// применить var  
Console.WriteLine(person.Item1); // Bob  
Console.WriteLine(person.Item2); // 23  
(string,int) GetPerson() => ("Bob", 23);
```

Кортежи хорошо сочетаются с обобщениями, так что все следующие типы законны:

```
Task<(string,int)>  
Dictionary<(string,int), Uri>  
IEnumerable<(int id, string name)> // Именование элементов описано ниже
```

Именование элементов кортежа

При создании лiteralных кортежей элементам можно дополнительно назначать *содержательные имена*:

```
var tuple = (name:"Bob", age:23);  
Console.WriteLine(tuple.name); // Bob  
Console.WriteLine(tuple.age); // 23
```

То же самое разрешено делать при указании *типов кортежей*:

```
var person = GetPerson();  
Console.WriteLine(person.name); // Bob  
Console.WriteLine(person.age); // 23  
(string name, int age) GetPerson() => ("Bob", 23);
```



В разделе “Записи” далее в главе будет показано, как можно определять простые классы или структуры, упрощая определение формального типа возвращаемого значения:

```
var person = GetPerson();
Console.WriteLine (person.Name);                                // Bob
Console.WriteLine (person.Age);                                 // 23

Person GetPerson() => new ("Bob", 23);
record Person (string Name, int Age);
```

В отличие от кортежей свойства записи (Name и Age) строго типизированы, поэтому их можно легко реорганизовать. Такой подход также снижает дублирование кода и способствует эффективному проектированию несколькими способами. Во-первых, процесс выбора простого, ненадуманного имени для типа помогает проверить правильность проекта (невозможность поступить так может указывать на отсутствие единой связной цели). Во-вторых, вполне вероятно, что в конечном итоге к записи будут добавлены методы или другой код (правильно названные типы обычно *привлекают код*), а перенос кода в данные является базовым принципом качественного объектно-ориентированного проектирования.

Обратите внимание, что элементы по-прежнему можно трактовать как неименованные и ссылаться на них как на Item1, Item2 и т.д. (хотя Visual Studio скрывает эти поля от средства IntelliSense).

Имена элементов автоматически выводятся из имен свойств или полей:

```
var now = DateTime.Now;
var tuple = (now.Day, now.Month, now.Year);
Console.WriteLine (tuple.Day);                                     // Нормально
```

Кортежи совместимы по типу друг с другом, если типы их элементов совпадают (по порядку). Совпадение имен элементов не обязательно:

```
(string name, int age, char sex) bob1 = ("Bob", 23, 'M');
(string age, int sex, char name) bob2 = bob1;                  // Ошибки нет!
```

Рассматриваемый пример приводит к сбивающим с толку результатам:

```
Console.WriteLine (bob2.Name);                                  // M
Console.WriteLine (bob2.Age);                                 // Bob
Console.WriteLine (bob2.Sex);                                // 23
```

Стирание типов

Ранее мы заявляли, что компилятор C# поддерживает анонимные типы путем построения специальных классов с именованными свойствами для каждого элемента. В случае кортежей компилятор C# работает по-другому и задействует существующее семейство обобщенных структур:

```
public struct ValueTuple<T1>
public struct ValueTuple<T1, T2>
public struct ValueTuple<T1, T2, T3>
...
```

Каждая структура `ValueTuple<>` содержит поля с именами `Item1`, `Item2` и т.д. Следовательно, `(string, int)` является псевдонимом для `ValueTuple<string, int>`, а это значит, что именованные элементы кортежей не имеют соответствующих имен свойств внутри лежащих в основе типов. Взамен имена существуют только в исходном коде и в “воображении” компилятора. Во время выполнения имена по обыкновению исчезают, так что после декомпиляции программы, которая ссылается на именованные элементы кортежей, вы увидите только ссылки на `Item1`, `Item2` и т.д. Более того, когда вы исследуете переменную типа кортежа в отладчике после ее присваивания экземпляру `object` (или выводите ее на экран в LINQPad), имена элементов отсутствуют. И самое главное — вы не можете использовать *рефлексию* (глава 18) для определения имен элементов кортежа во время выполнения. Это означает, что с помощью таких API-интерфейсов, как `System.Net.Http.HttpClient`, кортежи не могут заменять анонимные типы в сценариях вроде показанного ниже:

```
// Создать полезную нагрузку JSON:  
var json = JsonContent.Create (new { id = 123, name = "Test" })
```



Мы говорим, что имена исчезают *по обыкновению*, т.к. существует исключение. Для методов/свойств, которые возвращают именованные типы кортежей, компилятор выпускает имена элементов, применяя к возвращаемому типу члена специальный атрибут под названием `TupleElementNamesAttribute` (см. раздел “Атрибуты” далее в главе). Это позволяет именованным элементам работать при вызове методов в другой сборке (исходный код которой компилятору не доступен).

Назначение псевдонимов кортежам (C# 12)

Начиная с версии C# 12, с использованием директивы `using` можно определять псевдонимы для кортежей:

```
using Point = (int, int);  
Point p = (3, 4);
```

Такой прием работает и с кортежами, имеющими именованные элементы:

```
using Point = (int X, int Y); // Допустимо (но не обязательно *хорошо*!)  
Point p = (3, 4);
```

Вскоре вы увидите, что записи предлагают полностью типизированное решение с таким же уровнем краткости:

```
Point p = new (3, 4);  
record Point (int X, int Y);
```

Метод `ValueTuple.Create`

Кортежи можно создавать также через фабричный метод из (необобщенного) типа `ValueTuple`:

```
ValueTuple<string,int> bob1 = ValueTuple.Create ("Bob", 23);  
(string,int) bob2 = ValueTuple.Create ("Bob", 23);  
(string name, int age) bob3 = ValueTuple.Create ("Bob", 23);
```

Деконструирование кортежей

Кортежи неявно поддерживают шаблон деконструирования (см. раздел “Деконструкторы” в главе 3), поэтому кортеж можно легко *деконструировать* в индивидуальные переменные. Взгляните на следующий код:

```
var bob = ("Bob", 23);
string name = bob.Item1;
int age = bob.Item2;
```

С помощью деконструктора кортежа этот код можно упростить:

```
var bob = ("Bob", 23);
(string name, int age) = bob;      // Деконструировать кортеж bob в
                                    // отдельные переменные (name и age).
Console.WriteLine (name);
Console.WriteLine (age);
```

Синтаксис деконструирования очень похож на синтаксис объявления кортежа с именованными элементами, что может привести к путанице. Разница подчеркивается в приведенном ниже коде:

```
(string name, int age)      = bob; // Деконструирование кортежа
(string name, int age) bob2 = bob; // Объявление нового кортежа
```

Вот еще один пример, на этот раз с вызовом метода и выведением типа (var):

```
var (name, age, sex) = GetBob();
Console.WriteLine (name);           // Bob
Console.WriteLine (age);            // 23
Console.WriteLine (sex);           // M
```

```
string, int, char) GetBob() => ( "Bob", 23, 'M');
```

Деконструировать можно также прямо в поля и свойства, что обеспечивает удобный сокращенный прием для заполнения множества полей или свойств в конструкторе:

```
class Point
{
    public readonly int X, Y;
    public Point (int x, int y) => (X, Y) = (x, y);
}
```

Сравнение эквивалентности

Как и в анонимных типах, метод Equals выполняет структурное сравнение эквивалентности, т.е. сравнивает *данные*, а не *ссылки*:

```
var t1 = ("one", 1);
var t2 = ("one", 1);
Console.WriteLine (t1.Equals (t2)); // True
```

Кроме того, типы ValueTuple<> перегружают операции == и !=:

```
Console.WriteLine (t1 == t2);      // True (начиная с версии C# 7.3)
```

Кортежи также переопределяют метод GetHashCode, делая практическим использование кортежей в качестве ключей в словарях. Сравнение эквивалентно

сти подробно рассматривается в разделе “Сравнение эквивалентности” главы 6, а словари — в главе 7.

Типы `ValueTuple`<> также реализуют интерфейс `IComparable` (см. раздел “Сравнение порядка” в главе 6), делая возможным применение кортежей как ключей сортировки.

Классы `System.Tuple`

В пространстве имен `System` вы обнаружите еще одно семейство обобщенных типов под названием `Tuple` (не `ValueTuple`). Они появились еще в 2010 году и были определены как классы (тогда как типы `ValueTuple` являются структурами). Оглядываясь назад, можно сказать, что определение кортежей как классов было заблуждением: в сценариях, в которых обычно используются кортежи, структуры дают небольшое преимущество в плане производительности (за счет избегания излишних выделений памяти), практически не имея недостатков. Поэтому при добавлении языковой поддержки для кортежей в версии C# 7 существующие типы `Tuple` были проигнорированы в пользу новых типов `ValueTuple`. Вы по-прежнему можете встречать классы `Tuple` в коде, написанном до выхода C# 7. Они не располагают специальной языковой поддержкой и применяются следующим образом:

```
Tuple<string,int> t = Tuple.Create ("Bob", 23);           // Фабричный метод
Console.WriteLine (t.Item1);                                // Bob
Console.WriteLine (t.Item2);                                // 23
```

Записи

Запись (*record*) представляет собой особый вид класса, который предназначен для эффективной работы с неизменяемыми (допускающими только чтение) данными. Наиболее полезной характеристикой записей является *неразрушающее изменение*, но их также удобно использовать для создания типов, которые просто объединяют или хранят данные. В простых случаях записи позволяют избавиться от стереотипного кода и одновременно соблюдают семантику эквивалентности, наиболее подходящую для неизменяемых типов.

Записи представляют собой конструкцию C#, существующую только на этапе компиляции. Во время выполнения среда CLR видит их просто как классы или структуры (с группой дополнительных “синтезированных” членов, добавленных компилятором).

Подоплека

Реализация неизменяемых типов (поля которых не могут быть модифицированы после инициализации) — популярная стратегия упрощения программного обеспечения и уменьшения количества дефектов. Она также является основным аспектом функционального программирования, при котором избегают изменяемого состояния, а функции трактуются как данные. Такой принцип был мотивом появления LINQ.

Чтобы “модифицировать” неизменяемый объект, потребуется создать новый объект и скопировать в него данные, включая модификации (что называется *неразрушающим изменением*). С точки зрения производительности это не настолько неэффективно, как можно было бы ожидать, поскольку всегда будет достаточно *поверхностной копии* (глубокая копия, предусматривающая копирование также подобъектов и коллекций, не нужна, если данные неизменяемы). Но с точки зрения усилий по написанию кода реализация неразрушающего изменения может оказаться крайне неэффективной, особенно при наличии множества свойств. Записи решают проблему посредством шаблона, поддерживающего самим языком.

Вторая проблема связана с тем, что программисты — особенно занимающиеся *функциональным программированием* — временами используют неизменяемые типы только для объединения данных (не добавляя линии поведения). Определение таких типов сопряжено с большими трудозатратами и требует написания конструктора для присваивания каждого параметра каждому открытому свойству (деконструктор тоже может оказаться полезным). Благодаря записям работа подобного рода возлагается на компилятор.

Наконец, одно из последствий неизменяемости объекта заключается в том, что его идентичность не может меняться, а потому для таких типов удобнее реализовывать *структурную эквивалентность*, нежели *сырьенную эквивалентность*. Структурная эквивалентность означает, что два экземпляра будут одинаковыми, если одинаковы их данные (как в случае кортежей). По умолчанию записи обеспечивают структурную эквивалентность — независимо от того, является базовый тип классом или структурой — без какого-либо стереотипного кода.

Определение записи

Определение записи похоже на определение класса или структуры и может содержать такие же виды членов, включая поля, свойства, методы и т.д. Записи могут реализовывать интерфейсы, а (основанные на классах) записи могут быть подклассами других (основанных на классах) записей.

По умолчанию типом, лежащим в основе в записи, является класс:

```
record Point { } // Point является классом
```

Начиная с версии C# 10, типом, лежащим в основе в записи, также может быть структура:

```
record struct Point { } // Point является структурой
```

(Разрешено определение record class, которое представляет собой то же самое, что и record.)

Простая запись может содержать лишь набор свойств, допускающих только инициализацию, и вполне вероятно конструктор:

```
record Point
{
    public Point (double x, double y) => (X, Y) = (x, y);
    public double X { get; init; }
    public double Y { get; init; }
}
```



В конструкторе задействовано сокращение, которое было описано в предыдущем разделе; показанный ниже код:

```
(X, Y) = (x, y);  
эквивалентен (в данном случае) следующему коду:  
{ this.X = x; this.Y = y; }
```

Компилятор C# трансформирует определение записи в класс (или структуру) и выполняет перечисленные ниже дополнительные шаги:

- реализует защищенный *конструктор копирования* (и скрытый метод *клоирования*), способствуя неразрушающему изменению;
- переопределяет/перегружает функции, связанные с эквивалентностью, для реализации структурной эквивалентности;
- переопределяет метод *ToString* (для расширения открытых свойств записи, как в случае анонимных типов).

Предыдущее определение записи расширяется примерно так:

```
class Point  
{  
    public Point (double x, double y) => (X, Y) = (x, y);  
    public double X { get; init; }  
    public double Y { get; init; }  
  
    protected Point (Point original) // "Конструктор копирования"  
    {  
        this.X = original.X; this.Y = original.Y  
    }  
  
    // Этот метод имеет странное имя, сгенерированное компилятором:  
    public virtual Point <Clone>$() => new Point (this); // Метод  
                                            // клоирования  
  
    // Дополнительный код для переопределения  
    // Equals, ==, !=, GetHashCode, ToString  
    // ...  
}
```



Хотя ничто не может воспрепятствовать добавлению к конструктору *необязательных параметров*, общепринятый шаблон (по крайней мере, в открытых библиотеках) предусматривает вынесение их за пределы конструктора и реализацию в виде свойств, допускающих только инициализацию:

```
new Foo (123, 234) { Optional2 = 345 };  
record Foo  
{  
    public Foo (int required1, int required2) { ... }  
    public int Required1 { get; init; }  
    public int Required2 { get; init; }  
    public int Optional1 { get; init; }  
    public int Optional2 { get; init; }  
}
```

Преимущество этого шаблона заключается в том, что позже вы сможете безопасно добавлять свойства только для инициализации, не нарушая двоичной совместимости с потребителями, которые были скомпилированы с более старыми версиями вашей сборки.

Списки параметров

Определение записи может быть сокращено посредством *списка параметров*:

```
record Point (double X, double Y)
{
    // Здесь можно определить дополнительные члены класса...
}
```

Параметры могут иметь модификаторы *in* и *params*, но не *out* или *ref*. Если список параметров указан, тогда компилятор выполняет следующие дополнительные шаги:

- реализует для каждого параметра свойство, допускающее только инициализацию;
- реализует *основной конструктор* для заполнения свойств;
- реализует деконструктор.

Это значит, что если мы объявим запись *Point* просто как

```
record Point (double X, double Y);
```

то компилятор в итоге сгенерирует (почти) в точности то, что было показано ранее в расширении определения записи. Небольшое отличие связано с тем, что основной конструктор будет иметь параметры *X* и *Y*, но не *x* и *y*:

```
public Point (double X, double Y)          // "Основной конструктор"
{
    this.X = X; this.Y = Y;
}
```



Кроме того, из-за принадлежности к основному конструктору параметры *X* и *Y* “магическим” образом становятся доступными любым инициализаторам полей или свойств в вашей записи. Мы обсудим все тонкости в разделе “Основные конструкторы” далее в главе.

Еще одно отличие при определении списка параметров связано с тем, что компилятор также генерирует деконструктор:

```
public void Deconstruct (out double X, out double Y) // Деконструктор
{
    X = this.X; Y = this.Y;
}
```

С применением следующего синтаксиса можно создавать подклассы записей со списками параметров:

```
record Point3D (double X, double Y, double Z) : Point (X, Y);
```

Компилятор тогда выпустит основной конструктор такого вида:

```
class Point3D : Point
{
    public double Z { get; init; }
    public Point3D (double X, double Y, double Z) : base (X, Y)
        => this.Z = Z;
}
```



Списки параметров предлагают удобный сокращенный прием на тот случай, когда необходим класс, который просто объединяет вместе набор значений (*тип-произведение* в функциональном программировании), и также могут быть полезны для создания прототипов. Как будет показано позже, они не настолько удобны, когда к средствам доступа только для инициализации нужно добавлять логику (вроде проверки достоверности аргументов).

Изменяемость с помощью структур типа записей

При определении списка параметров в структуре типа записи компилятор генерирует свойства, поддерживающие запись, вместо свойств, доступных только для инициализации, если только объявление записи не предварено префиксом `readonly`:

```
readonly record struct Point (double X, double Y);
```

Обоснование заключается в том, что в типовых случаях использования преимущества неизменяемости в плане безопасности возникают не из-за того, что *структуря неизменяема*, а по той причине, что ее *исходная часть является неизменяемой*. В следующем примере нельзя изменять поле `X`, хотя оно доступно для записи:

```
var test = new Immutable();
test.Field.X++; // Запрещено, т.к. Field допускает только чтение
test.Prop.X++; // Запрещено, т.к. в Prop определено только средство {get;}
class Immutable
{
    public readonly Mutable Field;
    public Mutable Prop { get; }
}
struct Mutable { public int X, Y; }
```

А пока можно было бы поступить следующим образом:

```
var test = new Immutable();
Mutable m = test.Prop;
m.X++;
```

Все, чего удастся добиться — это изменить локальную переменную (*копию* `test.Prop`). Изменение локальной переменной может быть полезной оптимизацией и не отменяет преимуществ системы неизменяемых типов.

И наоборот, если сделать `Field` доступным для записи полем, а `Prop` — доступным для записи свойством, то можно было бы просто заменить их содержимое вне зависимости от способа объявления структуры `Mutable`.

Неразрушающее изменение

Самым важным шагом, который компилятор выполняет со всеми записями, считается реализация **конструктора копирования** (и скрытого метода **клонирования**), что делает возможным неразрушающее изменение через ключевое слово **with**:

```
Point p1 = new Point (3, 3);
Point p2 = p1 with { Y = 4 };
Console.WriteLine (p2);           // Point { X = 3, Y = 4 }
record Point (double X, double Y);
```

В приведенном примере **p2** является копией **p1**, но с его свойством **Y**, установленным в 4. Преимущество становится более явным при наличии большего количества свойств:

```
Test t1 = new Test (1, 2, 3, 4, 5, 6, 7, 8);
Test t2 = t1 with { A = 10, C = 30 };
Console.WriteLine (t2);
record Test (int A, int B, int C, int D, int E, int F, int G, int H);
```

Вот вывод:

```
Test { A = 10, B = 2, C = 30, D = 4, E = 5, F = 6, G = 7, H = 8 }
```

Неразрушающее изменение происходит в два этапа.

Сначала **конструктор копирования** клонирует запись. По умолчанию он копирует каждое внутреннее поле записи, создавая точную копию и минуя накладные расходы на выполнение любой логики в средствах доступа только для инициализации. Включаются все поля (открытые и закрытые, а также скрытые поля, которые поддерживают автоматические свойства).

Затем обновляется каждое свойство в **списке инициализаторов членов** (на этот раз с использованием средств доступа только для инициализации).

Компилятор транслирует следующий код:

```
Test t2 = t1 with { A = 10, C = 30 };
```

в примерно такой функционально эквивалентный вариант:

```
Test t2 = new Test(t1);          // Использовать конструктор копирования для
                                // клонирования t1 поле за полем
t2.A = 10;                      // Обновить свойство A
t2.C = 30;                      // Обновить свойство C
```

(Тот же самый код не скомпилируется, если вы напишете его явно, поскольку **A** и **C** — свойства, допускающие только инициализацию. Кроме того, конструктор копирования является **зацищенным**; компилятор C# обходит это за счет его вызова через скрытый метод **public** по имени **<Clone>\$**, реализованный для записи.) При необходимости вы можете определять собственный конструктор копирования. Тогда компилятор C# будет применять ваше определение, не генерируя свое:

```
protected Point (Point original)
{
    this.X = original.X; this.Y = original.Y;
}
```

Писать специальный конструктор копирования может быть полезно, когда ваша запись содержит изменяемые подобъекты или коллекции, которые вы хотите клонировать, либо при наличии вычисляемых полей, которые желательно очистить. К сожалению, стандартную реализацию можно только заменять, но не расширять.



При создании подкласса другой записи конструктор копирования несет ответственность за копирование только собственных полей. Чтобы скопировать поля базовой записи, делегируйте ей работу:

```
protected Point (Point original) : base (original)
{
    ...
}
```

Проверка достоверности свойств

С помощью явных свойств логику проверки достоверности можно помещать внутрь средств доступа только для инициализации. В приведенном ниже примере мы гарантируем, что X не может быть NaN:

```
record Point
{
    // Обратите внимание, что мы присваиваем x свойству X (не полю _x):
    public Point (double x, double y) => (X, Y) = (x, y);

    double _x;
    public double X
    {
        get => _x;
        init
        {
            if (double.IsNaN (value))
                throw new ArgumentException ("X Cannot be NaN");
                // X не может быть NaN
            _x = value;
        }
    }
    public double Y { get; init; }
}
```

Наше решение обеспечивает выполнение проверки достоверности как при конструировании, так и во время неразрушающего изменения объекта:

```
Point p1 = new Point (2, 3);
Point p2 = p1 with { X = double.NaN }; // Генерируется исключение
```

Вспомните, что генерируемый компилятором *конструктор копирования* копирует все поля и автоматические свойства. Это означает, что сгенерированный конструктор копирования теперь будет выглядеть примерно так:

```
protected Point (Point original)
{
    _x = original._x; Y = original.Y;
}
```

Обратите внимание, что копирование поля `_x` обходит средство доступа свойства `X`. Тем не менее, ничего не нарушается, т.к. в точности копируется объект, который уже был благополучно заполнен через средство доступа только для инициализации свойства `X`.

Вычисляемые поля и ленивая оценка

Популярным шаблоном функционального программирования, который хорошо работает с неизменяемыми типами, является **ленивая оценка**, когда значение не вычисляется до тех пор, пока оно не потребуется, и затем кешируется для многократного потребления. Предположим, например, что мы хотим определить в записи `Point` свойство, которое возвращает расстояние от начала координат $(0, 0)$:

```
record Point (double X, double Y)
{
    public double DistanceFromOrigin => Math.Sqrt (X*X + Y*Y);
}
```

Давайте теперь попробуем провести рефакторинг, чтобы избежать затрат на повторное вычисление значения `DistanceFromOrigin` при каждом обращении к свойству. Мы начнем с удаления списка свойств и определения `X`, `Y` и `DistanceFromOrigin` как свойств только для чтения. Затем мы можем вычислять `DistanceFromOrigin` в конструкторе:

```
record Point
{
    public double X { get; }
    public double Y { get; }
    public double DistanceFromOrigin { get; }
    public Point (double x, double y) =>
        (X, Y, DistanceFromOrigin) = (x, y, Math.Sqrt (x*x + y*y));
}
```

Прием работает, но он не поддерживает неразрушающее изменение (изменение `X` и `Y` на свойства, допускающие только инициализацию, нарушит работу кода, поскольку свойство `DistanceFromOrigin` будет устаревать после выполнения средств доступа только для инициализации). Решение также не оптимально в том, что вычисление всегда выполняется вне зависимости от того, читается ли в принципе свойство `DistanceFromOrigin`. Оптимальное решение предусматривает кеширование значения свойства в поле и его заполнение **ленивым образом** (при первом использовании):

```
record Point
{
    ...
    double? _distance;
    public double DistanceFromOrigin
    {
        get
        {
            if (_distance == null)
```



```
_distance = Math.Sqrt (X*X + Y*Y);  
return _distance.Value;
```

Формально в показанном выше коде мы *изменяем* `_distance`. Однако тип `Point` по-прежнему законно называть неизменяемым. Изменение поля исключительно для заполнения значения, оцениваемого ленивым образом, не делает недействительными принципы или преимущества неизменяемости и даже может быть замаскировано с помощью типа `Lazy<T>`, который рассматривается в главе 21.

Благодаря операции присваивания с объединением с null (??=) языка C# мы можем сократить объявление свойства до одной строки кода:

```
public double DistanceFromOrigin => _distance ??= Math.Sqrt (X*X + Y*Y);
```

(Код означает следующее: возвратить значение `_distance`, если оно не равно `null`, а иначе возвратить значение `Math.Sqrt (X*X + Y*Y)` и одновременно присвоить его `distance`.)

Для работы со свойствами, допускающими только инициализацию, необходим еще один шаг — очистка кешированного поля `_distance`, когда X или Y обновляется через средство доступа `init`. Ниже приведен полный код:

```
record Point
{
    public Point (double x, double y) => (X, Y) = (x, y);
    double _x, _y;
    public double X { get => _x; init { _x = value; _distance = null; } }
    public double Y { get => _y; init { _y = value; _distance = null; } }
    double? _distance;
    public double DistanceFromOrigin => _distance ??= Math.Sqrt (X*X + Y*Y);
}
```

Теперь запись Point можно изменять неразрушающим образом:

```
Point p1 = new Point (2, 3);
Console.WriteLine (p1.DistanceFromOrigin);           // 3.605551275463989
Point p2 = p1 with { Y = 4 };
Console.WriteLine (p2.DistanceFromOrigin);           // 4.47213595499958
```

Приятный бонус заключается в том, что автоматически сгенерированный конструктор копирования переписывает кешированное поле `_distance`. Это означает, что если запись имеет другие свойства, которые не участвуют в вычислении, тогда неразрушающее изменение таких свойств не приведет к нежелательной утрате кешированного значения. Если указанный бонус вас не интересует, то альтернативой очистке кешированного значения в средствах доступа `init` является реализация специального конструктора копирования, который игнорирует кешированное поле. Код будет более лаконичным, потому что он работает со списками параметров, к тому же специальный конструктор копирования может задействовать деконструктор:

```
record Point (double X, double Y)
{
    double? _distance;
    public double DistanceFromOrigin => _distance ??= Math.Sqrt (X*X + Y*Y);
    protected Point (Point other) => (X, Y) = other;
}
```

Имейте в виду, что в любом решении добавление полей, вычисляемых ленивым образом, нарушает стандартное сравнение структурной эквивалентности (поскольку такие поля могут заполняться, а могут и не заполняться), хотя вскоре будет показано, что это относительно легко исправить.

Основные конструкторы

При определении записи со списком параметров компилятор автоматически генерирует объявления свойств, а также *основной конструктор* (и деконструктор). Как демонстрировалось ранее, прием хорошо работал в простых случаях, а в более сложных ситуациях список параметров можно было опускать и записывать код объявления свойств и конструктора вручную.

В языке C# также предлагается умеренно полезный промежуточный вариант (если вы готовы иметь дело с необычной семантикой основных конструкторов), который предусматривает определение списка параметров и самостоятельное написание отдельных или всех объявлений свойств:

```
record Student (string ID, string LastName, string GivenName)
{
    public string ID { get; } = ID;
}
```

В этом случае мы “взяли на себя” объявление свойства ID, определив его как допускающее только чтение (а не только инициализацию), что предотвращает его участие в неразрушающем изменении. Если вам не нужно изменять специфическое свойство неразрушающим образом, то его превращение в допускающее только чтение позволяет хранить вычисляемые данные в записи без необходимости в написании кода механизма обновления.

Обратите внимание, что мы должны включить *инициализатор свойства* (выделенный полужирным):

```
public string ID { get; } = ID;
```

Когда вы берете на себя объявление свойства, то также отвечаете за инициализацию его значения; основной конструктор больше не делает ее автоматически. (Это в точности соответствует поведению при определении основных конструкторов в классах или структурах.) Кроме того, имейте в виду, что выделенный полужирным ID относится к *параметру основного конструктора*, в не к свойству ID.



Благодаря структурам типа записей появляется законная возможность переопределить свойство как поле:

```
record struct Student (string ID)
{
    public string ID = ID;
}
```

В соответствии с семантикой основных конструкторов классов и структур (см. раздел “Основные конструкторы” ранее в главе), параметры основного конструктора (в данном случае `ID`, `LastName` и `GivenName`) “магическим” образом доступны всем инициализаторам полей и свойств. Мы можем проиллюстрировать сказанное, расширив пример:

```
record Student (string ID, string LastName, string FirstName)
{
    public string ID { get; } = ID;
    readonly int _enrollmentYear = int.Parse (ID.Substring (0, 4));
}
```

И снова выделенный полужирным идентификатор `ID` относится к параметру основного конструктора, а не к свойству. (Причина отсутствия неоднозначности заключается в том, что доступ к свойствам из инициализаторов незаконен.)

В показанном примере мы вычисляем `_enrollmentYear` по первым четырем цифрам `ID`. Хотя безопасно хранить это в поле только для чтения (т.к. свойство `ID` допускает только чтение и потому не может быть изменено неразрушающим образом), в реальности данный код не будет работать настолько же хорошо. Дело в том, что из-за отсутствия явного конструктора нет центрального места для проверки достоверности значения `ID` и генерации содержательного исключения, если оно недопустимо (общее требование).

Проверка достоверности также является веской причиной написания явных средств доступа только для инициализации (как обсуждалось в разделе “Проверка достоверности свойств” ранее в главе). К сожалению, в таком сценарии основные конструкторы не работают надлежащим образом. В целях иллюстрации рассмотрим следующую запись, где средство доступа `init` выполняет проверку на предмет равенства `null`:

```
record Person (string Name)
{
    string _name = Name;
    public string Name
    {
        get => _name;
        init => _name = value ?? throw new ArgumentNullException ("Name");
    }
}
```

Поскольку `Name` — не автоматическое свойство, для него нельзя определить инициализатор. Лучшее, что мы можем предпринять — это обеспечить инициализатор для поддерживающего поля (выделен полужирным). К несчастью, в таком случае пропускается проверка на предмет равенства `null`:

```
var p = new Person (null);      // Успешно! (Проверка на равенство null
                                // пропускается)
```

Сложность в том, что нет способа присвоить параметр основного конструктора свойству, не написав код самого конструктора. Несмотря на существование обходных путей (вроде вынесения логики проверки достоверности из средства доступа `init` в отдельный статический метод, который вызывается дважды), самый простой прием заключается в том, чтобы полностью отказаться от спи-

ска параметров и реализовать обычный конструктор вручную (а также при необходимости деструктор):

```
record Person
{
    public Person (string name) => Name = name; // Присвоить значение *СВОЙСТВУ*
    string _name;
    public string Name { get => _name; init => ... }
}
```

Записи и сравнение эквивалентности

Подобно структурам, анонимным типам и кортежам записи обеспечивают структурную эквивалентность в готовом виде, т.е. две записи равны, если равны их поля (и автоматические свойства):

```
var p1 = new Point (1, 2);
var p2 = new Point (1, 2);
Console.WriteLine (p1.Equals (p2));           // True
record Point (double X, double Y);
```

Операция эквивалентности работает также и с записями (как с кортежами):

```
Console.WriteLine (p1 == p2);                // True
```

Стандартная реализация эквивалентности для записей неизбежно оказывается хрупкой. В частности, ее работа нарушается, если запись содержит значения, вычисляемые ленивым образом, переходные значения, массивы или коллекции (которые требуют специальной обработки при сравнении эквивалентности). К счастью, ситуацию относительно легко исправить (когда нужно, чтобы сравнение эквивалентности работало), причем объем работы будет меньше, чем при добавлении полноценного поведения эквивалентности в классы или структуры.

В отличие от классов и структур вы не переопределяете метод `object.Equals` (и не можете это делать), а замен определяете открытый метод `Equals` со следующей сигнатурой:

```
record Point (double X, double Y)
{
    double _someOtherField;
    public virtual bool Equals (Point other) =>
        other != null && X == other.X && Y == other.Y;
}
```

Метод `Equals` должен быть `virtual` (не `override`) и строго типизированным, чтобы принимать фактический тип записи (в данном случае `Point`, не `object`). Как только вы обеспечите корректную сигнатуру, компилятор будет автоматически вставлять ваш метод.

В рассматриваемом примере мы изменяем логику эквивалентности, так что сравниваются только `X` и `Y` (а `_someOtherField` игнорируется).

В случае создания подкласса другой записи вы можете вызывать метод `base.Equals`:

```
public virtual bool Equals (Point other) => base.Equals (other) && ...
```

Как и с любым типом, если вы занялись сравнением эквивалентности, то также должны переопределить метод GetHashCode. Приятная особенность записей в том, что вы не перегружаете операцию != или == и не реализуете интерфейс IEquatable<T>: все это сделано за вас. Мы полностью раскроем тему сравнения эквивалентности в разделе “Сравнение эквивалентности” главы 6.

Шаблоны

В главе 3 было продемонстрировано, как использовать операцию is для проверки, будет ли успешным ссылочное преобразование:

```
if (obj is string)
    Console.WriteLine (((string) obj).Length);
```

Или более лаконично:

```
if (obj is string s)
    Console.WriteLine (s.Length);
```

Здесь задействован так называемый *шаблон типа*. Операция is также поддерживает другие шаблоны, которые появились в недавних версиях C#, наподобие *шаблона свойства*:

```
if (obj is string { Length: 4 })
    Console.WriteLine ("A string with 4 characters");
```

Шаблоны поддерживаются в следующих контекстах:

- после операции is (переменная is шаблон);
- в операторах switch;
- в выражениях switch.

Мы уже раскрывали шаблон типа (и вкратце шаблон кортежа) в разделах “Переключение по типам” главы 2 и “Операция is” главы 3. В настоящем разделе мы рассмотрим более сложные шаблоны, которые были введены в последних версиях C#.

Некоторые из более специализированных шаблонов предназначены главным образом для применения в операторах/выражениях switch, где они уменьшают потребность в конструкциях when и позволяют использовать switch там, где раньше делать это было нельзя.



Шаблоны, обсуждаемые в настоящем разделе, в небольшой или умеренной степени полезны в ряде сценариев. Помните, что вы всегда можете заменить сильно шаблонизированные выражения switch простыми операторами if — или в некоторых случаях тернарной условной операцией — и часто без значительного объема добавочного кода.

Шаблон константы

Шаблон константы позволяет выполнять сопоставление непосредственно с константой; он удобен при работе с типом `object`:

```
void Foo (object obj)
{
    if (obj is 3) ...
}
```

Выражение, выделенное полужирным, эквивалентно такому выражению:

```
obj is int && (int) obj == 3
```

(Из-за статической типизации компилятор C# не разрешит использовать операцию `==` для сравнения экземпляра `object` напрямую с константой, т.к. ему заранее должны быть известны типы.)

Сам по себе данный шаблон не особо полезен, потому что существует разумная альтернатива:

```
if (3.Equals (obj)) ...
```

Как вскоре будет показано, шаблон константы может стать более полезным благодаря комбинаторам шаблонов.

Реляционные шаблоны

Начиная с версии C# 9, в шаблонах можно применять операции `<`, `>`, `<=` и `>=`:

```
if (x is > 100) Console.WriteLine ("x is greater than 100"); // x больше 100
```

Еще большее удобство это приносит в `switch`:

```
string GetWeightCategory (decimal bmi) => bmi switch
{
    < 18.5m => "underweight",           // недостаток в весе
    < 25m    => "normal",                // нормальный вес
    < 30m    => "overweight",            // избыточный вес
    >=        => "obese"                 // ожирение
};
```

Реляционные шаблоны становятся даже более полезными в сочетании с комбинаторами шаблонов.



Реляционный шаблон также работает в ситуации, когда на этапе компиляции переменная имеет тип `object`, но вы должны быть крайне осторожными при использовании числовых констант. В следующем примере последняя строка кода выводит `False`, т.к. там предпринимается попытка сопоставления десятичного значения с целочисленным литералом:

```
object obj = 2m;                      // obj - десятичное значение
Console.WriteLine (obj is < 3m); // True
Console.WriteLine (obj is < 3); // False
```

Комбинаторы шаблонов

Начиная с версии C# 9, можно применять ключевые слова `and`, `or` и `not` для объединения шаблонов:

```
bool IsJanetOrJohn (string name) => name.ToUpper() is "JANET" or "JOHN";
bool IsVowel (char c) => c is 'a' or 'e' or 'i' or 'o' or 'u';
bool Between1And9 (int n) => n is >= 1 and <= 9;
bool IsLetter (char c) => c is >= 'a' and <= 'z'
                           or >= 'A' and <= 'Z';
```

Подобно операциям `&&` и `||` комбинатор `and` имеет более высокий приоритет, чем комбинатор `or`. Вы можете переопределить это с помощью круглых скобок.

Интересным трюком будет объединение комбинатора `not` с шаблоном типа для проверки, не относится ли объект к указанному типу:

```
if (obj is not string) ...
```

Такое решение выглядит изящнее следующего:

```
if (!(obj is string)) ...
```

Шаблон `var`

Шаблон `var` является вариацией шаблона *типа*, в котором имя типа заменено ключевым словом `var`. Такое преобразование всегда проходит успешно, а потому его цель — просто позволить повторно использовать переменную, следующую за `var`:

```
bool IsJanetOrJohn (string name) =>
    name.ToUpper() is var upper && (upper == "JANET" || upper == "JOHN");
```

Ниже представлен эквивалентный код:

```
bool IsJanetOrJohn (string name)
{
    string upper = name.ToUpper();
    return upper == "JANET" || upper == "JOHN";
}
```

Возможность введения и применения промежуточной переменной (в данном случае `upper`) в методах, сжатых до выражений, весьма удобна — в частности в лямбда-выражениях. К сожалению, она работает только когда метод возвращает тип `bool`.

Шаблоны кортежей и позиционные шаблоны

Шаблон кортежа (появившийся в версии C# 8) позволяет выполнять сопоставление с кортежем:

```
var p = (2, 3);
Console.WriteLine (p is (2, 3)); // True
```

Вы можете использовать его для переключения по множеству значений:

```
int AverageCelsiusTemperature (Season season, bool daytime) =>
    (season, daytime) switch
    {
        (Season.Spring, true) => 20,
        (Season.Spring, false) => 16,
        (Season.Summer, true) => 27,
        (Season.Summer, false) => 22,
        (Season.Fall, true) => 18,
        (Season.Fall, false) => 12,
        (Season.Winter, true) => 10,
        (Season.Winter, false) => -2,
        _ => throw new Exception ("Unexpected combination")
            // Непредвиденная комбинация
    };
enum Season { Spring, Summer, Fall, Winter };
```

Шаблон кортежа можно считать особым случаем *позиционного шаблона* (C# 8+), который обеспечивает сопоставление с любым типом, имеющим метод Deconstruct (см. раздел “Деконструкторы” в главе 3). В приведенном ниже примере задействован генерированный компилятором деконструктор записи Point:

```
var p = new Point (2, 2);
Console.WriteLine (p is (2, 2)); // True
record Point (int X, int Y); // Имеет генерированный компилятором деконструктор
```

В ходе сопоставления деконструировать можно с использованием следующего синтаксиса:

```
Console.WriteLine (p is (var x, var y) && x == y); // True
```

Вот выражение switch, которое объединяет шаблон типа с позиционным шаблоном:

```
string Print (object obj) => obj switch
{
    Point (0, 0) => "Empty point",
    Point (var x, var y) when x == y => "Diagonal"
    ...
};
```

Шаблоны свойств

Шаблон свойства (C# 8+) соответствует одному или большему количеству значений свойств объекта. Ранее мы приводили простой пример в контексте операции is:

```
if (obj is string { Length: 4 }) ...
```

Тем не менее, экономия получается не особо большая по сравнению со следующим подходом:

```
if (obj is string s && s.Length == 4) ...
```

Шаблоны свойств более полезны с операторами и выражениями switch. Возьмем класс System.Uri, который представляет URI. Он имеет свойства, включающие Scheme, Host, Port и IsLoopback. При реализации брандмауэра мы могли бы принимать решение о том, разрешать или блокировать URI, с применением выражения switch, в котором используются шаблоны свойств:

```
bool ShouldAllow (Uri uri) => uri switch
{
    { Scheme: "http", Port: 80 } => true,
    { Scheme: "https", Port: 443 } => true,
    { Scheme: "ftp", Port: 21 } => true,
    { IsLoopback: true } => true,
    => false
};
```

Свойства можно вкладывать друг в друга, делая законной следующую конструкцию:

```
{ Scheme: { Length: 4 }, Port: 80 } => true,
```

которую, начиная с версии C# 10, можно упростить:

```
{ Scheme.Length: 4, Port: 80 } => true,
```

Внутри шаблонов свойств можно применять другие шаблоны, включая реляционный шаблон:

```
{ Host: { Length: < 1000 }, Port: > 0 } => true,
```

Более сложные условия могут выражаться посредством конструкции when:

```
{ Scheme: "http" } when string.IsNullOrWhiteSpace (uri.Query) => true,
```

Можно также объединять шаблон свойства с шаблоном типа:

```
bool ShouldAllow (object uri) => uri switch
{
    Uri { Scheme: "http", Port: 80 } => true,
    Uri { Scheme: "https", Port: 443 } => true,
    ...
}
```

Как и можно было ожидать от шаблонов типов, в конце конструкции допускается вводить переменную и затем пользоваться ею:

```
Uri { Scheme: "http", Port: 80 } httpUri => httpUri.Host.Length < 1000,
```

Эту переменную можно также использовать в конструкции when:

```
Uri { Scheme: "http", Port: 80 } httpUri
      when httpUri.Host.Length < 1000 => true,
```

С шаблонами свойств связан несколько причудливый трюк — вводить переменные можно также на уровне свойств:

```
{ Scheme: "http", Port: 80, Host: string host } => host.Length < 1000,
```

Поскольку разрешена неявная типизация, string можно заменить var. Вот полный пример:

```
bool ShouldAllow (Uri uri) => uri switch
{
    { Scheme: "http", Port: 80, Host: var host } => host.Length < 1000,
    { Scheme: "https", Port: 443 } => true,
    { Scheme: "ftp", Port: 21 } => true,
    { IsLoopback: true } => true,
    => false
};
```

Придумать примеры, в которых такой прием сэкономил бы значительное количество символов, не особенно легко. В нашем случае альтернатива на самом деле короче:

```
{ Scheme: "http", Port: 80 } => uri.Host.Length < 1000 => ...
```

Или:

```
{ Scheme: "http", Port: 80, Host: { Length: < 1000 } } => ...
```

Шаблоны списков

Шаблоны списков (введенные в версии C# 11) работают с коллекцией любого вида, которая поддерживает подсчет (с помощью свойства Count или Length) и индексацию (с помощью индексатора типа int или System.Index).

Шаблон списка соответствует последовательности элементов в квадратных скобках:

```
int[] numbers = { 0, 1, 2, 3, 4 };
Console.Write (numbers is [0, 1, 2, 3, 4]); // True
```

Подчеркивание соответствует одному элементу любого значения:

```
Console.Write (numbers is [0, 1, _, _, 4]); // True
```

Шаблон var также работает при сопоставлении с одним элементом:

```
Console.Write (numbers is [0, 1, var x, 3, 4] && x > 1); // True
```

Две точки обозначают срез. Срез соответствует нулю или большему количеству элементов:

```
Console.Write (numbers is [0, .., 4]); // True
```

Массивы и другие типы, поддерживающие индексы и диапазоны (см. раздел “Индексы и диапазоны” в главе 2), позволяют следовать за срезом с помощью шаблона var:

```
Console.Write (numbers is [0, .. var mid, 4] && mid.Contains (2)); // True
```

Шаблон списка может включать не более одного среза.

Атрибуты

Вам уже знакомо понятие снабжения элементов кода признаками в форме модификаторов, таких как virtual или ref. Эти конструкции встроены в язык. Атрибуты представляют собой расширяемый механизм для добавления специ-

альной информации к элементам кода (сборкам, типам, членам, возвращаемым значениям и параметрам обобщенных типов). Такая расширяемость удобна для служб, глубоко интегрированных в систему типов, и не требует специальных ключевых слов или конструкций в языке C#.

Хороший сценарий для атрибутов связан с *сериализацией* — процессом преобразования произвольных объектов в и из определенного формата с целью хранения или передачи. В таком сценарии атрибут на поле может указывать трансляцию между представлением поля в C# и его представлением в используемом формате.

Классы атрибутов

Атрибут определяется классом, который унаследован (прямо или косвенно) от абстрактного класса `System.Attribute`. Чтобы присоединить атрибут к элементу кода, перед элементом необходимо указать имя типа атрибута в квадратных скобках. Например, в показанном ниже коде к классу `Foo` присоединяется атрибут `ObsoleteAttribute`:

```
[ObsoleteAttribute]  
public class Foo { ... }
```

Данный атрибут распознается компилятором и приводит к тому, что компилятор выдает предупреждение, если производится ссылка на тип или член, помеченный как устаревший (`obsolete`). По соглашению имени всех типов атрибутов заканчиваются на слово `Attribute`. Такое соглашение поддерживается компилятором C# и позволяет опускать суффикс `Attribute`, когда присоединяется атрибут:

```
[Obsolete]  
public class Foo { ... }
```

Тип `ObsoleteAttribute` объявлен в пространстве имен `System` следующим образом (для краткости код упрощен):

```
public sealed class ObsoleteAttribute : Attribute { ... }
```

Библиотеки .NET включают множество предопределенных атрибутов. В главе 18 будет объясняться, как создавать собственные атрибуты.

Именованные и позиционные параметры атрибутов

Атрибуты могут иметь параметры. В показанном далее примере мы применяем к классу атрибут `XmlAttributeAttribute`. Он указывает сериализатору XML (из пространства имен `System.Xml.Serialization`), каким образом объект представлен в XML, и принимает несколько *параметров атрибута*. Следующий атрибут отображает класс `CustomerEntity` на XML-элемент по имени `Customer`, принадлежащий пространству имен `http://oreilly.com`:

```
[XmlAttribute ("Customer", Namespace="http://oreilly.com")]  
public class CustomerEntity { ... }
```

(Сериализация XML и JSON описана в дополнительных материалах книги, доступных на веб-сайте издательства.)

Параметры атрибутов относятся к одной из двух категорий: *позиционные* либо *именованные*. В предыдущем примере первый аргумент является позиционным параметром, а второй — именованным параметром. Позиционные параметры соответствуют параметрам открытых конструкторов типа атрибута. Именованные параметры соответствуют открытым полям или открытым свойствам типа атрибута.

При указании атрибута должны включаться позиционные параметры, которые соответствуют одному из конструкторов класса атрибута. Именованные параметры необязательны.

В главе 18 будут описаны допустимые типы параметров и правила, используемые для их проверки.

Применение атрибутов к сборкам и поддерживающим полям

Неявно целью атрибута является элемент кода, находящийся непосредственно за атрибутом, который обычно представляет собой тип или член типа. Тем не менее, атрибуты можно присоединять и к сборке. При этом требуется явно указывать цель атрибута. Вот как использовать атрибут AssemblyFileVersion для присоединения версии к сборке:

```
[assembly: AssemblyFileVersion ("1.2.3.4")]
```

С помощью префикса `field:` атрибут можно применять к поддерживающим полям автоматических свойств, что удобно в особых случаях, таких как применение (теперь устаревшего) атрибута `NonSerialized`:

```
[field:NonSerialized]  
public int MyProperty { get; set; }
```

Применение атрибутов к лямбда-выражениям

Начиная с версии C# 10, атрибуты можно применять к методу, параметрам и возвращаемому значению лямбда-выражения:

```
Action<int> a = [Description ("Method")]  
    [return: Description ("Return value")]  
    ([Description ("Parameter")])int x) => Console.WriteLine (x);
```



Прием полезен при работе с инфраструктурами вроде ASP.NET, которые полагаются на размещение атрибутов в написанных вами методах. Благодаря данному средству вы можете избежать необходимости создавать именованные методы для простых операций.

Эти атрибуты применяются к методу, созданному компилятором, на который указывает делегат. В главе 18 будет показано, как использовать рефлексию атрибутов в коде. На данный момент вот дополнительный код, который понадобится для разрешения такой косвенности:

```
var methodAtt = a.GetMethodInfo ().GetCustomAttributes ();  
var paramAtt = a.GetMethodInfo ().GetParameters () [0].GetCustomAttributes ();  
var returnAtt = a.GetMethodInfo ().ReturnParameter.GetCustomAttributes ();
```

Чтобы избежать синтаксической неоднозначности во время применения атрибутов к параметру лямбда-выражения, всегда требуются круглые скобки. В лямбда-выражениях дерева выражений атрибуты не допускаются.

Указание нескольких атрибутов

Для одного элемента кода разрешено указывать несколько атрибутов. Атрибуты могут быть заданы либо внутри единственной пары квадратных скобок (и разделяться запятыми), либо в отдельных парах квадратных скобок (или с помощью комбинации двух способов). Следующие три примера семантически идентичны:

```
[Serializable, Obsolete, CLSCompliant(false)]
public class Bar {...}

[Serializable] [Obsolete] [CLSCompliant(false)]
public class Bar {...}

[Serializable, Obsolete]
[CLSCompliant(false)]
public class Bar {...}
```

Атрибуты информации о вызывающем компоненте

Необязательные параметры можно помечать одним из трех *атрибутов информации о вызывающем компоненте*, которые сообщают компилятору о том, что в стандартное значение параметра необходимо поместить информацию, полученную из исходного кода вызывающего компонента:

- `[CallerMemberName]` применяет имя члена вызывающего компонента;
- `[CallerFilePath]` применяет путь к файлу исходного кода вызывающего компонента;
- `[CallerLineNumber]` применяет номер строки в файле исходного кода вызывающего компонента.

В показанном ниже методе `Foo` демонстрируется использование всех трех атрибутов:

```
using System;
using System.Runtime.CompilerServices;
class Program
{
    static void Main() => Foo();
    static void Foo (
        [CallerMemberName] string memberName = null,
        [CallerFilePath] string filePath = null,
        [CallerLineNumber] int lineNumber = 0)
    {
        Console.WriteLine (memberName);
        Console.WriteLine (filePath);
        Console.WriteLine (lineNumber);
    }
}
```

Предполагая, что код находится в файле c:\source\test\Program.cs, вывод будет таким:

```
Main  
c:\source\test\Program.cs  
6
```

Как и со стандартными необязательными параметрами, подстановка делается в месте вызова. Следовательно, наш метод Main является "синтаксическим сахаром" для следующего кода:

```
static void Main() => Foo ("Main", @"c:\source\test\Program.cs", 6);
```

Атрибуты информации о вызывающем компоненте полезны для регистрации в журнале, а также для реализации шаблонов, подобных инициированию одиночного события уведомления об изменении всякий раз, когда модифицируется любое свойство объекта. На самом деле для этого в пространстве имен System.ComponentModel предусмотрен стандартный интерфейс по имени INotifyPropertyChanged:

```
public interface INotifyPropertyChanged  
{  
    event PropertyChangedEventHandler PropertyChanged;  
}  
public delegate void PropertyChangedEventHandler  
    (object sender, PropertyChangedEventArgs e);  
public class PropertyChangedEventArgs : EventArgs  
{  
    public PropertyChangedEventArgs (string propertyName);  
    public virtual stringPropertyName { get; }  
}
```

Обратите внимание, что конструктор класса PropertyChangedEventArgs требует имени свойства, значение которого изменяется. Однако за счет применения атрибута [CallerMemberName] мы можем реализовать данный интерфейс и вызвать событие, даже не указывая имена свойств:

```
public class Foo : INotifyPropertyChanged  
{  
    public event PropertyChangedEventHandler PropertyChanged = delegate { };  
    void RaisePropertyChanged ([CallerMemberName] string propertyName = null)  
        => PropertyChanged (this, new PropertyChangedEventArgs (propertyName));  
    string customerName;  
    public string CustomerName  
    {  
        get => customerName;  
        set  
        {  
            if (value == customerName) return;  
            customerName = value;  
            RaisePropertyChanged ();  
            // Компилятор преобразует предыдущую строку кода в:  
            // RaisePropertyChanged ("CustomerName");  
        }  
    }  
}
```

Атрибут `CallerArgumentExpression`

Параметр метода, к которому применяется атрибут `[CallerArgumentExpression]` (C# 10), захватывает выражение аргумента из вызывающего компонента:

```
Print (Math.PI * 2);
void Print (double number,
    [CallerArgumentExpression("number")] string expr = null)
    => Console.WriteLine (expr);
// Вывод: Math.PI * 2
```

Компилятор буквальным образом передает исходный код вызывающего выражения, включая комментарии:

```
Print (Math.PI /*(п)*/ * 2);
// Вывод: Math.PI /*(п)*/ * 2
```

Данное средство в основном используется при написании библиотек проверки достоверности и утверждений. В следующем примере выдается исключение, сообщение которого включает текст `2 + 2 == 5`, что помогает в отладке:

```
Assert (2 + 2 == 5);
void Assert (bool condition,
    [CallerArgumentExpression ("condition")] string message = null)
{
    if (!condition) throw new Exception ("Assertion failed: " + message);
}
```

Еще один пример — статический метод `ThrowIfNull` класса `ArgumentNullException`, который появился в .NET 6 и определяется следующим образом:

```
public static void ThrowIfNull (object argument,
    [CallerArgumentExpression("argument")] string paramName = null)
{
    if (argument == null)
        throw new ArgumentNullException (paramName);
}
```

Вот как он применяется:

```
void Print (string message)
{
    ArgumentNullException.ThrowIfNull (message);
    ...
}
```

Атрибут `CallerArgumentExpression` можно использовать несколько раз, чтобы захватить множество выражений аргументов.

Динамическое связывание

Динамическое связывание откладывает *связывание* — процесс распознавания типов, членов и операций — с этапа компиляции до времени выполнения. Динамическое связывание удобно, когда на этапе компиляции вы знаете, что определенная функция, член или операция существует, но компилятору об этом

ничего не известно. Обычно подобное происходит при взаимодействии с динамическими языками (такими как IronPython) и COM, а также в сценариях, в которых иначе использовалась бы рефлексия.

Динамический тип объявляется с помощью контекстного ключевого слова `dynamic`:

```
dynamic d = GetSomeObject();
d.Quack();
```

Динамический тип предлагает компилятору смягчить требования. Мы ожидаем, что тип времени выполнения `d` должен иметь метод `Quack`. Мы просто не можем доказать это статически. Поскольку `d` относится к динамическому типу, компилятор откладывает связывание `Quack` с `d` до времени выполнения. Понимание смысла такого действия требует уяснения различий между *статическим связыванием* и *динамическим связыванием*.

Сравнение статического и динамического связывания

Каноническим примером связывания является отображение имени на специфическую функцию при компиляции выражения. Чтобы скомпилировать следующее выражение, компилятор должен найти реализацию метода по имени `Quack`:

```
d.Quack();
```

Давайте предположим, что статическим типом `d` является `Duck`:

```
Duck d = ...
d.Quack();
```

В простейшем случае компилятор осуществляет связывание за счет поиска в типе `Duck` метода без параметров по имени `Quack`. Если найти такой метод не удалось, тогда компилятор распространяет поиск на методы, принимающие необязательные параметры, методы базовых классов `Duck` и расширяющие методы, которые принимают тип `Duck` в своем первом параметре. Если совпадений не обнаружено, тогда возникает ошибка компиляции. Независимо от того, к какому методу произведено связывание, суть в том, что связывание делается компилятором, и оно полностью зависит от статических сведений о типах операндов (в данном случае `d`). Именно потому описанный процесс называется *статическим связыванием*.

А теперь изменим статический тип `d` на `object`:

```
object d = ...
d.Quack();
```

Вызов `Quack` приводит к ошибке на этапе компиляции, т.к. несмотря на то, что хранящееся в `d` значение способно содержать метод по имени `Quack`, компилятор не может об этом знать, поскольку единственной доступной ему информацией является тип переменной — `object` в рассматриваемом случае. Но давайте изменим статический тип `d` на `dynamic`:

```
dynamic d = ...
d.Quack();
```

Тип `dynamic` похож на `object`, т.к. он в равной степени не описывает тип. Отличие заключается в том, что тип `dynamic` допускает применение способами, которые на этапе компиляции не известны. Динамический объект связывается во время выполнения на основе своего типа времени выполнения, а не типа при компиляции. Когда компилятор встречает динамически связываемое выражение (которое в общем случае представляет собой выражение, содержащее любое значение типа `dynamic`), он просто упаковывает его так, чтобы связывание могло быть произведено позже во время выполнения.

Если динамический объект реализует `IDynamicMetaObjectProvider`, то данный интерфейс используется для связывания во время выполнения. Если же нет, то связывание проходит в основном так же, как в ситуации, когда компилятору известен тип динамического объекта времени выполнения. Две упомянутые альтернативы называются *специальным связыванием* и *языковым связыванием*.

Специальное связывание

Специальное связывание происходит, когда динамический объект реализует интерфейс `IDynamicMetaObjectProvider` (`IDMOP`). Хотя интерфейс `IDMOP` можно реализовать в типах, которые вы пишете на языке C#, и поступать так удобно, более распространенный случай предусматривает запрос объекта, реализующего `IDMOP`, из динамического языка, который внедряется в .NET через исполняющую среду динамического языка (*Dynamic Language Runtime — DLR*), скажем, *IronPython* или *IronRuby*. Объекты из таких языков неявно реализуют интерфейс `IDMOP` в качестве способа для прямого управления смыслом выполняемых над ними операций.

Мы детально обсудим специальные средства привязки в главе 19, но в целях демонстрации сейчас рассмотрим простое средство привязки:

```
using System;
using System.Dynamic;

dynamic d = new Duck();
d.Quack();      // Вызван метод Quack
d.Waddle();    // Вызван метод Waddle

public class Duck : DynamicObject
{
    public override bool TryInvokeMember (
        InvokeMemberBinder binder, object[] args, out object result)
    {
        Console.WriteLine (binder.Name + " method was called");
        result = null;
        return true;
    }
}
```

Класс `Duck` на самом деле не имеет метода `Quack`. Взамен он применяет специальное связывание для перехвата и интерпретации всех обращений к методам.

Языковое связывание

Языковое связывание происходит, если динамический объект не реализует интерфейс `IDMOP`. Языковое связывание удобно, когда приходится иметь дело с неудачно спроектированными типами или врожденными ограничениями системы типов `.NET` (в главе 19 будут представлены и другие сценарии). Обычной проблемой, возникающей во время работы с числовыми типами, является отсутствие общего интерфейса. Ранее было показано, что методы могут быть привязаны динамически; то же самое справедливо и для операций:

```
int x = 3, y = 4;  
Console.WriteLine (Mean (x, y));  
  
dynamic Mean (dynamic x, dynamic y) => (x + y) / 2;
```

Преимущество очевидно — не приходится дублировать код для каждого числового типа. Тем не менее, утрачивается безопасность типов, повышая риск генерации исключений во время выполнения вместо получения ошибок на этапе компиляции.



Динамическое связывание обходит статическую безопасность типов, но не динамическую безопасность типов. В отличие от рефлексии (см. главу 18) динамическое связывание не позволяет обойти правила доступности членов.

Согласно проектному решению языковое связывание во время выполнения ведет себя максимально похоже на статическое связывание, как будто типы времени выполнения динамических объектов были известны еще на этапе компиляции. Если в предыдущем примере жестко закодировать метод `Mean` для работы с типом `int`, то поведение программы останется идентичным. Наиболее заметным исключением при проведении аналогии между статическим и динамическим связыванием являются расширяющие методы, которые мы рассмотрим в разделе “Невызываемые функции” далее в главе.



Динамическое связывание также приводит к снижению производительности. Однако из-за механизмов кеширования среды `DLR` повторяющиеся обращения к одному и тому же динамическому выражению оптимизируются, позволяя эффективно работать с динамическими выражениями в цикле. Такая оптимизация доводит типичные накладные расходы в плане времени при выполнении простого динамического выражения на современном оборудовании до менее 100 нс.

Иключение `RuntimeBinderException`

Если привязка к члену не удается, тогда генерируется исключение `RuntimeBinderException`. Его можно считать ошибкой этапа компиляции, перенесенной на время выполнения:

```
dynamic d = 5;
d.Hello(); // Генерируется исключение RuntimeBinderException
```

Исключение генерируется из-за того, что тип `int` не имеет метода `Hello`.

Представление типа `dynamic` во время выполнения

Между типами `dynamic` и `object` имеется глубокая эквивалентность. Исполняющая среда трактует следующее выражение как `true`:

```
typeof (dynamic) == typeof (object)
```

Данный принцип распространяется на составные типы и массивы:

```
typeof (List<dynamic>) == typeof (List<object>)
typeof (dynamic[]) == typeof (object[])
```

Подобно объектной ссылке динамическая ссылка может указывать на объект любого типа (кроме типов указателей):

```
dynamic x = "hello";
Console.WriteLine (x.GetType().Name); // String
x = 123; // Ошибки нет (несмотря на то, что переменная та же самая)
Console.WriteLine (x.GetType().Name); // Int32
```

Структурно какие-либо отличия между объектной ссылкой и динамической ссылкой отсутствуют. Динамическая ссылка просто разрешает выполнение динамических операций над объектом, на который она указывает. Чтобы выполнить любую динамическую операцию над `object`, тип `object` можно преобразовать в `dynamic`:

```
object o = new System.Text.StringBuilder();
dynamic d = o;
d.Append ("hello");
Console.WriteLine (o); // hello
```



При выполнении рефлексии для типа, предлагающего (открытые) члены `dynamic`, обнаруживается, что такие члены представлены как аннотированные члены типа `object`. Например, следующее определение:

```
public class Test
{
    public dynamic Foo;
}
```

эквивалентно такому:

```
public class Test
{
    [System.Runtime.CompilerServices.DynamicAttribute]
    public object Foo;
}
```

Это позволяет потребителям типа знать, что член `Foo` должен трактоваться как `dynamic`, а другим языкам, не поддерживающим динамическое связывание, необходимо работать с ним как с `object`.

Динамические преобразования

Тип `dynamic` поддерживает неявные преобразования во все остальные типы и из них:

```
int i = 7;
dynamic d = i;
long j = d; // Приведение не требуется (неявное преобразование)
```

Чтобы преобразование прошло успешно, тип времени выполнения динамического объекта должен быть неявно преобразуемым в целевой статический тип. Предшествующий пример работает по той причине, что тип `int` неявно преобразуем в `long`.

В следующем примере генерируется исключение `RuntimeBinderException`, т.к. тип `int` не может быть неявно преобразован в `short`:

```
int i = 7;
dynamic d = i;
short j = d; // Генерируется исключение RuntimeBinderException
```

Сравнение `var` и `dynamic`

Несмотря на внешнее сходство типов `var` и `dynamic`, разница между ними существенна:

- `var` сообщает: позволить компилятору выяснить тип;
- `dynamic` сообщает: позволить исполняющей среде выяснить тип.

Ниже показана иллюстрация:

```
dynamic x = "hello";      // Статическим типом является dynamic,
                          // а типом времени выполнения - string
var y = "hello";          // Статическим типом является string,
                          // а типом времени выполнения - string
int i = x;                // Ошибка во время выполнения
                          // (невозможно преобразовать string в int)
int j = y;                // Ошибка на этапе компиляции
                          // (невозможно преобразовать string в int)
```

Статическим типом переменной, объявленной с ключевым словом `var`, может быть `dynamic`:

```
dynamic x = "hello";
var y = x; // Статическим типом у является dynamic
int z = y; //Ошибка во время выполнения (невозможно преобразовать string в int)
```

Динамические выражения

Поля, свойства, методы, события, конструкторы, индексаторы, операции и преобразования можно вызывать динамически. Попытка потребления результата динамического выражения с возвращаемым типом `void` пресекается — точно как в случае статически типизированного выражения. Отличие связано с тем, что ошибка возникает во время выполнения:

```
dynamic list = new List<int>();
var result = list.Add (5); //Генерируется исключение RuntimeBinderException
```

Выражения, содержащие динамические операнды, обычно сами являются динамическими, т.к. эффект отсутствия информации о типе имеет каскадный характер:

```
dynamic x = 2;
var y = x * 3;           // Статическим типом у является dynamic
```

Из такого правила существует пара очевидных исключений. Во-первых, приведение динамического выражения к статическому типу дает статическое выражение:

```
dynamic x = 2;
var y = (int)x;          // Статическим типом у является int
```

Во-вторых, вызовы конструкторов всегда дают статические выражения — даже если они производятся с динамическими аргументами. В следующем примере переменная x статически типизирована как `StringBuilder`:

```
dynamic capacity = 10;
var x = new System.Text.StringBuilder(capacity);
```

Кроме того, существует несколько краевых случаев, когда выражение, содержащее динамический аргумент, является статическим, включая передачу индекса массиву и выражения для создания делегатов.

Динамические вызовы без динамических получателей

В каноническом сценарии использования `dynamic` участвует динамический получатель. Это значит, что получателем динамического вызова функции будет динамический объект:

```
dynamic x = ...;
x.Foo();                // x является получателем
```

Тем не менее, статически известные функции можно вызывать также и с динамическими аргументами. Такие вызовы распознаются динамической перегрузкой и могут включать:

- статические методы;
- конструкторы экземпляра;
- методы экземпляра на получателях со статически известным типом.

В приведенном ниже примере конкретный метод `Foo`, который привязывается динамически, зависит от типа времени выполнения динамического аргумента:

```
class Program
{
    static void Foo (int x)    => Console.WriteLine ("int");
    static void Foo (string x) => Console.WriteLine ("string");
    static void Main()
    {
        dynamic x = 5;
        dynamic y = "watermelon";
        Foo (x); // int
        Foo (y); // string
    }
}
```

Поскольку динамический получатель не задействован, компилятор может статически выполнить базовую проверку успешности динамического вызова. Он проверяет, существует ли функция с корректным именем и количеством параметров. Если кандидаты не найдены, тогда возникает ошибка на этапе компиляции:

```
class Program
{
    static void Foo (int x)    => Console.WriteLine ("int");
    static void Foo (string x) => Console.WriteLine ("string");
    static void Main()
    {
        dynamic x = 5;
        Foo (x, x); //Ошибка на этапе компиляции - неправильное количество параметров
        Fook (x); //Ошибка на этапе компиляции - метод с таким именем отсутствует
    }
}
```

Статические типы в динамических выражениях

Тот факт, что динамические типы применяются в динамическом связывании, вполне очевиден. Однако участие в динамическом связывании также и статических типов не настолько очевидно. Взгляните на следующий код:

```
class Program
{
    static void Foo (object x, object y) { Console.WriteLine ("oo"); }
    static void Foo (object x, string y) { Console.WriteLine ("os"); }
    static void Foo (string x, object y) { Console.WriteLine ("so"); }
    static void Foo (string x, string y) { Console.WriteLine ("ss"); }
    static void Main()
    {
        object o = "hello";
        dynamic d = "goodbye";
        Foo (o, d); // os
    }
}
```

Вызов `Foo (o, d)` привязывается динамически, т.к. один из его аргументов, `d`, определен как `dynamic`. Но поскольку переменная `o` статически известна, связывание — хотя оно происходит динамически — будет использовать именно ее. В данном случае механизм распознавания перегруженных версий выберет вторую реализацию `Foo` из-за статического типа `o` и типа времени выполнения `d`. Другими словами, компилятор является “настолько статическим, насколько это возможно”.

Невызываемые функции

Некоторые функции не могут быть вызваны динамически. Нельзя вызывать динамически:

- расширяющие методы (через синтаксис расширяющих методов);
- члены интерфейса, если для этого необходимо выполнить приведение к данному интерфейсу;
- члены базового класса, которые скрыты подклассом.

Понимание причин важно для понимания динамического связывания в целом.

Динамическое связывание требует двух порций информации: имени вызываемой функции и объекта, на котором должна вызываться функция. Тем не менее, в каждом из трех невызываемых сценариев задействован дополнительный *тип*, который известен только на этапе компиляции. На момент написания главы не было способа динамического указания таких дополнительных типов.

При вызове расширяющих методов этот дополнительный тип является неявным. Он представляет собой статический класс, в котором определен расширяющий метод. Компилятор ищет его с учетом директив `using`, присутствующих в исходном коде. В результате расширяющие методы превращаются в концепции, существующие только на этапе компиляции, т.к. после компиляции директивы `using` исчезают (после завершения своей работы в рамках процесса связывания, которая заключается в отображении простых имен на имена, уточненные пространствами имен).

При вызове членов через интерфейс дополнительный тип указывается посредством неявного или явного приведения. Существуют два сценария, когда подобное может пригодиться: при вызове явно реализованных членов интерфейса и при вызове членов интерфейса, реализованных в типе, который является внутренним в другой сборке. Второй сценарий можно проиллюстрировать с помощью следующих двух типов:

```
interface IFoo { void Test(); }
class Foo : IFoo { void IFoo.Test() {} }
```

Для вызова метода `Test` потребуется выполнить приведение к интерфейсу `IFoo`. При статической типизации это делается легко:

```
IFoo f = new Foo(); // Неявное приведение к типу интерфейса
f.Test();
```

А теперь рассмотрим ситуацию с динамической типизацией:

```
IFoo f = new Foo();
dynamic d = f;
d.Test(); // Генерируется исключение
```

Выделенное полужирным неявное приведение сообщает компилятору о необходимости привязки последующих вызовов членов `f` к `IFoo`, а не к `Foo` — другими словами, для просмотра данного объекта сквозь призму интерфейса `IFoo`. Однако во время выполнения такая призма теряется, а потому среда DLR не может завершить связывание. Упомянутая потеря продемонстрирована ниже:

```
Console.WriteLine (f.GetType().Name); // Foo
```

Похожая ситуация возникает при вызове скрытого члена базового класса: дополнительный тип должен указываться либо через приведение, либо с помощью ключевого слова `base` — и этот дополнительный тип утрачивается во время выполнения.



Если нужно динамически обращаться к членам интерфейса, то обходной путь предусматривает использование библиотеки с открытым кодом `Uncapsulator`, которая доступна на NuGet и GitHub. Библиотека `Uncapsulator` была написана автором книги для решения этой проблемы и применяет *специальное связывание* для обеспечения лучшей динамичности, чем `dynamic`:

```
IFoo f = new Foo();  
dynamic uf = f.Uncapsulate();  
uf.Test();
```

Библиотека `Uncapsulator` также позволяет выполнять приведение к базовым типам и интерфейсам по имени, динамически вызывать статические члены и получать доступ к закрытым членам типа.

Перегрузка операций

Операции могут быть перегружены для предоставления специальным типам более естественного синтаксиса. Перегрузку операций целесообразнее всего применять при реализации специальных структур, которые представляют относительно примитивные типы данных. Например, хорошим кандидатом на перегрузку операций может служить специальный числовой тип.

Разрешено перегружать следующие символические операции:

+ (унарная)	- (унарная)	!	~	++
--	+	-	*	/
%	&		^	<<
>>	==	!=	>	<
>=	<=			

Перечисленные ниже операции также могут быть перегружены:

- явные и неявные преобразования (с использованием ключевых слов `explicit` и `implicit`);
- операции (не литералы) `true` и `false`.

Следующие операции перегружаются косвенно:

- составные операции присваивания (например, `+=`, `/=`) неявно перегружаются при перегрузке обычных операций (т.е. `+`, `/`);
- условные операции `&&` и `||` неявно перегружаются при перегрузке побитовых операций `&` и `|`.

Функции операций

Операция перегружается за счет объявления функции *операции*. Функция операции подчиняется перечисленным далее правилам.

- Имя функции указывается с помощью ключевого слова `operator`, за которым следует символ операции.
- Функция операции должна быть помечена как `static` и `public`.
- Параметры функции операции представляют операнды.
- Возвращаемый тип функции операции представляет результат выражения.
- По меньшей мере, один из operandов должен иметь тип, для которого объявляется функция операции.

В следующем примере мы определяем структуру по имени `Note`, представляющую музыкальную ноту, и затем перегружаем операцию `+`:

```
public struct Note
{
    int value;
    public Note (int semitonesFromA) { value = semitonesFromA; }
    public static Note operator + (Note x, int semitones)
    {
        return new Note (x.value + semitones);
    }
}
```

Перегруженная версия операции `+` позволяет добавлять значение `int` к `Note`:

```
Note B = new Note (2);
Note CSharp = B + 2;
```

Перегрузка операции приводит к автоматической перегрузке соответствующей составной операции присваивания. Поскольку в примере перегружена операция `+`, можно также применять операцию `+=`:

```
CSharp += 2;
```

Подобно методам и свойствам функции операций, состоящие из одиночного выражения, в C# разрешено записывать более компактно с помощью синтаксиса функций, сжатых до выражений:

```
public static Note operator + (Note x, int semitones)
    => new Note (x.value + semitones);
```

Проверяемые операции

Начиная с версии C# 11, при объявлении функции операции также можно объявить версию `checked`:

```
public static Note operator + (Note x, int semitones)
    => new Note (x.value + semitones);

public static Note operator checked + (Note x, int semitones)
    => checked (new Note (x.value + semitones));
```

Проверяемая версия будет вызываться внутри проверяемых выражений или блоков:

```
Note B = new Note (2);
Note other = checked (B + int.MaxValue); // генерирует исключение OverflowException
```

Перегрузка операций эквивалентности и сравнения

Операции эквивалентности и сравнения часто перегружаются при написании структур и в редких случаях — при написании классов. При перегрузке операций эквивалентности и сравнения должны соблюдаться специальные правила и обязательства, которые будут подробно рассматриваться в главе 6. Ниже приведен краткий обзор этих правил.

- **Парность.** Компилятор C# требует, чтобы операции, которые представляют собой логические пары, были определены обе. Такими операциями являются `(== !=)`, `(< >)` и `(<= >=)`.
- **Equals и GetHashCode.** В большинстве случаев при перегрузке операций `==` и `!=` потребуется переопределять методы `Equals` и `GetHashCode` класса `object`, чтобы обеспечить осмысленное поведение. Компилятор C# выдаст предупреждение, если это не сделано. (Дополнительные сведения предоставлялись в разделе “Сравнение эквивалентности” ранее в главе.)
- **IComparable и IComparable<T>.** Если перегружаются операции `(< >)` и `(<= >=)`, тогда обязательно должны быть реализованы интерфейсы `IComparable` и `IComparable<T>`.

Специальные неявные и явные преобразования

Неявные и явные преобразования являются перегружаемыми операциями. Как правило, они перегружаются для того, чтобы сделать преобразования между тесно связанными типами (такими как числовые типы) лаконичными и естественными.

Для преобразования между слабо связанными типами больше подходят следующие стратегии:

- написать конструктор, принимающий параметр типа, из которого выполняется преобразование;
- написать методы `ToXXX` и (статические) методы `FromXXX`, предназначенные для преобразования между типами.

Как объяснялось при обсуждении типов, логическое обоснование неявных преобразований заключается в том, что они гарантированно выполняются успешно и не приводят к потере информации. И наоборот, явное преобразование должно быть обязательным либо когда успешность преобразования определяется обстоятельствами во время выполнения, либо если в результате преобразования может быть потеряна информация.

В показанном далее примере мы определяем преобразования между типом `Note` и типом `double` (с использованием которого представлена частота в герцах данной ноты):

```
...
// Преобразование в герцы
public static implicit operator double (Note x)
=> 440 * Math.Pow (2, (double) x.value / 12 );
```

```
// Преобразование из герц (с точностью до ближайшего полутона)
public static explicit operator Note (double x)
    => new Note ((int) (0.5 + 12 * (Math.Log (x/440) / Math.Log(2) ) ));
...
Note n = (Note) 554.37;                                // явное преобразование
double x = n;                                         // неявное преобразование
```



Следуя нашим собственным принципам, этот пример можно реализовать более эффективно с помощью метода `ToFrequency` (и статического метода `FromFrequency`) вместо неявной и явной операции.



Операции `as` и `is` игнорируют специальные преобразования:

```
Console.WriteLine (554.37 is Note); // False
Note n = 554.37 as Note;          // Ошибка
```

Перегрузка операций `true` и `false`

Операции `true` и `false` перегружаются в крайне редких случаях для типов, которые являются булевскими “по духу”, но не имеют преобразования в `bool`. Примером может служить тип, реализующий логику с тремя состояниями: за счет перегрузки `true` и `false` этот тип может гладко работать с условными операторами и операциями, а именно — `if`, `do`, `while`, `for`, `&&`, `||` и `??:`. Такую функциональность предоставляет структура `System.Data.SqlTypes.SqlBoolean`:

```
SqlBoolean a = SqlBoolean.Null;
if (a)
    Console.WriteLine ("True");
else if (!a)
    Console.WriteLine ("False");
else
    Console.WriteLine ("Null");
```

Вот вывод:

Null

Приведенный ниже код представляет собой повторную реализацию частей структуры `SqlBoolean`, необходимой для демонстрации работы с операциями `true` и `false`:

```
public struct SqlBoolean
{
    public static bool operator true (SqlBoolean x)
        => x.m_value == True.m_value;
    public static bool operator false (SqlBoolean x)
        => x.m_value == False.m_value;
    public static SqlBoolean operator ! (SqlBoolean x)
    {
        if (x.m_value == Null.m_value) return Null;
        if (x.m_value == False.m_value) return True;
        return False;
    }
}
```

```
public static readonly SqlBoolean Null = new SqlBoolean(0);
public static readonly SqlBoolean False = new SqlBoolean(1);
public static readonly SqlBoolean True = new SqlBoolean(2);

private SqlBoolean (byte value) { m_value = value; }
private byte m_value;
}
```

Статический полиморфизм

В разделе “Вызов статических виртуальных/абстрактных членов интерфейсов” главы 18 будет представлено расширенное средство, благодаря которому интерфейс может определять члены `static virtual` или `static abstract`, впоследствии реализуемые как статические члены классами и структурами. В разделе “Ограничения обобщений” главы 3 было показано, что применение ограничения интерфейса к параметру типа обеспечивает методу доступ к членам данного интерфейса. Здесь мы объясним, как это обеспечивает *статический полиморфизм*, делая возможными такие средства, как обобщенная математика.

В целях иллюстрации рассмотрим следующий интерфейс, в котором определен статический метод для создания случайного экземпляра типа `T`:

```
interface ICreateRandom<T>
{
    static abstract T CreateRandom(); // Создает случайный экземпляр типа T
}
```

Предположим, что этот интерфейс необходимо реализовать в следующей записи:

```
record Point (int X, int Y);
```

Вот как можно реализовать статический метод `CreateRandom` с помощью класса `System.Random` (метод `Next` которого генерирует случайное целое число):

```
record Point (int X, int Y) : ICreateRandom<Point>
{
    static Random rnd = new();
    public static Point CreateRandom() => new Point (rnd.Next(), rnd.Next());
}
```

Для вызова этого метода через интерфейс используется *ограниченный параметр типа*. В приведенном ниже методе создается массив тестовых данных с применением такого приема:

```
T[] CreateTestData<T> (int count) where T : ICreateRandom<T>
{
    T[] result = new T[count];
    for (int i = 0; i < count; i++)
        result [i] = T.CreateRandom();
    return result;
}
```

Следующая строка кода демонстрирует его использование:

```
Point[] testData = CreateTestData<Point>(50); // Создать 50 случайных
// экземпляров Point.
```

Вызов статического метода `CreateRandom` в `CreateTestData` является полиморфным, т.к. он работает не только с `Point`, но и с любым типом, реализующим `ICreateRandom<T>`. Подход отличается от полиморфизма *экземпляров*, потому что для вызова `CreateRandom` экземпляр реализации `ICreateRandom<T>` не нужен; метод `CreateRandom` вызывается на самом типе.

Полиморфные операции

Поскольку операции по существу являются статическими функциями (см. раздел “Перегрузка операций” ранее в главе), они также могут быть объявлены в виде статических виртуальных членов интерфейса:

```
interface IAddable<T> where T : IAddable<T>
{
    abstract static T operator + (T left, T right);
}
```



Самоссылающееся ограничение типа в приведенном определении интерфейса необходимо для удовлетворения правил компилятора по перегрузке операций. Вспомните, что при определении функции операции хотя бы один из операндов должен быть того типа, с которым объявлена сама функция операции. В рассмотренном примере операнды имеют тип `T`, тогда как содержащих их типом является `IAddable<T>`, поэтому требуется самоссылающееся ограничение типа, чтобы `T` можно было трактовать как `IAddable<T>`.

Вот как можно реализовать данный интерфейс:

```
record Point (int X, int Y) : IAddable<Point>
{
    public static Point operator + (Point left, Point right) =>
        new Point (left.X + right.X, left.Y + right.Y);
}
```

Затем с использованием ограниченного параметра типа можно написать метод, который полиморфно вызывает операцию сложения (для краткости обработка краевых случаев опущена):

```
T Sum<T> (params T[] values) where T : IAddable<T>
{
    T total = values[0];
    for (int i = 1; i < values.Length; i++)
        total += values[i]; return total;
}
```

Обращение к операции `+` (через операцию `+=`) является полиморфным, т.к. оно привязывается к `IAddable<T>`, а не к `Point`. Следовательно, метод `Sum` работает со всеми типами, реализующими `IAddable<T>`.

Конечно, интерфейс вроде `IAddable<T>` был бы гораздо полезнее, если бы он определялся в исполняющей среде .NET и его реализовывали все числовые типы .NET. К счастью, это действительно так в .NET 7: пространство имен `System.Numerics` содержит (более сложную версию) интерфейса `IAddable`, а также множество других арифметических интерфейсов, большинство из которых включено в `INumber<TSelf>`.

Обобщенная математика

До выхода .NET 7 код, выполняющий арифметические операции, должен был быть жестко запрограммирован для определенного числового типа:

```
int Sum (params int[] numbers)      // Работает только с int.  
{                                         // Нельзя использовать с double, decimal и т.д.  
    int total = 0;  
    foreach (int n in numbers)  
        total += n;  
    return total;  
}
```

В версии .NET 7 появился интерфейс `INumber<TSelf>`, предназначенный для унификации арифметических операций среди числовых типов. Это означает, что теперь можно написать обобщенную версию предыдущего метода:

```
T Sum<T> (params T[] numbers) where T : INumber<T>  
{  
    T total = T.Zero;  
    foreach (T n in numbers)  
        total += n; // Вызывает операцию сложения для любого числового типа  
    return total;  
}  
int intSum = Sum (3, 5, 7);  
double doubleSum = Sum (3.2, 5.3, 7.1);  
decimal decimalSum = Sum (3.2m, 5.3m, 7.1m);
```

Интерфейс `INumber<TSelf>` реализуется всеми вещественными и целочисленными числовыми типами в .NET (а также типом `char`) и может рассматриваться как обобщающий интерфейс, который включает в себя другие, более детальные интерфейсы для каждого вида арифметических операций (сложение, вычитание, умножение, деление, деление по модулю, сравнение и т.д.), а также интерфейсы для разбора и форматирования. Вот один из таких интерфейсов:

```
public interface IAdditionOperators<TSelf, TOther, TResult>  
    where TSelf : IAdditionOperators<TSelf, TOther, TResult>?  
{  
    static abstract TResult operator + (TSelf left, TOther right);  
    public static virtual TResult operator checked +  
        (TSelf left, TOther right) => left + right; //Вызывает предыдущую операцию  
}
```

Операция `+`, объявленная как `static abstract`, позволяет операции `+=` работать внутри нашего метода `Sum`. Также обратите внимание на использование `static virtual` в проверяемой операции: это обеспечивает стандартное резервное поведение для разработчиков, которые не предоставляют проверяемую версию операции сложения.

Пространство имен `System.Numerics` содержит также интерфейсы, не являющиеся частью `INumber`, которые поддерживают операции, специфичные для определенных типов чисел (например, с плавающей точкой). Скажем, чтобы вычислить среднеквадратичное значение, можно добавить интерфейс `IRootFunctions<T>` в список ограничений, чтобы предоставить его статический метод `RootN` для `T`:

```

T RMS<T> (params T[] values) where T : INumber<T>, IRootFunctions<T>
{
    T total = T.Zero;
    for (int i = 0; i < values.Length; i++)
        total += values [i] * values [i];
    // Use T.CreateChecked to convert values.Length (type int) to T
    T count = T.CreateChecked (values.Length);
    return T.RootN (total / count, 2); // Вычислить квадратный корень
}

```

Небезопасный код и указатели

Язык C# поддерживает прямые манипуляции с памятью через указатели внутри блоков кода, которые помечены как `unsafe` (небезопасные). Типы указателей полезны при взаимодействии с собственными API-интерфейсами, для доступа в память за пределами управляемой кучи и для реализации микрооптимизаций в “горячих” точках, критичных к производительности.

Основы указателей

Для каждого типа значения или ссылочного типа V имеется соответствующий тип указателя `V*`. Экземпляр указателя хранит адрес переменной. Тип указателя может быть (небезопасно) приведен к любому другому типу указателя. Ниже описаны основные операции над указателями.

Операция	Описание
<code>&</code>	Операция взятия адреса возвращает указатель на адрес переменной
<code>*</code>	Операция разыменования возвращает значение переменной, которая находится по адресу, заданному указателем
<code>-></code>	Операция указателя на член является синтаксическим сокращением, т.е. <code>x->y</code> эквивалентно <code>(*x).y</code>

В файлах проектов, содержащих небезопасный код, должен быть указан элемент `<AllowUnsafeBlocks>true</AllowUnsafeBlocks>`.

Небезопасный код

Помечая тип, член типа или блок операторов ключевым словом `unsafe`, вы разрешаете внутри этой области видимости использовать типы указателей и выполнять операции над указателями в стиле С. Ниже показан пример применения указателей для быстрой обработки битовой карты:

```

unsafe void BlueFilter (int[,] bitmap)
{
    int length = bitmap.Length;
    fixed (int* b = bitmap)
    {
        int* p = b;
        for (int i = 0; i < length; i++)
            *p++ &= 0xFF;
    }
}

```

Небезопасный код может выполняться быстрее, чем соответствующая ему безопасная реализация. В последнем случае код потребует вложенного цикла с индексацией в массиве и проверкой границ. Небезопасный метод C# может также оказаться быстрее, чем вызов внешней функции С, поскольку не будет никаких накладных расходов, связанных с покиданием управляемой среды выполнения.

Оператор `fixed`

Оператор `fixed` необходим для закрепления управляемого объекта, такого как битовая карта в предыдущем примере. Во время выполнения программы многие объекты распределяются в куче и впоследствии освобождаются. Во избежание нежелательных затрат или фрагментации памяти сборщик мусора перемещает объекты внутри кучи. Указатель на объект бесполезен, если адрес объекта может измениться, пока на него производится ссылка, и потому оператор `fixed` сообщает сборщику мусора о необходимости “закрепления” объекта, чтобы он никуда не перемещался. Это может оказать влияние на эффективность программы во время выполнения, так что блоки `fixed` должны использоваться только кратковременно, а внутри блока `fixed` следует избегать распределения памяти в куче.

В рамках оператора `fixed` можно получать указатель на любой тип значения, массив типов значений или строку. В случае массивов и строк указатель будет в действительности указывать на первый элемент, который относится к типу значения. Типы значений, объявленные внутри ссылочных типов, требуют закрепления ссылочных типов, как показано ниже:

```
Test test = new Test();
unsafe
{
    fixed (int* p = &test.X) // Закрепляет test
    {
        *p = 9;
    }
    Console.WriteLine (test.X);
}
class Test { public int X; }
```

Более подробно оператор `fixed` рассматривается в разделе “Отображение структуры на неуправляемую память” главы 24.

Операция указателя на член

В дополнение к операциям `&` и `*` язык C# также предлагает операцию `->` в стиле C++, которая может применяться при работе со структурами:

```
Test test = new Test();
unsafe
{
    Test* p = &test;
    p->X = 9;
    System.Console.WriteLine (test.X);
}
struct Test { public int X; }
```

Ключевое слово `stackalloc`

Память может быть выделена в блоке внутри стека явно с использованием ключевого слова `stackalloc`. Из-за распределения в стеке время жизни блока памяти ограничивается выполнением метода, в точности как для любой другой локальной переменной. К данному блоку можно применять операцию `[]` для проведения индексации в рамках памяти:

```
int* a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
    Console.WriteLine (a[i]);
```

В главе 23 будет описано, как можно использовать `Span<T>` для управления памятью, выделенной в стеке, без применения ключевого слова `unsafe`:

```
Span<int> a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
    Console.WriteLine (a[i]);
```

Буферы фиксированных размеров

С ключевым словом `fixed` связан еще один сценарий использования — создание буферов фиксированных размеров внутри структур (что может быть полезно при вызове неуправляемой функции; см. главу 24):

```
new UnsafeClass ("Christian Troy");
unsafe struct UnsafeUnicodeString
{
    public short Length;
    public fixed byte Buffer[30]; // Выделить блок из 30 байтов
}
unsafe class UnsafeClass
{
    UnsafeUnicodeString uus;
    public UnsafeClass (string s)
    {
        uus.Length = (short)s.Length;
        fixed (byte* p = uus.Buffer)
            for (int i = 0; i < s.Length; i++)
                p[i] = (byte) s[i];
    }
}
```

Буферы фиксированных размеров не являются массивами: если бы `Buffer` был массивом, то он состоял бы из ссылки на объект, хранящийся в (управляемой) куче, а не из 30 байтов внутри самой структуры.

В приведенном выше примере ключевое слово `fixed` применяется еще и для закрепления в куче объекта, содержащего буфер (который будет экземпляром `UnsafeClass`). Таким образом, ключевое слово `fixed` имеет два разных смысла: фиксация размера и фиксация места. Оба случая использования часто встречаются вместе, поскольку для работы с буфером фиксированного размера он должен быть зафиксирован на месте.

void*

Указатель `void (void*)` не делает никаких предположений о типе лежащих в основе данных и удобен для функций, которые имеют дело с низкоуровневой памятью. Существует явное преобразование из любого типа указателя в `void*`. Указатель `void*` не допускает разыменования и выполнения над ним арифметических операций. Вот пример:

```
short[] a = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
unsafe
{
    fixed (short* p = a)
    {
        // Операция sizeof возвращает размер типа значения в байтах
        Zap (p, a.Length * sizeof (short));
    }
}
foreach (short x in a)
    Console.WriteLine (x); // Выводит все нули
unsafe void Zap (void* memory, int byteCount)
{
    byte* b = (byte*)memory;
    for (int i = 0; i < byteCount; i++)
        *b++ = 0;
}
```

Целочисленные типы с собственным размером

Целочисленные типы с собственным размером `nint` и `nuint` (появившиеся в версии C# 9) имеют размер, соответствующий адресному пространству процесса во время выполнения (на практике 32 или 64 бита). Целые числа с собственным размером ведут себя подобно стандартным целым числам, полностью поддерживая арифметические операции и проверку на предмет переполнения:

```
nint x = 123, y = 234;
checked
{
    nint sum = x + y, product = x * y;
    Console.WriteLine (product);
}
```

Целые числа с собственным размером могут быть 32-битными целочисленными константами (но не 64-битными целочисленными константами, потому что во время выполнения может возникнуть переполнение). Для преобразования целочисленных типов с собственным размером в другие целочисленные типы либо наоборот можно применять явное приведение.

Целые числа с собственным размером можно использовать для представления адресов или смещений в памяти, не прибегая к помощи указателей. Тип `nuint` также является естественным типом для представления длины блока памяти.

При работе с указателями целые числа с собственным размером могут повысить эффективность, поскольку результатом вычитания двух указателей в C# всегда является 64-битное целое число (типа long), что незэффективно на 32-разрядных платформах. Если сначала привести указатели к nint, тогда результатом вычитания тоже окажется nint (который будет занимать 32 бита на 32-разрядной платформе):

```
unsafe nint AddressDif (char* x, char* y) => (nint)x - (nint)y;
```



Хорошим примером реального использования типов nint и nuint в сочетании с указателями является реализация Buffer.Memory.Copy. Ее можно увидеть в файле Buffer.cs исходного кода .NET на GitHub или декомпилировать метод в ILSPy. Упрощенная версия также включена в примеры LINQPad для этой книги.

Обработка исполняющей средой при нацеливании на .NET 7+

Для проектов, ориентированных на .NET 7 или более позднюю версию, типы nint и nuint действуют как синонимы базовых типов System.IntPtr и System.UIntPtr из .NET (точно так же, как тип int выступает синонимом System.Int32). Это работает, поскольку типы IntPtr и UIntPtr (существовавшие начиная с .NET Framework 1.0, но обладающие ограниченной функциональностью) в .NET 7 были расширены, чтобы обеспечить полные возможности в плане арифметических операций и проверку на предмет переполнения с помощью компилятора C#.



Добавление к IntPtr/UIntPtr возможности проверяемой арифметики формально является критическим изменением. Тем не менее, эффекты ограничены, поскольку работа унаследованного кода, полагающегося на то, что IntPtr не учитывает блоки checked, не нарушается при простом запуске под управлением .NET 7+; чтобы работа была нарушена, проект должен быть перекомпилирован с нацеливанием на .NET 7+. Таким образом, авторам библиотек не нужно беспокоиться о критических изменениях, пока они не выпустят новую версию, специально предназначенную для .NET 7 или более поздней версии.

Обработка исполняющей средой при нацеливании на .NET 6 или предыдущую версию

Для проектов, ориентированных на .NET 6 или более раннюю версию (либо .NET Standard), типы nint и nuint по-прежнему используют IntPtr и UIntPtr в качестве базовых типов исполняющей среды. Однако поскольку унаследованные типы IntPtr и UIntPtr не поддерживают большинство арифметических операций, компилятор заполняет бреши, заставляя типы nint/nuint вести себя так, как они ведут себя в .NET 7+ (включая разрешение операций checked). Можно считать, что переменная типа nint/nuint относится к типу IntPtr/UIntPtr со специальным верхним слоем. Компилятор воспринимает ее

как имеющую современный тип IntPtr/UIntPtr. Естественно, специальный верхний слой утрачивается, если позже привести переменную к типу IntPtr/ UIntPtr:

```
nint x = 123;
Console.WriteLine (x * x);      // Нормально: умножение поддерживается

IntPtr y = x;
Console.WriteLine (y * y);      // Ошибка на этапе компиляции: операция *
                                // не поддерживается
```

Указатели на функции

Указатель на функцию (появившийся в версии C# 9) похож на делегат, но без косвенного обращения к экземпляру делегата; взамен он указывает непосредственно на метод. Указатель на функцию может указывать только на статические методы, не обладает возможностью группового вызова и требует контекста unsafe (поскольку обходит систему безопасности типов во время выполнения). Основная его цель — упрощение и оптимизация взаимодействия с неуправляемыми API-интерфейсами (см. раздел “Обратные вызовы из неуправляемого кода” в главе 24).

Тип указателя на функцию объявляется следующим образом (последним задается возвращаемый тип):

```
delegate*<int, char, string, void> // (void - возвращаемый тип)
```

Такой указатель на функцию соответствует функции с сигнатурой вида:

```
void SomeFunction (int x, char y, string z)
```

Операция & создает указатель на функцию из группы методов. Вот полный пример:

```
unsafe
{
    delegate*<string, int> functionPointer = &GetLength;
    int length = functionPointer ("Hello, world");
    static int GetLength (string s) => s.Length;
}
```

В этом примере functionPointer не является *объектом*, на котором можно вызывать метод, такой как Invoke (или с помощью ссылки на объект Target). Напротив, functionPointer представляет собой переменную, которая указывает непосредственно на адрес целевого метода в памяти:

```
Console.WriteLine ((IntPtr)functionPointer);
```

Подобно любому другому указателю он не подвергается проверке типов во время выполнения. В показанном ниже коде возвращаемое значение нашей функции трактуется как значение decimal (которое длиннее значения int, поэтому в выводе будет присутствовать и случайное содержимое дополнительных ячеек памяти):

```
var pointer2 = (delegate*<string, decimal>) (IntPtr) functionPointer;
Console.WriteLine (pointer2 ("Hello, unsafe world"));
```

Атрибут `SkipLocalsInit`

Когда компилятор C# компилирует метод, он выпускает флаг, который инструктирует исполняющую среду о необходимости инициализации локальных переменных метода их стандартными значениями (путем обнуления памяти). Начиная с версии C# 9, компилятору можно сообщить о том, чтобы он не выпускал такой флаг, применяя к методу атрибут `[SkipLocalsInit]` (из пространства имен `System.Runtime.CompilerServices`):

```
[SkipLocalsInit]  
void Foo() ...
```

Данный атрибут можно также применять к типу, что эквивалентно его применению ко всем методам типа, или даже к целому модулю (контейнеру для сборки):

```
[module: System.Runtime.CompilerServices.SkipLocalsInit]
```

В обычных безопасных сценариях атрибут `[SkipLocalsInit]` мало влияет на функциональность или производительность, поскольку политика определенного присваивания C# требует явного присваивания значений локальным переменным, прежде чем их можно будет использовать. Это означает, что оптимизатор JIT, вероятно, будет выпускать одинаковый машинный код независимо от того, применялся атрибут `[SkipLocalsInit]` или нет.

Тем не менее, в небезопасном контексте использование `[SkipLocalsInit]` может успешно избавить среду CLR от накладных расходов по инициализации локальных переменных типа значения, обеспечивая небольшой выигрыш в производительности для методов, которые широко задействуют стек (посредством крупной операции `stackalloc`). В показанном ниже примере в случае применения атрибута `[SkipLocalsInit]` выводится содержимое неинициализированной памяти (вместо всех нулей):

```
[SkipLocalsInit]  
unsafe void Foo()  
{  
    int local;  
    int* ptr = &local;  
    Console.WriteLine (*ptr);  
    int* a = stackalloc int [100];  
    for (int i = 0; i < 100; ++i) Console.WriteLine (a [i]);  
}
```

Интересно отметить, что того же самого результата в “безопасном” контексте можно достичь за счет использования `Span<T>`:

```
[SkipLocalsInit]  
void Foo()  
{  
    Span<int> a = stackalloc int [100];  
    for (int i = 0; i < 100; ++i) Console.WriteLine (a [i]);  
}
```

Следовательно, применение атрибута `[SkipLocalsInit]` требует компиляции проекта с элементом `<AllowUnsafeBlocks>true</AllowUnsafeBlocks>` внутри файла проекта, даже если ни один из методов не помечен как `unsafe`.

Директивы препроцессора

Директивы препроцессора снабжают компилятор дополнительной информацией о разделах кода. Наиболее распространенными директивами препроцессора считаются директивы условной компиляции, которые предоставляют способ включения либо исключения разделов кода из процесса компиляции:

```
#define DEBUG
class MyClass
{
    int x;
    void Foo()
    {
        #if DEBUG
        Console.WriteLine ("Testing: x = {0}", x);
        #endif
    }
    ...
}
```

В классе `MyClass` оператор внутри метода `Foo` компилируется условным образом в зависимости от существования символа `DEBUG`. Если удалить определение символа `DEBUG`, тогда этот оператор в `Foo` компилироваться не будет. Символы препроцессора могут определяться внутри файла исходного кода (что и было сделано в примере) или на уровне проекта в файле `.csproj`:

```
<PropertyGroup>
    <DefineConstants>DEBUG;ANOTHERSYMBOL</DefineConstants>
</PropertyGroup>
```

В директивах `#if` и `#elif` можно применять операции `&&`, `||` и `!` для выполнения логических действий *И*, *ИЛИ* и *НЕ* над несколькими символами. Представленная ниже директива указывает компилятору на необходимость включения следующего за ней кода, если определен символ `TESTMODE` и не определен символ `DEBUG`:

```
#if TESTMODE && !DEBUG
    ...

```

Тем не менее, имейте в виду, что вы не строите обычное выражение C#, а символы, которыми вы оперируете, не имеют абсолютно никакого отношения к *переменным* — статическим или каким-то другим.

Директивы `#error` и `#warning` предотвращают случайное неправильное использование директив условной компиляции, заставляя компилятор генерировать предупреждение или сообщение об ошибке, которое вызвано неподходящим набором символов компиляции. Директивы препроцессора перечислены в табл. 4.1.

Таблица 4.1. Директивы препроцессора

Директива препроцессора	Действие
#define <i>символ</i>	Определяет символ
#undef <i>символ</i>	Отменяет определение символа
#if <i>символ</i> [<i>операция символ2</i>] ...	Проверяет, определен ли символ, и компилирует, если это так; операциями являются ==, !=, && и , за которыми следуют директивы #else, #elif и #endif
#else	Компилирует код до следующей директивы #endif
#elif <i>символ</i> [<i>операция символ2</i>]	Комбинирует ветвь #else и проверку #if
#endif	Заканчивает директивы условной компиляции
#warning <i>текст</i>	Заставляет компилятор вывести предупреждение с указанным текстом
#error <i>текст</i>	Заставляет компилятор вывести сообщение об ошибке с указанным текстом
#error <i>версия</i>	Заставляет компилятор вывести номер своей версии и закончить работу
#pragma warning [disable restore]	Отключает или восстанавливает выдачу компилятором предупреждения (предупреждений)
#line [<i>номер</i> [" <i>файл</i> "] hidden]	Номер задает строку в исходном коде (начиная с версии C# 10, можно также задавать колонку); в " <i>файл</i> " указывается имя файла для помещения в вывод компилятора; hidden инструктирует инструменты отладки о необходимости пропуска кода от этой точки до следующей директивы #line
#region <i>имя</i>	Обозначает начало раздела
#endregion	Обозначает конец раздела
#nullable <i>выбор</i>	См. раздел "Ссыльные типы, допускающие null" ранее в главе

Условные атрибуты

Атрибут, декорированный атрибутом Conditional, будет компилироваться, только если определен заданный символ препроцессора:

```
// file1.cs
#define DEBUG
using System;
using System.Diagnostics;
[Conditional("DEBUG")]
public class TestAttribute : Attribute {}

// file2.cs
#define DEBUG
[Test]
```

```
class Foo
{
    [Test]
    string s;
}
```

Компилятор включит атрибуты [Test], только если символ DEBUG определен в области видимости для файла file2.cs.

Директива #pragma warning

Компилятор генерирует предупреждение, когда обнаруживает в коде что-то, выглядящее непреднамеренным. В отличие от ошибок предупреждения обычно не препятствуют компиляции приложения.

Предупреждения компилятора могут быть исключительно полезными при выявлении ошибок. Тем не менее, их полезность снижается в случае выдачи ложных предупреждений. Чтобы заметить *настоящие* предупреждения в крупном приложении, важно поддерживать подходящее соотношение “сигнал-шум”.

С этой целью компилятор позволяет избирательно подавлять выдачу предупреждений посредством директивы #pragma warning. В следующем примере мы указываем компилятору, чтобы он не выдавал предупреждения о том, что поле Message не применяется:

```
public class Foo
{
    static void Main() { }
    #pragma warning disable 414
    static string Message = "Hello";
    #pragma warning restore 414
}
```

Отсутствие числа в директиве #pragma warning означает, что будет отключена или восстановлена выдача предупреждений со всеми кодами.

Если вас интересует всестороннее использование данной директивы, тогда можете скомпилировать код с переключателем /warnaserror, который сообщает компилятору о необходимости трактовать любые оставшиеся предупреждения как ошибки.

XML-документация

Документирующий комментарий — это порция встроенного XML-кода, которая документирует тип или член типа. Документирующий комментарий располагается непосредственно перед объявлением типа или члена и начинается с трех символов косой черты:

```
/// <summary>Прекращает выполняющийся запрос.</summary>
public void Cancel() { ... }
```

Многострочные комментарии записываются следующим образом:

```
/// <summary>
/// Прекращает выполняющийся запрос.
/// </summary>
public void Cancel() { ... }
```

или так (обратите внимание на дополнительный символ звездочки в начале):

```
/**  
 * <summary>Прекращает выполняющийся запрос.</summary>  
 */  
public void Cancel() { ... }
```

В случае добавления показанных ниже строк к файлу .csproj:

```
<PropertyGroup>  
    <DocumentationFile>SomeFile.xml</DocumentationFile>  
</PropertyGroup>
```

компилятор извлекает и накапливает документирующие комментарии в указанном XML-файле, с которым связаны два основных сценария использования.

Если он размещен в той же папке, что и скомпилированная сборка, то инструменты вроде Visual Studio и LINQPad автоматически читают такой XML-файл и применяют информацию из него для предоставления списка членов через средство IntelliSense потребителям сборки с таким же именем, как у XML-файла. Сторонние инструменты (такие как Sandcastle и NDoc) могут трансформировать XML-файл в справочный HTML-файл.

Стандартные XML-дескрипторы документации

Ниже перечислены стандартные XML-дескрипторы, которые распознаются Visual Studio и генераторами документации.

<summary>

```
<summary>...</summary>
```

Указывает всплывающую подсказку, которую средство IntelliSense должно отображать для типа или члена; обычно это одиночное выражение или предложение.

<remarks>

```
<remarks>...</remarks>
```

Дополнительный текст, который описывает тип или член. Генераторы документации объединяют его с полным описанием типа или члена.

<param>

```
<param name="имя">...</param>
```

Объясняет параметр метода.

<returns>

```
<returns>...</returns>
```

Объясняет возвращаемое значение метода.

<exception>

```
<exception [cref="тип"]>...</exception>
```

Указывает исключение, которое метод может генерировать (в cref задается тип исключения).

<permission>

```
<permission [cref="тип"]>...</permission>
```

Указывает тип IPermission, требуемый документируемым типом или членом.

<example>

```
<example>...</example>
```

Обозначает пример (используемый генераторами документации). Как правило, содержит описательный текст и исходный код (исходный код обычно заключен в дескриптор `<c>` или `<code>`).

<c>

```
<c>...</c>
```

Указывает внутристочный фрагмент кода. Этот дескриптор обычно применяется внутри блока `<example>`.

<code>

```
<code>...</code>
```

Указывает многострочный пример кода. Этот дескриптор обычно используется внутри блока `<example>`.

<see>

```
<see cref="член">...</see>
```

Вставляет внутристочную перекрестную ссылку на другой тип или член. Генераторы HTML-документации обычно преобразуют ее в гиперссылку. Компилятор выдает предупреждение, если указано недопустимое имя типа или члена.

<seealso>

```
<seealso cref="член">...</seealso>
```

Вставляет перекрестную ссылку на другой тип или член. Генераторы документации обычно помещают ее внутрь отдельного раздела “See Also” (“См. также”) в нижней части страницы.

<paramref>

```
<paramref name="имя"/>
```

Вставляет ссылку на параметр внутри дескриптора `<summary>` или `<remarks>`.

<list>

```
<list type=[ bullet | number | table ]>
  <listheader>
    <term>...</term>
    <description>...</description>
  </listheader>
  <item>
    <term>...</term>
    <description>...</description>
  </item>
</list>
```

Инструктирует генератор документации о необходимости выдачи маркированного (bullet), нумерованного (number) или табличного (table) списка.

<para>

```
<para>...</para>
```

Инструктирует генератор документации о необходимости форматирования содержимого в виде отдельного абзаца.

<include>

```
<include file='имя-файла' path='путь-к-дескриптору[@name="идентификатор"]'>
  ...
</include>
```

Выполняет объединение с внешним XML-файлом, содержащим документацию. В атрибуте path задается XPath-запрос к конкретному элементу из этого файла.

Дескрипторы, определяемые пользователем

С предопределенными XML-дескрипторами, распознаваемыми компилятором C#, не связано ничего особенного, и вы можете также определять собственные дескрипторы. Компилятор организует специальную обработку только для дескриптора `<param>` (роверяет имя параметра, а также выясняет, документированы ли все параметры метода) и атрибута `cref` (роверяет, что атрибут ссылается на реальный тип или член, и расширяет его в полностью заданный идентификатор типа или члена). Атрибут `cref` можно также применять в собственных дескрипторах; он проверяется и расширяется в точности как для предопределенных дескрипторов `<exception>`, `<permission>`, `<see>` и `<seealso>`.

Перекрестные ссылки на типы или члены

Имена типов и перекрестные ссылки на типы или члены транслируются в идентификаторы, уникальным образом определяющие тип или член. Такие имена образованы из префикса, который определяет, что конкретно представляет идентификатор, и сигнатуры типа или члена. Префиксы членов перечислены ниже.

XML-префикс типа	К какому идентификатору применяется
N	Пространство имен
T	Тип (класс, структура, перечисление, интерфейс, делегат)
F	Поле
P	Свойство (включая индексаторы)
M	Метод (включая специальные методы)
E	Событие
!	Ошибка

Правила генерации сигнатур хорошо документированы, хотя и довольно сложны.

Вот пример типа и сгенерированных идентификаторов:

```
// Пространства имен не имеют независимых сигнатур
namespace NS
{
    /// T:NS.MyClass
    class MyClass
    {
        /// F:NS.MyClass.aField
        string aField;

        /// P:NS.MyClass.aProperty
        short aProperty {get {...} set {...);}

        /// T:NS.MyClass.NestedType
        class NestedType {...};

        /// M:NS.MyClass.X()
        void X() {...}

        /// M:NS.MyClass.Y(System.Int32,System.Double@,System.Decimal@)
        void Y(int p1, ref double p2, out decimal p3) {...}

        /// M:NS.MyClass.Z(System.Char[],System.Single[0:,0:])
        void Z(char[] p1, float[,] p2) {...}

        /// M:NS.MyClass.op_Addition(NS.MyClass,NS.MyClass)
        public static MyClass operator+(MyClass c1, MyClass c2) {...}

        /// M:NS.MyClass.op_Implicit(NS.MyClass)`System.Int32
        public static implicit operator int(MyClass c) {...}

        /// M:NS.MyClass.#ctor
        MyClass() {...}

        /// M:NS.MyClass.Finalize
        ~MyClass() {...}

        /// M:NS.MyClass.#cctor
        static MyClass() {...}
    }
}
```



5

Обзор .NET

Почти все возможности исполняющей среды .NET 8 доступны через обширное множество управляемых типов. Типы организованы в иерархические пространства имен и упакованы в набор сборок.

Некоторые типы .NET используются напрямую CLR и являются критически важными для среды управляемого размещения. Такие типы находятся в сборке по имени `System.Private.CoreLib.dll` (`mscorlib.dll` в .NET Framework) и включают встроенные типы C#, а также базовые классы коллекций, типы для обработки потоков данных, сериализации, рефлексии, многопоточности и собственной возможности взаимодействия.

Уровнем выше находятся дополнительные типы, которые расширяют функциональность уровня CLR, предоставляя такие средства, как XML, JSON, взаимодействие с сетью и LINQ. Они образуют библиотеку базовых классов (Base Class Library — BCL). Выше находятся *прикладные слои*, которые предоставляют API-интерфейсы для разработки определенных видов приложений наподобие веб-приложения или обогащенного клиентского приложения.

Вот что предлагается в настоящей главе:

- краткий обзор библиотеки BCL (которая будет более подробно рассматриваться в оставшихся главах книги);
- высокоуровневый обзор прикладных слоев.

Библиотеки базовых классов в .NET 7 и .NET 8 включают множество новых функциональных средств и улучшений производительности.

- Формат архивов Tar, популярный в системах Unix, теперь поддерживается через типы в новом пространстве имен `System.Formats.Tar` (см. раздел “Работа с файлами Tar” в главе 15). Класс `ZipFile` тоже был усовершенствован, чтобы позволить архивировать папки с файлами непосредственно в поток либо из него.
- Класс `Stream` теперь предоставляет методы `ReadExactly` и `ReadAtLeast` для упрощения чтения из потоков (см. раздел “Чтение и запись” в главе 15).
- Появилась поддержка работы с разрешениями файлов Unix (см. раздел “Безопасность файлов в Unix” в главе 15).
- Расширена поддержка `Span<T>` и `ReadOnlySpan<T>`. В частности, числовые и другие простые типы теперь поддерживают форматирование и разбор UTF-8 непосредственно в `Span<byte>` через новые интерфейсы `IUtf8SpanFormattable` и `IUtf8SpanParseable<TSelf>`, а класс `MemoryExtensions` содержит дополнительные расширяющие методы, которые помогают искать значения внутри промежутков (см. раздел “Поиск в промежутках” в главе 23).
- Класс `Random` теперь содержит метод `GetItems` для выбора случайных элементов из коллекции и метод `Shuffle` для случайного перемешивания элементов (см. раздел “Класс Random” в главе 6).
- Типы даты и времени .NET теперь предоставляют свойства `Microsecond` и `Nanosecond`.
- Класс `JsonNode` имеет ряд новых методов, включая `GetValueKind`, `DeepEquals`, `DeepClone` и `ReplaceWith` (см. раздел “Класс JsonNode” в главе 11).
- Появились два новых типа коллекций, доступных только для чтения: `FrozenDictionary<K,V>` и `FrozenSet<T>`. Они похожи на существующие типы `ImmutableDictionary<K,V>` и `ImmutableHashSet<T>`, но оптимизированы исключительно для чтения, без методов неразрушающего изменения (см. раздел “Замороженные коллекции” в главе 7).
- Класс `RegEx` теперь поддерживает флаг `RegexOptions.NonBacktracking`, который помогает избежать атак типа отказа в обслуживании с использованием выражений, предоставленных пользователем (см. раздел “Перечисление флагов `RegexOptions`” в главе 25). Кроме того, механизм регулярных выражений стал работать быстрее.
- Теперь доступны типы для хеширования SHA-3 при условии его поддержки операционной системой (см. раздел “Алгоритмы хеширования в .NET” в главе 20).

Механизм сериализации JSON тоже был усовершенствован: в нем появились новые функциональные средства и повышена его производительность.

Целевые платформы и TFM

Элемент `<TargetFramework>` в файле проекта определяет, для какой исполняющей среды создается проект (его целевая платформа или целевая исполняющая среда) и обозначается с помощью моникера целевой платформы (Target Framework Moniker — TFM). Допустимыми значениями являются `net8.0`, `net7.0`, `net6.0`, `net5.0` (для версий .NET 8, 7, 6 и 5), `netcoreapp3.1` (для .NET Core 3.1), `net48` (для .NET Framework 4.8) и `netstandard2.0` (стандарт, который будет обсуждаться в следующем разделе). Например, вот как нацелиться на .NET 8:

```
<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
<PropertyGroup>
```

Можно нацелииться на несколько исполняющих сред, указав взамен `<TargetFramework>` элемент `<TargetFrameworks>`. Моникеры TFM отделяются друг от друга точкой с запятой:

```
<TargetFrameworks>net8.0;net48</TargetFrameworks>
```

При нацеливании на множество исполняющих сред компилятор создает для каждой цели отдельную выходную сборку.

Целевая платформа кодируется в выходной сборке с помощью атрибута `TargetFramework`. Сборка может выполняться под управлением более новой (но не более старой) исполняющей среды, нежели указанная для нее цель.

.NET Standard

Изобилие публичных библиотек, которые доступны через NuGet, не было бы настолько ценным, если бы они поддерживали только .NET 8. При написании библиотеки вы часто хотите поддерживать различные платформы и версии исполняющей среды. Чтобы достичь такой цели, не создавая отдельные сборки для каждой исполняющей среды (при множественном нацеливании), вы должны ориентироваться на наименьший общий знаменатель. Это относительно легко при желании поддерживать только непосредственных предшественников .NET 8: скажем, если проект нацелен на .NET 6 (`net6.0`), то ваша библиотека будет работать под управлением .NET 6, .NET 7 и .NET 8.

Ситуация становится более запутанной, когда желательно поддерживать также и .NET Framework (или унаследованные исполняющие среды вроде Xamarin). Дело в том, что каждая такая исполняющая среда имеет CLR и BCL с перекрывающимися функциональными средствами — ни одна исполняющая среда не является чистым подмножеством остальных.

.NET Standard решает проблему путем определения искусственных подмножеств, которые обеспечивают работу под управлением целого набора исполняющих сред. Нацеливаясь на .NET Standard, вы можете легко создавать библиотеки с широким охватом.



.NET Standard — не исполняющая среда, а просто спецификация, описывающая минимальный базовый уровень функциональности (типы и члены), который гарантирует совместимость с определенным набором исполняющих сред. Концепция похожа на интерфейсы C#: стандарт .NET Standard подобен интерфейсу, который конкретные типы (исполняющие среды) могут реализовывать.

.NET Standard 2.0

Наиболее полезной версией является .NET Standard 2.0. Библиотека, которая ориентирована на .NET Standard 2.0, а не на специфическую исполняющую среду, будет работать без каких-либо изменений под управлением современных версий .NET (.NET 8/7/6/5 вплоть до .NET Core 2) и .NET Framework (4.6.1+). Она также будет поддерживать унаследованные платформы UWP (начиная с версии 10.0.16299+) и Mono 5.4+ (CLR/BCL, используемые более старыми версиями Xamarin).

Для нацеливания на .NET Standard 2.0 добавьте в свой файл .csproj следующие строки:

```
<PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
<PropertyGroup>
```

Большинство описанных в книге API-интерфейсов поддерживаются стандартом .NET Standard 2.0 (а большинство остальных доступны в виде пакетов NuGet).

Другие стандарты .NET Standard

.NET Standard 2.1 является надмножеством стандарта .NET Standard 2.0, которое поддерживает (только) следующие платформы:

- .NET Core 3+
- Mono 6.4+

Стандарт .NET Standard 2.1 не поддерживается ни одной из версий .NET Framework, что делает его менее полезным, чем .NET Standard 2.0.

Существуют также более старые стандарты .NET Standard, такие как 1.1, 1.2, 1.3 и 1.6, совместимость которых распространяется на устаревшие среды выполнения, подобные .NET Core 1.0 или .NET Framework 4.5. В стандартах 1.x отсутствуют тысячи API-интерфейсов, которые имеются в версии 2.0 (включая большую часть того, что описано в настоящей книге), так что стандарты 1.x фактически прекратили свое существование.

Совместимость .NET Framework и .NET 8

Инфраструктура .NET Framework существует настолько долго, что нередко можно встретить библиотеки, которые доступны только для .NET Framework (без каких-либо эквивалентов для .NET Standard, .NET Core или .NET 8). Для

смягчения последствий в такой ситуации проектам .NET 5+ и .NET Core разрешено ссылаться на сборки .NET Framework при соблюдении следующих условий.

- При обращении в сборке .NET Framework к API-интерфейсу, который не поддерживается в .NET Core, должно генерироваться исключение.
- Распознавание нетривиальных зависимостей может оказаться безуспешным (и часто таковым оказывается).

Скорее всего, на практике это будет работать в простых случаях, таких как сборка, которая представляет собой оболочку для неуправляемой DLL-библиотеки.

Ссылочные сборки

Когда ваш проект нацелен на .NET Standard, он неявно ссылается на сборку по имени `netstandard.dll`, которая содержит все разрешенные типы и члены для выбранной версии .NET Standard. Она называется *ссылочной сборкой*, потому что существует только в интересах компилятора и не содержит скомпилированный код. Во время выполнения “настоящие” сборки идентифицируются через атрибуты перенаправления сборок (выбор сборок зависит от того, под управлением какой исполняющей среды и платформы в итоге будет производиться запуск).

Интересно отметить, что похожие вещи происходят при нацеливании на .NET 8. Ваш проект неявно ссылается на набор ссылочных сборок, типы которых отражают содержимое сборок времени выполнения для выбранной версии .NET. Это помогает управлять версиями и межплатформенной совместимостью и также позволяет нацеливаться на версию .NET, которая отличается от версии, установленной на вашей машине.

Версии исполняющих сред и языка C#

По умолчанию используемую версию языка C# определяет целевая исполняющая среда проекта:

Целевая исполняющая среда	Версия C#
.NET 8	C# 12
.NET 7	C# 11
.NET 6	C# 10
.NET 5	C# 9
.NET Core 3.x и 2.x	C# 8
.NET Framework	C# 7.3
.NET Standard 2.0	C# 7.3

Причина в том, что более поздние версии C# включают функциональные средства, которые основаны на типах, представленных в более поздних исполняющих средах.

Переопределить версию языка можно с помощью элемента `<LangVersion>` в файле проекта. Использование более старой исполняющей среды (скажем, .NET Framework) с более поздней версией языка (например, C# 12) означает, что языковые средства, основанные на новых типах .NET, работать не будут (хотя в некоторых случаях вы можете определить такие типы самостоятельно либо импортировать их из пакета NuGet).

Среда CLR и библиотека BCL

Системные типы

Наиболее фундаментальные типы находятся непосредственно в пространстве имен `System`. В их состав входят встроенные типы C#, базовый класс `Exception`, базовые классы `Enum`, `Array` и `Delegate`, а также типы `Nullable`, `Type`, `DateTime`, `TimeSpan` и `Guid`. Кроме того, пространство имен `System` включает типы для выполнения математических функций (`Math`), генерации случайных чисел (`Random`) и преобразования между различными типами (`Convert` и `BitConverter`).

Фундаментальные типы описаны в главе 6 вместе с интерфейсами, которые определяют стандартные протоколы, используемые повсеместно в .NET для решения таких задач, как форматирование (`IFormattable`) и сравнение порядка (`IComparable`).

В пространстве имен `System` также определен интерфейс `IDisposable` и класс `GC` для взаимодействия со сборщиком мусора; мы рассмотрим их в главе 12.

Обработка текста

Пространство имен `System.Text` содержит класс `StringBuilder` (редактируемый или изменяемый родственник `string`) и типы для работы с кодировками текста, такими как UTF-8 (`Encoding` и его подтипы). Мы раскроем их в главе 6.

Пространство имен `System.Text.RegularExpressions` содержит типы, которые выполняют расширенные операции поиска и замены на основе образца; такие типы будут описаны в главе 25.

Коллекции

В .NET предлагаются разнообразные классы для управления коллекциями элементов. Они включают структуры, основанные на списках и словарях, и работают в сочетании с набором стандартных интерфейсов, которые унифицируют их общие характеристики. Все типы коллекций определены в следующих пространствах имен, описанных в главе 7:

```
System.Collections           // Необобщенные коллекции
System.Collections.Generic // Обобщенные коллекции
System.Collections.Frozen  // Высокопроизводительные коллекции,
                           // допускающие только чтение
System.Collections.Immutable // Универсальные коллекции,
                           // допускающие только чтение
System.Collections.Specialized // Строго типизированные коллекции
System.Collections.ObjectModel // Базовые типы для создания
                               // собственных коллекций
System.Collections.Concurrent // Коллекции, безопасные в отношении
                               // потоков (глава 22)
```

Запросы

Язык LINQ позволяет выполнять безопасные в отношении типов запросы к локальным и удаленным коллекциям (например, к таблицам SQL Server) и описан в главах 8, 9 и 10. Крупное преимущество языка LINQ заключается в том, что он предоставляет согласованный API-интерфейс запросов для разнообразных предметных областей. Основные типы находятся в перечисленных ниже пространствах имен:

```
System.Linq                  // LINQ to Objects и PLINQ
System.Linq.Expressions       // Для ручного построения выражений
System.Xml.Linq              // LINQ to XML
```

XML и JSON

XML и JSON поддерживаются в .NET повсеместно. Внимание в главе 10 сосредоточено целиком на LINQ to XML — легковесной модели документных объектов (Document Object Model — DOM) для XML, которую можно конструировать и опрашивать с помощью LINQ. В главе 11 описаны высокопроизводительные низкоуровневые классы для чтения/записи разметки XML, схем и таблиц стилей XML, а также типы для работы с данными JSON:

```
System.Xml          // XmlReader, XmlWriter и старая модель W3C DOM
System.Xml.Linq     // DOM-модель LINQ to XML
System.Xml.Schema   // Поддержка для XSD
System.Xml.Serialization // Декларативная сериализация XML для типов .NET
System.Xml.XPath    // Язык запросов XPath
System.Xml.Xsl      // Поддержка таблиц стилей
System.Text.Json    // Средство чтения/записи JSON и DOM
System.Text.Json.Nodes // API-интерфейс JsonNode (DOM)
```

Сериализатор JSON описан в дополнительных материалах, доступных для загрузки на веб-сайте издательства.

Диагностика

В главе 13 мы рассмотрим регистрацию в журнале и утверждения, а также покажем, как взаимодействовать с другими процессами, выполнять запись в журнал событий Windows и проводить мониторинг производительности.

Соответствующие типы определены в пространстве имен System.Diagnostics и его подпространствах.

Параллелизм и асинхронность

Многим современным приложениям в каждый момент времени приходится иметь дело с несколькими действиями. Начиная с версии C# 5.0, решение стало проще за счет асинхронных функций и таких высокоуровневых конструкций, как задачи и комбинаторы задач. Все это подробно объясняется в главе 14, которая начинается с рассмотрения основ многопоточности. Типы для работы с потоками и асинхронными операциями находятся в пространствах имен System.Threading и System.Threading.Tasks.

Потоки данных и ввод-вывод

В .NET предоставляется потоковая модель для низкоуровневого ввода-вывода. Потоки данных обычно применяются для чтения и записи напрямую в файлы и сетевые подключения и могут соединяться в цепочки либо помещаться внутрь декорированных потоков с целью добавления функциональности сжатия или шифрования. В главе 15 описана потоковая архитектура, а также специфическая поддержка для работы с файлами и каталогами, сжатием, изолированным хранилищем, каналами и файлами, отображенными в память. Тип Stream и типы ввода-вывода определены в пространстве имен System.IO и его подпространствах.

Работа с сетями

С помощью типов из пространства имен System.Net можно напрямую работать с большинством стандартных сетевых протоколов вроде HTTP, TCP/IP и SMTP. В главе 16 будет показано, как взаимодействовать с применением каждого из упомянутых протоколов, начиная с простых задач вроде загрузки веб-страницы и заканчивая применением TCP/IP для извлечения сообщений электронной почты POP3. Ниже перечислены пространства имен, которые будут рассмотрены:

```
System.Net
System.Net.Http      // HttpClient
System.Net.Mail      // Для отправки электронной почты через SMTP
System.Net.Sockets   // TCP, UDP и IP
```

Сборки, рефлексия и атрибуты

Сборки, в которые компилируются программы на C#, состоят из исполняемых инструкций (представленных на языке IL) и метаданных, которые описывают типы, члены и атрибуты программы. С помощью рефлексии можно просматривать метаданные во время выполнения и предпринимать действия вроде динамического вызова методов. Посредством пространства имен Reflection.Emit можно конструировать новый код на лету.

В главе 17 мы опишем строение сборок и объясним, как их динамически загружать и изолировать. В главе 18 мы раскроем рефлексию и атрибуты — покажем, как инспектировать метаданные, динамически вызывать функции, записывать специальные атрибуты, выпускать новые типы и производить разбор низкоуровневого кода IL. Типы для применения рефлексии и работы со сборками находятся в следующих пространствах имен:

```
System  
System.Reflection  
System.Reflection.Emit
```

Динамическое программирование

В главе 19 мы рассмотрим несколько паттернов для динамического программирования и работы со средой DLR. Мы покажем, как реализовать паттерн “Посетитель” (Visitor), создавать специальные динамические объекты и взаимодействовать с IronPython. Типы, предназначенные для динамического программирования, находятся в пространстве имен System.Dynamic.

Криптография

.NET обеспечивает всестороннюю поддержку для популярных протоколов хеширования и шифрования. В главе 20 мы раскроем хеширование, симметричное шифрование и шифрование с открытым ключом, а также API-интерфейс Windows Data Protection. Типы для этого определены в следующих пространствах имен:

```
System.Security  
System.Security.Cryptography
```

Расширенная многопоточность

Асинхронные функции в C# значительно облегчают параллельное программирование, поскольку снижают потребность во взаимодействии с низкоуровневыми технологиями. Тем не менее, все еще возникают ситуации, когда нужны сигнальные конструкции, локальное хранилище потока, блокировки чтения/записи и т.д. Данные вопросы подробно обсуждаются в главе 21. Типы, связанные с многопоточностью, находятся в пространстве имен System.Threading.

Параллельное программирование

В главе 22 мы рассмотрим библиотеки и типы для работы с многоядерными процессорами, включая API-интерфейсы для реализации параллелизма задач, императивного параллелизма данных и функционального параллелизма (PLINQ).

Span<T> и Memory<T>

Для содействия микрооптимизации в “горячих” точках в плане производительности среда CLR предлагает несколько типов, которые помогают програм-

мировать так, чтобы снизить нагрузку на диспетчер памяти. Двумя ключевыми типами подобного рода являются `Span<T>` и `Memory<T>`, которые будут описаны в главе 23.

Возможность взаимодействия с собственным кодом и COM

Вы можете взаимодействовать с собственным кодом и с кодом COM. Возможность взаимодействия с собственным кодом позволяет вызывать функции из неуправляемых DLL-библиотек, регистрировать обратные вызовы, отображать структуры данных и работать с собственными типами данных. Возможность взаимодействия с COM позволяет обращаться к типам COM (на машинах Windows) и открывать для COM доступ к типам .NET. Типы, поддерживающие такую функциональность, определены в пространстве имен `System.Runtime.InteropServices` и рассматриваются в главе 24.

Регулярные выражения

В главе 25 мы покажем, как можно использовать регулярные выражения для сопоставления с символьными шаблонами в строках.

Сериализация

.NET предлагает несколько систем для сохранения и восстановления объектов в двоичном или текстовом представлении. Такие системы могут применяться для передачи данных, а также для сохранения и восстановления объектов из файлов. В дополнительных материалах, доступных для загрузки на веб-сайте издательства, раскрываются все четыре механизма сериализации: двоичный сериализатор, (обновленный) сериализатор JSON, сериализатор XML и сериализатор на основе контрактов данных.

Компилятор Roslyn

Сам компилятор C# написан на языке C# — проект называется Roslyn, а библиотеки доступны в виде пакетов NuGet. С помощью этих библиотек вы можете эксплуатировать функциональность компилятора многими способами помимо компиляции исходного кода в сборку, скажем, писать инструменты для анализа и рефакторинга кода. Компилятор Roslyn рассматривается в дополнительных материалах, доступных для загрузки на веб-сайте издательства.

Прикладные слои

Приложения, основанные на пользовательском интерфейсе, можно разделить на две категории: *тонкий клиент*, равнозначный веб-сайту, и *обогащенный клиент*, представляющий собой программу, которую конечный пользователь должен загрузить и установить на компьютере или на мобильном устройстве.

Для разработки приложений тонких клиентов на языке C# предусмотрена инфраструктура ASP.NET Core, которая запускается в среде Windows, Linux и

macOS. Кроме того, инфраструктура ASP.NET Core позволяет реализовывать API-интерфейсы для веб-сети.

Для разработки приложений обогащенных клиентов на выбор доступно несколько API-интерфейсов:

- слой Windows Desktop, который включает популярные API-интерфейсы WPF и Windows Forms и функционирует на настольных компьютерах с Windows 7/8/10/11;
- WinUI 3 (Windows App SDK) — преемник UWP, который работает (только) на настольных компьютерах с Windows 10+;
- UWP позволяет писать приложения для Магазина Windows, которые функционируют на настольных компьютерах с Windows 10+ и таких устройствах, как Xbox или HoloLens;
- MAUI (ранее Xamarin) работает на мобильных устройствах iOS и Android. MAUI также позволяет создавать межплатформенные настольные приложения, предназначенные для выполнения в средах macOS (через Catalyst) и Windows (через Windows App SDK).

Вдобавок существуют сторонние межплатформенные библиотеки для построения пользовательского интерфейса, подобные Avalonia. В отличие от MAUI библиотека Avalonia также поддерживает Linux и не использует слой косвенного управления Catalyst/WinUI для платформ настольных систем, что упрощает разработку и отладку.

ASP.NET Core

ASP.NET Core — это легковесный модульный преемник ASP.NET, который подходит для создания веб-сайтов, API-интерфейсов на основе REST и микрослужб. Кроме того, ASP.NET Core может работать в сочетании с двумя популярными фреймворками для одностраничных приложений: React и Angular.

ASP.NET поддерживает популярный паттерн *MVC* (Model-View-Controller — модель-представление-контроллер), а также более новую технологию под названием *Blazor*, где клиентский код реализован на C#, а не JavaScript.

ASP.NET Core функционирует под управлением Windows, Linux и macOS и может самостоятельно размещаться в специальном процессе. В отличие от своего предшественника из .NET Framework (ASP.NET) архитектура ASP.NET Core не зависит от *System.Web* и от исторического багажа веб-форм.

Как и любая архитектура тонкого клиента, ASP.NET Core обладает следующими общими преимуществами по сравнению с обогащенными клиентами:

- отсутствует развертывание на клиентской стороне;
- клиент может запускаться на любой платформе, которая поддерживает веб-браузер;
- легко развертывать обновления.

Windows Desktop

Прикладной слой Windows Desktop предлагает на выбор два API-интерфейса для реализации пользовательских интерфейсов в приложениях обогащенных клиентов: WPF и Windows Forms. Оба API-интерфейса работают в среде Windows Desktop/Server 7–11.

WPF

Инфраструктура WPF появилась в 2006 году и с тех пор неоднократно расширялась. В отличие от своей предшественницы, Windows Forms, она явно визуализирует элементы управления с использованием DirectX, обеспечивая перечисленные ниже преимущества.

- Инфраструктура WPF поддерживает развитую графику, включая произвольные трансформации, трехмерную визуализацию, мультимедиа-возможности и подлинную прозрачность. Оформление поддерживается через стили и шаблоны.
- Основная единица измерения не базируется на пикселях, поэтому приложения корректно отображаются при любой настройке DPI (Dots Per Inch — точек на дюйм).
- Она располагает обширной и гибкой поддержкой динамической компоновки, означающей возможность локализации приложения без опасности того, что элементы будут перекрывать друг друга.
- Применение DirectX делает визуализацию быстрой и способной извлекать преимущества от аппаратного ускорения графики.
- Она предлагает надежную привязку к данным.
- Пользовательские интерфейсы могут быть описаны декларативно в XAML-файлах, которые допускают сопровождение независимо от файлов отдельного кода, что помогает разнести внешний вид и функциональность.

Инфраструктура WPF требует некоторого времени на изучение из-за своего размера и сложности. Типы, предназначенные для написания WPF-приложений, находятся в пространстве имен `System.Windows` и во всех его подпространствах за исключением `System.Windows.Forms`.

Windows Forms

Windows Forms — это API-интерфейс обогащенного клиента, который поставлялся с первой версией .NET Framework в 2000 году. По сравнению с WPF она является относительно простой технологией, которая предлагает большинство возможностей, необходимых во время разработки типового Windows-приложения. Она также играла важную роль в сопровождении унаследованных приложений. Тем не менее, в сравнении с WPF инфраструктура Windows Forms обладает рядом недостатков, большинство из которых объясняется тем, что она является оболочкой для GDI+ и библиотеки элементов управления Win32.

- Хотя Windows Forms предоставляет механизмы для осведомленности о настройках DPI, все же слишком легко получать приложения, которые не-

корректно отображаются на клиентах с настройками DPI, отличающимися от таких настроек у разработчика.

- Для рисования нестандартных элементов управления используется API-интерфейс GDI+, который вопреки достаточно высокой гибкости медленно визуализирует крупные области (и без двойной буферизации может вызывать мерцание).
- Элементы управления лишены подлинной прозрачности.
- Большинство элементов управления не поддерживают компоновку. Например, поместить элемент управления изображением внутрь заголовка элемента управления вкладкой не удастся. Настройка списковых представлений, полей с раскрывающимися списками и элементов управления с вкладками, которая была бы тривиальной в WPF, требует много времени и сил.
- Трудно корректно и надежно реализовать динамическую компоновку.

Последний пункт является веской причиной отдавать предпочтение WPF перед Windows Forms, даже если разрабатывается бизнес-приложение, которому необходим только пользовательский интерфейс, а не учет “поведенческих особенностей пользователей”. Элементы компоновки в WPF, подобные Grid, упрощают организацию меток и текстовых полей таким образом, что они будут всегда выровненными — даже при смене языка локализации — без запутанной логики и какого-либо мерцания. Кроме того, не придется приводить все к наименьшему общему знаменателю в смысле экранного разрешения — элементы компоновки WPF изначально проектировались с поддержкой изменения размеров.

В качестве положительного момента следует отметить, что инфраструктура Windows Forms относительно проста в изучении и все еще поддерживается в немалом количестве сторонних элементов управления.

Типы Windows Forms находятся в пространствах имен System.Windows.Forms (сборка System.Windows.Forms.dll) и System.Drawing (сборка System.Drawing.dll). Последнее пространство имен также содержит типы GDI+ для рисования специальных элементов управления.

UWP и WinUI 3

Универсальная платформа Windows (Universal Windows Platform — UWP) представляет собой API-интерфейс обогащенного клиента, который предназначен для разработки сенсорных приложений, ориентированных на настольные компьютеры и устройства Windows 10+. Слово “универсальная” относится к способности UWP функционировать на различных устройствах Windows 10, в том числе Xbox, Surface Hub, HoloLens и (на то время) Windows Phone.

API-интерфейс UWP использует XAML и кое в чем похож на WPF. Ниже перечислены его ключевые отличия.

- Приложения UWP поставляются главным образом через Магазин Microsoft.
- Приложения UWP работают в песочнице, чтобы уменьшить угрозу со стороны вредоносного программного обеспечения, а это означает, что они не могут выполнять такие задачи, как чтение или запись произвольных файлов, и их нельзя запускать с повышенными административными правами.

- Платформа UWP опирается на типы WinRT, которые являются частью операционной системы (Windows), а не управляемой исполняющей среды. В результате при написании приложений вы обязаны объявлять диапазон версий Windows (скажем, от Windows 10 сборки 17763 до Windows 10 сборки 18362). Таким образом, вам необходимо либо ориентировать приложения на старый API-интерфейс, либо требовать от пользователей установки самого последнего обновления Windows.

Из-за ограничений, вызванных этими различиями, UWP никогда не удавалось сравняться по популярности с WPF и Windows Forms. Для решения проблемы в Microsoft решили превратить UWP в новую технологию под названием Windows App SDK (со слоем пользовательского интерфейса под названием WinUI 3).

Windows App SDK переносит API-интерфейсы WinRT из операционной системы в среду выполнения, предоставляя тем самым полностью управляемый интерфейс и устранивая необходимость ориентироваться на определенный диапазон версий операционной системы. Ниже перечислены его особенности.

- Он лучше интегрируется с API-интерфейсами Windows Desktop (Windows Forms и WPF).
- Он позволяет создавать приложения, которые работают за пределами пе- сочницы Магазина Windows.
- Он функционирует на базе последней версии .NET (вместо привязки к .NET Core 2.2, как в случае с UWP).

Несмотря на такие усовершенствования, WinUI 3 не завоевал такой же популярности, как классические API-интерфейсы Windows Desktop. Кроме того, на момент написания книги Windows App SDK не поддерживал Xbox или HoloLens и требовал отдельной загрузки конечным пользователем.

MAUI

MAUI (ранее Xamarin) позволяет разрабатывать мобильные приложения на C#, предназначенные для сред iOS и Android (а также межплатформенные настольные приложения, ориентированные на macOS и Windows, с помощью Catalyst и Windows App SDK).

Среда CLR и библиотека BCL, работающие под управлением iOS и Android, вместе называются Mono (исполняющая среда, происходящая от Mono с открытым кодом). Исторически сложилось так, что Mono не была полностью совместимой с .NET, а библиотеки, функционирующие как в Mono, так и в .NET, ориентировались на .NET Standard. Тем не менее, начиная с версии .NET 6, открытый интерфейс Mono был объединен с .NET, в результате чего среда Mono по существу стала реализацией .NET.

MAUI включает унифицированный интерфейс проекта, горячую перезагрузку, а также поддержку Blazor Desktop и гибридных приложений. Дополнительную информацию можно получить по ссылке <https://github.com/dotnet/maui>.



Основы .NET

Многие ключевые возможности, необходимые во время программирования, предоставляются не языком C#, а типами в библиотеке .NET BCL. В настоящей главе мы рассмотрим типы, которые помогают решать фундаментальные программные задачи, такие как виртуальное сравнение эквивалентности, сравнение порядка и преобразование типов. Мы также обсудим базовые типы .NET, подобные `String`, `DateTime` и `Enum`.

Описываемые здесь типы находятся в пространстве имен `System` со следующими исключениями:

- тип `StringBuilder` определен в пространстве имен `System.Text`, т.к. относится к типам для *кодировок текста*;
- тип `CultureInfo` и связанные с ним типы определены в пространстве имен `System.Globalization`;
- тип `XmlConvert` определен в пространстве имен `System.Xml`.

Обработка строк и текста

Тип `char`

Тип `char` в C# представляет одиночный символ Unicode и является псевдонимом структуры `System.Char`. В главе 2 было показано, как выражать литералы `char`:

```
char c = 'A';
char newLine = '\n';
```

В структуре `System.Char` определен набор статических методов для работы с символами, в том числе `ToUpper`, `ToLower` и `IsWhiteSpace`. Их можно вызывать либо через тип `System.Char`, либо через его псевдоним `char`:

```
Console.WriteLine (System.Char.ToUpper ('c'));           // C
Console.WriteLine (char.IsWhiteSpace ('\t')));           // True
```

Методы `ToUpper` и `ToLower` учитывают локаль конечного пользователя, что может приводить к неуловимым ошибкам. Следующее выражение дает `false` для турецкой локали:

```
char.ToUpper ('i') == 'I'
```

Причина в том, что в данном случае `char.ToUpper ('i')` равно '`i`' (обратите внимание на точку в верхней части буквы). Чтобы избежать проблем подобного рода, тип `System.Char` (и `System.String`) также предлагает независимые от культуры версии методов `ToUpper` и `ToLower`, имена которых завершаются словом `Invariant`. В них всегда применяются правила культуры английского языка:

```
Console.WriteLine (char.ToUpperInvariant ('i')); // I
```

Это является сокращением для такого кода:

```
Console.WriteLine (char.ToUpper ('i', CultureInfo.InvariantCulture));
```

Дополнительные сведения о локалах и культурах можно найти в разделе “Форматирование и разбор” далее в главе.

Большинство оставшихся статических методов типа `char` имеет отношение к категоризации символов; они перечислены в табл. 6.1.

Таблица 6.1. Статические методы для категоризации символов

Статический метод	Включает символы	Включает категории Unicode
<code>IsLetter</code>	A–Z, a–z и все буквы из других алфавитов	<code>UpperCaseLetter</code> <code>LowerCaseLetter</code> <code>TitleCaseLetter</code> <code>ModifierLetter</code> <code>OtherLetter</code>
<code>IsUpper</code>	Буквы в верхнем регистре	<code>UpperCaseLetter</code>
<code>IsLower</code>	Буквы в нижнем регистре	<code>LowerCaseLetter</code>
<code>IsDigit</code>	0–9 и цифры из других алфавитов	<code>DecimalDigitNumber</code>
<code>IsLetterOrDigit</code>	Буквы и цифры	<code>IsLetter</code> , <code>IsDigit</code>
<code>IsNumber</code>	Все цифры, а также дроби Unicode и числовые символы латинского набора	<code>DecimalDigitNumber</code> <code>LetterNumber</code> <code>OtherNumber</code>
<code>IsSeparator</code>	Пробел и все символы разделителей Unicode	<code>LineSeparator</code> <code>ParagraphSeparator</code>
<code>IsWhiteSpace</code>	Все разделители, а также <code>\n</code> , <code>\r</code> , <code>\t</code> , <code>\f</code> и <code>\v</code>	<code>LineSeparator</code> <code>ParagraphSeparator</code>
<code>IsPunctuation</code>	Символы, используемые для пунктуации в латинском и других алфавитах	<code>DashPunctuation</code> <code>ConnectorPunctuation</code> <code>InitialQuotePunctuation</code> <code>FinalQuotePunctuation</code>
<code>IsSymbol</code>	Большинство других печатаемых символов	<code>MathSymbol</code> <code>ModifierSymbol</code> <code>OtherSymbol</code>
<code>IsControl</code>	Непечатаемые “управляющие” символы с кодами меньше 0x20, такие как <code>\r</code> , <code>\n</code> , <code>\t</code> , <code>\0</code> и символы с кодами между 0x7F и 0x9A	—

Для более детальной категоризации тип `char` предоставляет статический метод по имени `GetUnicodeCategory`; он возвращает перечисление `UnicodeCategory`, члены которого были показаны в правой колонке табл. 6.1.



При явном приведении целочисленного значения вполне возможно получить значение `char`, выходящее за пределы выделенного набора Unicode. Для проверки допустимости символа необходимо вызвать метод `char.GetUnicodeCategory`: если результатом оказывается `UnicodeCategory.OtherNotAssigned`, то символ является недопустимым.

Значение `char` имеет ширину 16 бит, которой достаточно для представления любого символа Unicode в базовой многоязыковой плоскости. Чтобы выйти за ее пределы, потребуется применять суррогатные пары: мы опишем необходимые методы в разделе “Кодировка текста и Unicode” далее в главе.

Тип `string`

Тип `string` (псевдоним класса `System.String`) в C# является неизменяемой последовательностью символов. В главе 2 мы объяснили, как выражать строковые литералы, выполнять сравнения эквивалентности и осуществлять конкатенацию двух строк. В текущем разделе мы рассмотрим остальные функции для работы со строками, которые доступны через статические члены и члены экземпляра класса `System.String`.

Конструирование строк

Простейший способ конструирования строки предусматривает присваивание литерала, как было показано в главе 2:

```
string s1 = "Hello";
string s2 = "First Line\r\nSecond Line";
string s3 = @"\\server\fileshare\helloworld.cs";
```

Чтобы создать повторяющуюся последовательность символов, можно использовать конструктор `string`:

```
Console.Write (new string ('*', 10)); // *****
```

Строку можно также конструировать из массива `char`. Метод `ToCharArray` выполняет обратное действие:

```
char[] ca = "Hello".ToCharArray();
string s = new string (ca); // s = "Hello"
```

Конструктор типа `string` имеет перегруженные версии, которые принимают разнообразные (небезопасные) типы указателей и предназначены для создания строк из таких типов, как `char*`.

Строки `null` и пустые строки

Пустая строка имеет нулевую длину. Чтобы создать пустую строку, можно применить либо литерал, либо статическое поле `string.Empty`; для проверки,

пуста ли строка, можно либо выполнить сравнение эквивалентности, либо просмотреть свойство Length строки:

```
string empty = "";
Console.WriteLine (empty == ""); // True
Console.WriteLine (empty == string.Empty); // True
Console.WriteLine (empty.Length == 0); // True
```

Поскольку строки являются ссылочными типами, они также могут быть null:

```
string nullString = null;
Console.WriteLine (nullString == null); // True
Console.WriteLine (nullString == ""); // False
Console.WriteLine (nullString.Length == 0); // Генерируется исключение
                                            // NullReferenceException
```

Статический метод string.IsNullOrEmpty является удобным сокращением для проверки, равна ли заданная строка null или же является пустой.

Доступ к символам внутри строки

Индексатор строки возвращает одиночный символ по указанному индексу. Как и во всех функциях для работы со строками, индекс начинается с нуля:

```
string str = "abcde";
char letter = str[1]; // letter == 'b'
```

Кроме того, тип string реализует интерфейс `IEnumerable<char>`, так что по символам строки можно проходить с помощью цикла foreach:

```
foreach (char c in "123") Console.Write (c + ","); // 1,2,3,
```

Поиск внутри строк

К простейшим методам поиска внутри строк относятся StartsWith, EndsWith и Contains. Все они возвращают true или false:

```
Console.WriteLine ("quick brown fox".EndsWith ("fox")); // True
Console.WriteLine ("quick brown fox".Contains ("brown")); // True
```

Эти методы перегружены, чтобы позволить указывать член перечисления StringComparison для управления чувствительностью к регистру символов и культуре (см. раздел “Одинарное сравнение или сравнение, чувствительное к культуре” далее в главе). По умолчанию выполняется сопоставление, чувствительное к культуре, с использованием правил, которые применимы к текущей (локализованной) культуре. В следующем случае производится поиск, нечувствительный к регистру символов, с использованием правил, не зависящих от культуры:

```
"abcdef".StartsWith ("aBc", StringComparison.InvariantCultureIgnoreCase)
```

Метод IndexOf возвращает позицию первого вхождения заданного символа или подстроки (или -1, если символ или подстрока не найдена):

```
Console.WriteLine ("abcde".IndexOf ("cd")); // 2
```

Метод `IndexOf` также перегружен для приема значения `startPosition` (индекса, с которого должен начинаться поиск) и перечисления `StringComparison`:

```
Console.WriteLine ("abcde abcde".IndexOf ("CD", 6,  
StringComparison.CurrentCultureIgnoreCase)); // 8
```

Метод `LastIndexOf` похож на `IndexOf`, но перемещается по строке в обратном направлении.

Метод `IndexOfAny` возвращает позицию первого вхождения любого символа из множества символов:

```
Console.Write ("ab,cd ef".IndexOfAny (new char[] { ' ', ',' } )); // 2  
Console.Write ("pas5w0rd".IndexOfAny ("0123456789".ToCharArray() )); // 3
```

Метод `LastIndexOfAny` делает то же самое, но в обратном направлении.

Манипулирование строками

Поскольку класс `String` неизменяемый, все методы, которые “манипулируют” строкой, возвращают новую строку, оставляя исходную строку незатронутой (то же самое происходит при повторном присваивании строковой переменной). Метод `Substring` извлекает порцию строки:

```
string left3 = "12345".Substring (0, 3); // left3 = "123";  
string mid3 = "12345".Substring (1, 3); // mid3 = "234";
```

Если длина не указана, тогда извлекается порция до самого конца строки:

```
string end3 = "12345".Substring (2); // end3 = "345";
```

Методы `Insert` и `Remove` вставляют либо удаляют символы в указанной позиции:

```
string s1 = "helloworld".Insert (5, " , "); // s1 = "hello, world"  
string s2 = s1.Remove (5, 2); // s2 = "helloworld";
```

Методы `PadLeft` и `PadRight` дополняют строку до указанной длины слева или справа заданным символом (или пробелом, если символ не указан):

```
Console.WriteLine ("12345".PadLeft (9, '*')); // ****12345  
Console.WriteLine ("12345".PadLeft (9)); // 12345
```

Если входная строка длиннее, чем длина для дополнения, тогда исходная строка возвращается без изменений.

Методы `TrimStart` и `TrimEnd` удаляют указанные символы из начала или конца строки; метод `Trim` делает то и другое. По умолчанию эти функции удаляют пробельные символы (включая пробелы, табуляции, символы новой строки и их вариации в Unicode):

```
Console.WriteLine (" abc \t\r\n ".Trim().Length); // 3
```

Метод `Replace` заменяет все (неперекрывающиеся) вхождения заданного символа или подстроки:

```
Console.WriteLine ("to be done".Replace (" ", " | ") ); // to | be | done  
Console.WriteLine ("to be done".Replace (" ", "") ); // tobedone
```

Методы `ToUpper` и `ToLower` возвращают версии входной строки в верхнем и нижнем регистре символов. По умолчанию они учитывают текущие языковые настройки у пользователя; методы `ToUpperInvariant` и `ToLowerInvariant` всегда применяют правила, принятые для английского алфавита.

Разделение и объединение строк

Метод `Split` разделяет строку на порции:

```
string[] words = "The quick brown fox".Split();
foreach (string word in words)
    Console.Write (word + "|");                                // The|quick|brown|fox|
```

По умолчанию в качестве разделителей метод `Split` использует пробельные символы; также имеется его перегруженная версия, принимающая массив `params` разделителей типа `char` или `string`. Кроме того, метод `Split` дополнительно принимает перечисление `StringSplitOptions`, в котором предусмотрен вариант для удаления пустых элементов: он полезен, когда слова в строке разделяются несколькими разделителями.

Статический метод `Join` выполняет действие, противоположное `Split`. Он требует указания разделителя и строкового массива:

```
string[] words = "The quick brown fox".Split();
string together = string.Join (" ", words);                // The quick brown fox
```

Статический метод `Concat` похож на `Join`, но принимает только строковый массив `params` и не задействует разделители. Метод `Concat` в точности эквивалентен операции `+` (на самом деле компилятор транслирует операцию `+` в вызов `Concat`):

```
string sentence      = string.Concat ("The", " quick", " brown", " fox");
string sameSentence = "The" + " quick" + " brown" + " fox";
```

Метод `string.Format` и смешанные форматные строки

Статический метод `Format` предлагает удобный способ для построения строк, которые содержат в себе переменные. Эти встроенные переменные (или значения) могут относиться к любому типу; метод `Format` просто вызывает на них `ToString`.

Главная строка, включающая встроенные переменные, называется *смешанной форматной строкой*. При вызове методу `string.Format` предоставляется такая смешанная форматная строка, а за ней по очереди все встроенные переменные:

```
string composite = "It's {0} degrees in {1} on this {2} morning";
string s = string.Format (composite, 35, "Perth", DateTime.Now.DayOfWeek);
// s == "It's 35 degrees in Perth on this Friday morning"
```

Для получения тех же результатов можно применять интерполированные строковые литералы (см. раздел “Строковый тип” в главе 2). Достаточно просто предварить строку символом `$` и поместить выражения в фигурные скобки:

```
string s = $"It's hot this {DateTime.Now.DayOfWeek} morning";
```

Каждое число в фигурных скобках называется *форматным элементом*. Число соответствует позиции аргумента и за ним может дополнительно следовать:

- запятая и *минимальная ширина*;
- двоеточие и *форматная строка*.

Минимальная ширина удобна для выравнивания колонок. Если значение отрицательное, тогда данные выравниваются влево, а иначе — вправо. Например:

```
string composite = "Name={0,-20} Credit Limit={1,15:C}";  
Console.WriteLine (string.Format (composite, "Mary", 500));  
Console.WriteLine (string.Format (composite, "Elizabeth", 20000));
```

Вот как выглядит результат:

Name=Mary	Credit Limit=	\$500.00
Name=Elizabeth	Credit Limit=	\$20,000.00

Ниже приведен эквивалентный код, в котором метод `string.Format` не применяется:

```
string s = "Name=" + "Mary".PadRight (20) +  
          " Credit Limit=" + 500.ToString ("C").PadLeft (15);
```

Значение кредитного лимита (`Credit Limit`) форматируется как денежное посредством форматной строки `"C"`. Форматные строки более подробно рассматриваются в разделе “Форматирование и разбор” далее в главе.

Сравнение строк

При сравнении двух значений в .NET проводится различие между концепциями *сравнения эквивалентности* и *сравнения порядка*. Сравнение эквивалентности проверяет, являются ли два экземпляра семантически одинаковыми; сравнение порядка выясняет, какой из двух экземпляров (если есть) будет следовать первым в случае их расположения по возрастанию или убыванию.



Сравнение эквивалентности не является *подмножеством* сравнения порядка; две системы сравнения имеют разное предназначение. Вполне допустимо иметь, скажем, два неравных значения в одной и той же порядковой позиции. Мы продолжим данную тему в разделе “Сравнение эквивалентности” позже в главе.

Для сравнения эквивалентности строк можно использовать операцию `==` или один из методов `Equals` типа `string`. Последние более универсальны, потому что позволяют указывать такие варианты, как нечувствительность к регистру символов.



Другое отличие связано с тем, что операция `==` не работает надежно со строками, если переменные приведены к типу `object`. Мы объясним это в разделе “Сравнение эквивалентности” далее в главе.

Для сравнения порядка строк можно применять либо метод экземпляра `CompareTo`, либо статические методы `Compare` и `CompareOrdinal`: они возвращают положительное или отрицательное число либо ноль — в зависимости от того, находится первое значение до, после или рядом со вторым.

Прежде чем углубляться в детали каждого вида сравнения, необходимо изучить лежащие в основе .NET алгоритмы сравнения строк.

Ординальное сравнение или сравнение, чувствительное к культуре

При сравнении строк используются два базовых алгоритма: алгоритм *ординального сравнения* и алгоритм сравнения, *чувствительного к культуре*. В случае ординального сравнения символы интерпретируются просто как числа (согласно своим числовым кодам Unicode), а в случае сравнения, чувствительного к культуре, символы интерпретируются со ссылкой на конкретный словарь. Существуют две специальных культуры: "текущая культура", которая основана на настройках, получаемых из панели управления компьютера, и "инвариантная культура", которая является одной и той же на всех компьютерах (и полностью соответствует американской культуре).

Для сравнения эквивалентности удобны оба алгоритма. Однако при упорядочивании сравнение, чувствительное к культуре, почти всегда предпочтительнее: для алфавитного упорядочения строк необходим алфавит. Ординальное сравнение полагается на числовые коды Unicode, которые в силу сложившихся обстоятельств выстраивают английские символы в алфавитном порядке — но не в точности так, как можно было бы ожидать. Например, предполагая включенную чувствительность к регистру символов, рассмотрим строки "Atom", "atom" и "Zamia". В случае инвариантной культуры они располагаются в следующем порядке:

"Atom", "atom", "Zamia"

Тем не менее, при ординальном сравнении результат выглядит так:

"Atom", "Zamia", "atom"

Причина в том, что инвариантная культура инкапсулирует алфавит, в котором символы в верхнем регистре находятся рядом со своими двойниками в нижнем регистре (AaBbCcDd...). Но при ординальном сравнении сначала идут все символы в верхнем регистре, а затем — все символы в нижнем регистре (A...Z, a...z). В сущности, производится возврат к набору символов ASCII, появившемуся в 1960-х годах.

Сравнение эквивалентности для строк

Несмотря на ограничения ординального сравнения, операция == в типе `string` всегда выполняет *ординальное сравнение, чувствительное к регистру*. То же самое касается версии экземпляра метода `string.Equals` в случае вызова без параметров; это определяет "стандартное" поведение сравнения эквивалентности для типа `string`.



Алгоритм ординального сравнения выбран для функций операции == и Equals типа `string` из-за того, что он высокоэффективен и детерминирован. Сравнение эквивалентности строк считается фундаментальной операцией и выполняется гораздо чаще, чем сравнение порядка.

"Строгое" понятие эквивалентности также согласуется с общим применением операции ==.

Следующие методы позволяют выполнять сравнение с учетом культуры или сравнение, нечувствительное к регистру:

```
public bool Equals (string value, StringComparison comparisonType);  
public static bool Equals (string a, string b,  
                           StringComparison comparisonType);
```

Статическая версия полезна тем, что она работает в случае, когда одна или обе строки равны null. Перечисление StringComparison определено так, как показано ниже:

```
public enum StringComparison  
{  
    CurrentCulture,                      // Чувствительное к регистру  
    CurrentCultureIgnoreCase,             // Чувствительное к регистру  
    InvariantCulture,                    // Чувствительное к регистру  
    InvariantCultureIgnoreCase,          // Чувствительное к регистру  
    Ordinal,                            // Чувствительное к регистру  
    OrdinalIgnoreCase  
}
```

Вот примеры:

```
Console.WriteLine (string.Equals ("foo", "FOO",  
                                 StringComparison.OrdinalIgnoreCase)); // True  
Console.WriteLine ("ú" == "ú");                         // False  
Console.WriteLine (string.Equals ("ú", "ú",  
                                 StringComparison.CurrentCulture)); // ?
```

(Результат в третьем операторе определяется текущими настройками языка на компьютере.)

Сравнение порядка для строк

Метод экземпляра CompareTo типа string выполняет чувствительное к культуре и регистру сравнение порядка. В отличие от операции == метод CompareTo не использует ординальное сравнение: для упорядочивания намного более полезен алгоритм сравнения, чувствительного к культуре.

Вот определение метода CompareTo:

```
public int CompareTo (string strB);
```



Метод экземпляра CompareTo реализует обобщенный интерфейс IComparable, который является стандартным протоколом сравнения, повсеместно применяемым в библиотеках .NET. Это значит, что метод CompareTo типа string определяет строки со стандартным поведением упорядочивания в таких реализациях, как сортированные коллекции, например. За дополнительной информацией по IComparable обращайтесь в раздел “Сравнение порядка” далее в главе.

Для других видов сравнения можно вызывать статические методы Compare и CompareOrdinal:

```
public static int Compare (string strA, string strB,
                         StringComparison comparisonType);
public static int Compare (string strA, string strB, bool ignoreCase,
                         CultureInfo culture);
public static int Compare (string strA, string strB, bool ignoreCase);
public static int CompareOrdinal (string strA, string strB);
```

Последние два метода являются просто сокращениями для вызова первых двух методов. Все методы сравнения порядка возвращают положительное число, отрицательное число или ноль в зависимости от того, как расположено первое значение относительно второго — до, после или рядом:

```
Console.WriteLine ("Boston".CompareTo ("Austin"));           // 1
Console.WriteLine ("Boston".CompareTo ("Boston"));           // 0
Console.WriteLine ("Boston".CompareTo ("Chicago"));          // -1
Console.WriteLine ("ü".CompareTo ("ū"));                      // 0
Console.WriteLine ("foo".CompareTo ("FOO"));                 // -1
```

В следующем операторе выполняется нечувствительное к регистру сравнение с использованием текущей культуры:

```
Console.WriteLine (string.Compare ("foo", "FOO", true));    // 0
```

Предоставляя объект `CultureInfo`, можно подключить любой алфавит:

```
// Класс CultureInfo определен в пространстве имен System.Globalization
CultureInfo german = CultureInfo.GetCultureInfo ("de-DE");
int i = string.Compare ("Müller", "Muller", false, german);
```

Класс `StringBuilder`

Класс `StringBuilder` (из пространства имен `System.Text`) представляет изменяемую (редактируемую) строку. С помощью `StringBuilder` можно добавлять (`Append`), вставлять (`Insert`), удалять (`Remove`) и заменять (`Replace`) подстроки, не заменяя целиком объект `StringBuilder`.

Конструктор `StringBuilder` дополнительно принимает начальное строковое значение, а также стартовый размер для внутренней емкости (по умолчанию 16 символов). Если начальная емкость превышена, тогда `StringBuilder` автоматически изменяет размеры своих внутренних структур (за счет небольшого снижения производительности) вплоть до максимальной емкости (которая по умолчанию составляет `int.MaxValue`).

Популярное применение класса `StringBuilder` связано с построением длинной строки путем повторяющихся вызовов `Append`. Такой подход намного более эффективен, чем выполнение множества конкатенаций обычных строковых типов:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 50; i++) sb.Append (i + ",");
```

Для получения финального результата необходимо вызвать метод `ToString`:

```
Console.WriteLine (sb.ToString());
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
```

Метод AppendLine выполняет добавление последовательности новой строки ("\\r\\n" в Windows). Он принимает смешанную форматную строку в точности как метод String.Format.

Помимо методов Insert, Remove и Replace (метод Replace функционирует подобно методу Replace в типе string) в классе StringBuilder определено свойство Length и записываемый индексатор для получения/установки отдельных символов.

Для очистки содержимого StringBuilder необходимо либо создать новый экземпляр StringBuilder, либо установить его свойство Length в 0.



Установка свойства Length экземпляра StringBuilder в 0 не сокращает его *внутреннюю* емкость. Таким образом, если экземпляр StringBuilder ранее содержал один миллион символов, то после обнуления его свойства Length он продолжит занимать около 2 Мбайт памяти. Чтобы освободить эту память, потребуется создать новый экземпляр StringBuilder и дать возможность старому покинуть область видимости (и подвергнуться сборке мусора).

Кодировка текста и Unicode

Набор символов — это распределение символов, с каждым из которых связан числовой код или *кодовая точка* (code point). Чаще всего используются два набора символов: Unicode и ASCII. Набор Unicode имеет адресное пространство для примерно миллиона символов, из которых в настоящее время распределено около 100 000. Набор Unicode охватывает самые распространенные в мире языки, а также ряд исторических языков и специальных символов. Набор ASCII — это просто первые 128 символов набора Unicode, которые покрывают большинство из того, что можно видеть на английской клавиатуре. Набор ASCII появился на 30 лет раньше Unicode и продолжает временами применяться из-за своей простоты и эффективности: каждый его символ представлен одним байтом.

Система типов .NET предназначена для работы с набором символов Unicode. Тем не менее, набор ASCII поддерживается неявно в силу того, что является подмножеством Unicode.

Кодировка текста отображает числовые кодовые точки символов на их двоичные представления. В .NET кодировки текста вступают в игру главным образом при работе с текстовыми файлами или потоками. Когда текстовый файл читается с помещением в строку, *кодировщик текста* транслирует данные файла из двоичного представления во внутреннее представление Unicode, которое ожидают типы char и string. Кодировка текста может ограничивать множество представляемых символов, а также влиять на эффективность хранения.

В .NET есть две категории кодировок текста:

- кодировки, которые отображают символы Unicode на другой набор символов;
- кодировки, которые используют стандартные схемы кодирования Unicode.

Первая категория содержит унаследованные кодировки, такие как IBM EBCDIC и 8-битные наборы символов с расширенными символами в области из 128 старших символов, которые были популярны до появления Unicode (они идентифицируются кодовой страницей). Кодировка ASCII тоже относится к данной категории: она кодирует первые 128 символов и отбрасывает остальные. Вдобавок эта категория содержит традиционную кодировку GB18030 — обязательный стандарт для приложений, которые написаны в Китае (или предназначены для продажи в Китае), начиная с 2000 года.

Во второй категории находятся кодировки UTF-8, UTF-16 и UTF-32 (и устаревшая UTF-7). Каждая из них отличается требованиями к пространству. Кодировка UTF-8 наиболее эффективна с точки зрения пространства для большинства видов текста: при представлении каждого символа в ней применяется от одного до четырех байтов. Первые 128 символов требуют только одного байта, делая эту кодировку совместимой с ASCII. Кодировка UTF-8 чаще всего используется в текстовых файлах и потоках (особенно в Интернете), к тому же она стандартно применяется для потокового ввода-вывода в .NET (фактически она является стандартом почти для всего, что неявно использует кодировку).

Кодировка UTF-16 для представления каждого символа применяет одно или два 16-битных слова и используется внутри .NET для представления символов и строк. Некоторые программы также записывают файлы в UTF-16.

Кодировка UTF-32 наименее экономична в плане пространства: она отображает каждую кодовую точку на 32 бита, т.е. любой символ потребляет четыре байта. По указанной причине кодировка UTF-32 применяется редко. Однако она существенно упрощает произвольный доступ, поскольку каждый символ занимает одинаковое количество байтов.

Получение объекта Encoding

Класс Encoding в пространстве имен System.Text — это общий базовый класс для классов, инкапсулирующих кодировки текста. Существует несколько подклассов, назначение которых заключается в инкапсуляции семейств кодировок с похожими возможностями. Наиболее распространенные кодировки также могут быть получены через выделенные статические свойства класса Encoding:

Название кодировки	Статическое свойство класса Encoding
UTF-8	Encoding.UTF8
UTF-16	Encoding.Unicode (не UTF16)
UTF-32	Encoding.UTF32
ASCII	Encoding.ASCII

Другие кодировки можно получить с помощью вызова метода Encoding.GetEncoding со стандартным именем набора символов IANA (Internet Assigned Numbers Authority — Комитет по цифровым адресам в Интернете):

```
// В .NET 5+ и .NET Core сначала должен быть вызван метод RegisterProvider:  
Encoding.RegisterProvider (CodePagesEncodingProvider.Instance);  
Encoding chinese = Encoding.GetEncoding ("GB18030");
```

Статический метод GetEncodings возвращает список всех поддерживаемых кодировок с их стандартными именами IANA:

```
foreach (EncodingInfo info in Encoding.GetEncodings())
    Console.WriteLine (info.Name);
```

Другой способ получения кодировки предполагает создание экземпляра класса кодировки напрямую. В таком случае появляется возможность устанавливать различные варианты через аргументы конструктора, включая описанные ниже.

- Должно ли генерироваться исключение, если при декодировании встречается недопустимая последовательность байтов. По умолчанию исключение не генерируется.
- Должно ли производиться кодирование/декодирование UTF-16/UTF-32 с наиболее значащими байтами в начале (*обратный порядок байтов*) или с наименее значащими байтами в начале (*прямой порядок байтов*). По умолчанию принимается прямой порядок байтов, который является стандартом в операционной системе Windows.
- Должен ли выдаваться маркер порядка байтов (префикс, указывающий конкретный *порядок следования байтов*).

Кодировка для файлового и потокового ввода-вывода

Самое распространенное использование объекта Encoding связано с управлением способом чтения и записи в файл или поток. Например, следующий код записывает строку "Testing..." в файл по имени data.txt с кодировкой UTF-16:

```
System.IO.File.WriteAllText ("data.txt", "Testing...", Encoding.Unicode);
```

Если опустить последний аргумент, тогда метод WriteAllText применит ведущую кодировку UTF-8.



UTF-8 является стандартной кодировкой текста для всего файлового и потокового ввода-вывода.

Мы еще вернемся к данной теме в разделе "АдAPTERЫ ПОТОКОВ" главы 15.

Кодировка для байтовых массивов

Объект Encoding можно также использовать для работы с байтовым массивом. Метод GetBytes преобразует string в byte[] с применением заданной кодировки, а метод GetString преобразует byte[] в string:

```
byte[] utf8Bytes = System.Text.Encoding.UTF8.GetBytes ("0123456789");
byte[] utf16Bytes = System.Text.Encoding.Unicode.GetBytes ("0123456789");
byte[] utf32Bytes = System.Text.Encoding.UTF32.GetBytes ("0123456789");

Console.WriteLine (utf8Bytes.Length); // 10
Console.WriteLine (utf16Bytes.Length); // 20
Console.WriteLine (utf32Bytes.Length); // 40

string original1 = System.Text.Encoding.UTF8.GetString (utf8Bytes);
string original2 = System.Text.Encoding.Unicode.GetString (utf16Bytes);
string original3 = System.Text.Encoding.UTF32.GetString (utf32Bytes);
```

```
Console.WriteLine (original1);          // 0123456789
Console.WriteLine (original2);          // 0123456789
Console.WriteLine (original3);          // 0123456789
```

UTF-16 и суррогатные пары

Вспомните, что символы и строки в .NET хранятся в кодировке UTF-16. Поскольку UTF-16 требует одного или двух 16-битных слов на символ, а тип `char` имеет длину только 16 бит, то некоторые символы Unicode требуют для своего представления два экземпляра `char`. Отсюда пара следствий:

- свойство `Length` строки может иметь более высокое значение, чем фактическое количество символов;
- одиночного символа `char` не всегда достаточно для полного представления символа Unicode.

Большинство приложений игнорируют указанные следствия, потому что почти все часто используемые символы попадают внутрь раздела Unicode, который называется *базовой многоязыковой плоскостью* (Basic Multilingual Plane — BMP) и требует только одного 16-битного слова в UTF-16. Плоскость BMP охватывает десятки мировых языков и включает свыше 30 000 китайских иероглифов. Исключениями являются символы ряда древних языков, символы для записи музыкальных произведений и некоторые менее распространенные китайские иероглифы.

Если необходимо поддерживать символы из двух слов, то следующие статические методы в `char` преобразуют 32-битную кодовую точку в строку из двух `char` и наоборот:

```
string ConvertFromUtf32 (int utf32)
int    ConvertToUtf32  (char highSurrogate, char lowSurrogate)
```

Символы из двух слов называются *суррогатными*. Их легко обнаружить, поскольку каждое слово находится в диапазоне от `0xD800` до `0xDFFF`. Для помощи в этом можно воспользоваться следующими статическими методами в `char`:

```
bool IsSurrogate      (char c)
bool IsHighSurrogate (char c)
bool IsLowSurrogate  (char c)
bool IsSurrogatePair (char highSurrogate, char lowSurrogate)
```

Класс `StringInfo` из пространства имен `System.Globalization` также предлагает ряд методов и свойств для работы с символами из двух слов.

Символы за пределами плоскости BMP обычно требуют специальных шрифтов и имеют ограниченную поддержку в операционных системах.

Дата и время

За работу по представлению даты и времени отвечают следующие неизменяемые структуры из пространства имен `System`:

```
DateTime, DateTimeOffset, TimeSpan, DateOnly, TimeOnly
```

В языке C# отсутствуют специальные ключевые слова, которые отображались бы на указанные типы.

Структура TimeSpan

Структура TimeSpan представляет временной интервал или время суток. В последней роли это просто “часы” (не имеющие даты), которые эквивалентны времени, прошедшему с полуночи, в предположении, что нет перехода на летнее время. Разрешающая способность TimeSpan составляет 100 нс, максимальное значение примерно соответствует 10 млн дней, а значение может быть положительным или отрицательным.

Есть три способа конструирования TimeSpan:

- с помощью одного из конструкторов;
- путем вызова одного из статических методов From...;
- за счет вычитания одного экземпляра DateTime из другого.

Ниже перечислены доступные конструкторы:

```
public TimeSpan (int hours, int minutes, int seconds);
public TimeSpan (int days, int hours, int minutes, int seconds);
public TimeSpan (int days, int hours, int minutes, int seconds,
                 int milliseconds);
public TimeSpan (int days, int hours, int minutes, int seconds,
                 int milliseconds, int microseconds);
public TimeSpan (long ticks);           // Каждый тик равен 100 нс
```

Статические методы From... более удобны, когда необходимо указать интервал в каких-то одних единицах, скажем, минутах, часах и т.д.:

```
public static TimeSpan FromDays (double value);
public static TimeSpan FromHours (double value);
public static TimeSpan FromMinutes (double value);
public static TimeSpan FromSeconds (double value);
public static TimeSpan FromMilliseconds (double value);
public static TimeSpan FromMicroseconds (double value);
```

Например:

```
Console.WriteLine (new TimeSpan (2, 30, 0));           // 02:30:00
Console.WriteLine (TimeSpan.FromHours (2.5));          // 02:30:00
Console.WriteLine (TimeSpan.FromHours (-2.5));         // -02:30:00
```

В структуре TimeSpan перегружены операции < и >, а также операции + и -. В результате вычисления приведенного ниже выражения получается значение TimeSpan, соответствующее 2,5 часам:

```
TimeSpan.FromHours(2) + TimeSpan.FromMinutes(30);
```

Следующее выражение дает в результате значение, которое на одну секунду короче 10 дней:

```
TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1);      // 9.23:59:59
```

С помощью приведенного ниже выражения можно проиллюстрировать работу целочисленных свойств Days, Hours, Minutes, Seconds и Milliseconds:

```
TimeSpan nearlyTenDays = TimeSpan.FromDays(10) - TimeSpan.FromSeconds(1);
Console.WriteLine (nearlyTenDays.Days);                  // 9
```

```
Console.WriteLine (nearlyTenDays.Hours); // 23
Console.WriteLine (nearlyTenDays.Minutes); // 59
Console.WriteLine (nearlyTenDays.Seconds); // 59
Console.WriteLine (nearlyTenDays.Milliseconds); // 0
```

Напротив, свойства Total... возвращают значения типа double, описывающие промежуток времени целиком:

```
Console.WriteLine (nearlyTenDays.TotalDays); // 9.99998842592593
Console.WriteLine (nearlyTenDays.TotalHours); // 239.999722222222
Console.WriteLine (nearlyTenDays.TotalMinutes); // 14399.9833333333
Console.WriteLine (nearlyTenDays.TotalSeconds); // 863999
Console.WriteLine (nearlyTenDays.TotalMilliseconds); // 863999000
```

Статический метод Parse представляет собой противоположность ToString, преобразуя строку в значение TimeSpan. Метод TryParse делает то же самое, но в случае неудачного преобразования возвращает false вместо генерации исключения. Класс XmlConvert также предоставляет методы для преобразования между TimeSpan и string, которые следуют стандартным протоколам форматирования XML.

Стандартным значением для TimeSpan является TimeSpan.Zero.

Структуру TimeSpan также можно применять для представления времени суток (времени, прошедшего с полуночи). Для получения текущего времени суток необходимо обратиться к свойству DateTime.Now.TimeOfDay.

Структуры DateTime и DateTimeOffset

Типы DateTime и DateTimeOffset — это неизменяемые структуры для представления даты и дополнительно времени. Их разрешающая способность составляет 100 нс, а поддерживаемый диапазон лет — от 0001 до 9999.

Структура DateTimeOffset функционально похожа на DateTime. Ее отличительной особенностью является сохранение также смещения UTC (Coordinated Universal Time — универсальное глобальное время), что позволяет получать более осмысленные результаты при сравнении значений из разных часовых поясов.

Выбор между DateTime и DateTimeOffset

Структуры DateTime и DateTimeOffset отличаются способом обработки часовых поясов. Структура DateTime содержит флаг с тремя состояниями, который указывает, относительно чего отсчитывается значение DateTime:

- местное время на текущем компьютере;
- UTC (современный эквивалент среднего времени по Гринвичу (Greenwich Mean Time));
- не определено.

Структура DateTimeOffset более специфична — она хранит смещение UTC в виде TimeSpan:

July 01 2019 03:00:00 -06:00

Это влияет на сравнения эквивалентности, которые являются главным фактором при выборе между `DateTime` и `DateTimeOffset`. В частности:

- во время сравнения `DateTime` игнорирует флаг с тремя состояниями и считает, что два значения равны, если они имеют одинаковый год, месяц, день, часы, минуты и т.д.;
- структура `DateTimeOffset` считает два значения равными, если они ссылаются на *ту же самую точку во времени*.



Переход на летнее время может сделать указанные различия важными, даже если приложение не нуждается в учете множества географических часовых поясов.

Таким образом, `DateTime` считает следующие два значения отличающимися, тогда как `DateTimeOffset` — равными:

July 01 2019 09:00:00 +00:00 (GMT)

July 01 2019 03:00:00 -06:00 (местное время, Центральная Америка)

В большинстве случаев логика эквивалентности `DateTimeOffset` предпочтительнее. Скажем, при выяснении, какое из двух международных событий произошло позже, структура `DateTimeOffset` даст полностью корректный ответ. Аналогично хакер, планирующий атаку типа распределенного отказа в обслуживании, определенно выберет `DateTimeOffset`! Чтобы сделать то же самое с помощью `DateTime`, в приложении потребуется повсеместное приведение к единому часовому поясу (обычно UTC). Такой подход проблематичен по двум причинам.

- Для обеспечения дружественности к конечному пользователю UTC-значения `DateTime` перед форматированием требуют явного преобразования в местное время.
- Довольно легко забыть и случайно воспользоваться местным значением `DateTime`.

Однако структура `DateTime` лучше при указании значения относительно локального компьютера во время выполнения — например, если нужно запланировать архивацию в каждом международном офисе на следующее воскресенье в 3 утра местного времени (когда наблюдается минимальная активность). В такой ситуации больше подойдет структура `DateTime`, т.к. она будет отражать местное время на каждой площадке.



Внутренне структура `DateTimeOffset` для хранения смещения UTC в минутах применяет короткий целочисленный тип. Она не хранит никакой региональной информации, так что ничего не указывает на то, к какому времени относится смещение +08:00, например, в Сингапуре или в Перте (Западная Австралия).

Мы рассмотрим часовые пояса и сравнение эквивалентности более подробно в разделе “Даты и часовые пояса” далее в главе.



В SQL Server 2008 была введена прямая поддержка `DateTimeOffset` через новый тип данных с таким же именем.

Конструирование `DateTime`

В структуре `DateTime` определены конструкторы, которые принимают целочисленные значения для года, месяца и дня, а также дополнительно для часов, минут, секунд, миллисекунд и микросекунд (начиная с версии .NET 7):

```
public DateTime (int year, int month, int day);
public DateTime (int year, int month, int day,
                 int hour, int minute, int second, int millisecond);
```

Если указывается только дата, то время неявно устанавливается в полночь (0:00). Конструкторы `DateTime` также позволяют задавать `DateTimeKind` — перечисление со следующими значениями:

`Unspecified`, `Local`, `Utc`

Оно соответствует флагу с тремя состояниями, который был описан в предыдущем разделе. `Unspecified` принимается по умолчанию и означает, что `DateTime` не зависит от часового пояса. `Local` означает отношение к местному часовому поясу, установленному на текущем компьютере. Такой экземпляр `DateTime` не содержит информации о конкретном часовом поясе и в отличие от `DateTimeOffset` не хранит числовое смещение UTC.

Свойство `Kind` в `DateTime` возвращает значение `DateTimeKind`.

Конструкторы `DateTime` также перегружены с целью приема объекта `Calendar`, что позволяет указывать дату с использованием подклассов `Calendar`, которые определены в пространстве имен `System.Globalization`:

```
DateTime d =
    new DateTime(5767, 1, 1, new System.Globalization.HebrewCalendar());
Console.WriteLine (d); // 12/12/2006 12:00:00 AM
```

(Форматирование даты в этом примере зависит от настроек в панели управления на компьютере.) Структура `DateTime` всегда работает со стандартным григорианским календарем — в приведенном примере во время конструирования происходит однократное преобразование. Для выполнения вычислений с применением другого календаря вы должны применять методы на самом подклассе `Calendar`.

Конструировать экземпляр `DateTime` можно также с единственным значением *тиков* типа `long`, где *тики* представляют собой количество 100 наносекундных интервалов, прошедших с полуночи 01/01/0001.

Для целей взаимодействия `DateTime` предоставляет статические методы `FromFileTime` и `FromFileTimeUtc`, которые обеспечивают преобразование времени файлов Windows (указанного как `long`), и статический метод `FromOADate`, преобразующий дату/время OLE Automation (типа `double`).

Чтобы сконструировать экземпляр `DateTime` из строки, понадобится вызвать статический метод `Parse` или `ParseExact`. Оба метода принимают дополнительные флаги и поставщики форматов; `ParseExact` также принимает форматную строку. Мы обсудим разбор более детально в разделе “Форматирование и разбор” далее в главе.

Конструирование DateTimeOffset

Структура `DateTimeOffset` имеет похожий набор конструкторов. Отличие в том, что также указывается смещение UTC в виде `TimeSpan`:

```
public DateTimeOffset (int year, int month, int day,
                      int hour, int minute, int second,
                      TimeSpan offset);

public DateTimeOffset (int year, int month, int day,
                      int hour, int minute, int second, int millisecond,
                      TimeSpan offset);
```

Значение `TimeSpan` должно составлять целое количество минут, иначе генерируется исключение.

В добавок структура `DateTimeOffset` имеет конструкторы, которые принимают объект `Calendar` и значение `тиков` типа `long`, а также статические методы `Parse` и `ParseExact`, принимающие строку.

Конструировать экземпляр `DateTimeOffset` можно из существующего экземпляра `DateTime` либо с использованием перечисленных ниже конструкторов:

```
public DateTimeOffset (DateTime dateTime);
public DateTimeOffset (DateTime dateTime, TimeSpan offset);
```

либо с помощью неявного приведения:

```
DateTimeOffset dt = new DateTime (2000, 2, 3);
```



Неявное приведение `DateTime` к `DateTimeOffset` удобно тем, что в большей части библиотеки .NET BCL поддерживается тип `DateTime` — не `DateTimeOffset`.

Если смещение не указано, тогда оно выводится из значения `DateTime` с применением следующих правил:

- если `DateTime` имеет значение `DateTimeKind`, равное `Utc`, то смещение равно нулю;
- если `DateTime` имеет значение `DateTimeKind`, равное `Local` или `Unspecified` (по умолчанию), то смещение берется из текущего часового пояса.

Для преобразования в другом направлении структура `DateTimeOffset` предлагает три свойства, которые возвращают значения типа `DateTime`:

- свойство `UtcDateTime` возвращает экземпляр `DateTime`, представленный как время UTC;
- свойство `LocalDateTime` возвращает экземпляр `DateTime` в текущем часовом поясе (при необходимости преобразованный);
- свойство `DateTime` возвращает экземпляр `DateTime` в любом часовом поясе, который был указан, со свойством `Kind`, равным `Unspecified` (т.е. возвращает время UTC плюс смещение).

Текущие значения DateTime/DateTimeOffset

Обе структуры DateTime и DateTimeOffset имеют статическое свойство Now, которое возвращает текущую дату и время:

```
Console.WriteLine (DateTime.Now);           // 11/11/2019 1:23:45 PM  
Console.WriteLine (DateTimeOffset.Now);     // 11/11/2019 1:23:45 PM -06:00
```

Структура DateTime также предоставляет свойство Today, возвращающее порцию даты:

```
Console.WriteLine (DateTime.Today);         // 11/11/2019 12:00:00 AM
```

Статическое свойство UtcNow возвращает текущую дату и время в UTC:

```
Console.WriteLine (DateTime.UtcNow);        // 11/11/2019 7:23:45 AM  
Console.WriteLine (DateTimeOffset.UtcNow);    // 11/11/2019 7:23:45 AM +00:00
```

Точность всех этих методов зависит от операционной системы и обычно находится в пределах 10–20 мс.

Работа с датой и временем

Структуры DateTime и DateTimeOffset предлагают похожий набор свойств экземпляра, которые возвращают элементы даты/времени:

```
DateTime dt = new DateTime (2000, 2, 3,  
                           10, 20, 30);  
  
Console.WriteLine (dt.Year);                // 2000  
Console.WriteLine (dt.Month);               // 2  
Console.WriteLine (dt.Day);                 // 3  
Console.WriteLine (dt.DayOfWeek);          // Thursday  
Console.WriteLine (dt.DayOfYear);          // 34  
  
Console.WriteLine (dt.Hour);                // 10  
Console.WriteLine (dt.Minute);              // 20  
Console.WriteLine (dt.Second);              // 30  
Console.WriteLine (dt.Millisecond);         // 0  
Console.WriteLine (dt.Ticks);               // 630851700300000000  
Console.WriteLine (dt.TimeOfDay);           // 10:20:30 (возвращает TimeSpan)
```

Структура DateTimeOffset также имеет свойство Offset типа TimeSpan.

Оба типа предоставляют указанные ниже методы экземпляра, которые предназначены для выполнения вычислений (большинство из них принимают аргумент типа double или int):

```
AddYears, AddMonths, AddDays,  
AddHours, AddMinutes, AddSeconds, AddMilliseconds, AddTicks
```

Все они возвращают новый экземпляр DateTime или DateTimeOffset и учитывают такие аспекты, как високосный год. Для вычитания можно передавать отрицательное значение.

Метод Add добавляет TimeSpan к DateTime или DateTimeOffset. Операция + перегружена для выполнения той же работы:

```
TimeSpan ts = TimeSpan.FromMinutes (90);  
Console.WriteLine (dt.Add (ts));  
Console.WriteLine (dt + ts);                // то же, что и выше
```

Можно также вычитать `TimeSpan` из `DateTime/DateTimeOffset` и вычитать один экземпляр `DateTime/DateTimeOffset` из другого. Последнее действие дает в результате `TimeSpan`:

```
DateTime thisYear = new DateTime(2015, 1, 1);
DateTime nextYear = thisYear.AddYears(1);
TimeSpan oneYear = nextYear - thisYear;
```

Форматирование и разбор даты и времени

Вызов метода `ToString` на экземпляре `DateTime` обеспечивает форматирование результата в виде *краткой даты* (все числа), за которой следует *полное время* (включающее секунды). Например:

```
11/11/2019 11:50:30 AM
```

По умолчанию панель управления операционной системы определяет аспекты вроде того, что должно указываться первым — день, месяц или год, должны ли использоваться ведущие нули и какой формат суток применяется (12- или 24-часовой).

Вызов метода `ToString` на `DateTimeOffset` дает то же самое, но в добавок возвращает и смещение:

```
11/11/2019 11:50:30 AM -06:00
```

Методы `ToShortDateString` и `ToLongDateString` возвращают только часть, касающуюся даты. Формат полной даты также определяется в панели управления; примером может служить “Monday, 11 November 2019”. Методы `ToShortTimeString` и `ToLongTimeString` возвращают только часть, касающуюся времени, скажем, 17:10:10 (`ToShortTimeString` не включает секунды).

Четыре только что описанных метода в действительности являются сокращениями для четырех разных *форматных строк*. Метод `ToString` перегружен для приема форматной строки и поставщика, позволяя указывать широкий диапазон вариантов и управлять применением региональных настроек. Мы рассмотрим эти методы более подробно в разделе “Форматирование и разбор” далее в главе.



Значения `DateTime` и `DateTimeOffset` могут быть некорректно разобраны, если текущие настройки культуры отличаются от настроек, использованных при форматировании. Такой проблемы можно избежать, применяя метод `ToString` вместе с форматной строкой, которая игнорирует настройки культуры (скажем, “o”):

```
DateTime dt1 = DateTime.Now;
string cannotBeMisparsed = dt1.ToString("o");
DateTime dt2 = DateTime.Parse(cannotBeMisparsed);
```

Статические методы `Parse/TryParse` и `ParseExact/TryParseExact` выполняют действие, противоположное методу `ToString` — преобразуют строку в `DateTime` или в `DateTimeOffset`. Данные методы также имеют перегруженные версии, которые принимают поставщик формата. Вместо генерации исключения `FormatException` методы `Try...` возвращают значение `false`.

Значения `DateTime` и `DateTimeOffset`, равные `null`

Поскольку `DateTime` и `DateTimeOffset` являются структурами, они по своей сути не способны принимать значение `null`. Когда требуется возможность иметь значение `null`, существуют два способа добиться цели:

- использовать тип, допускающий `null` (т.е. `DateTime?` или `DateTimeOffset?`);
- применять статическое поле `DateTime.MinValue` или `DateTimeOffset.MinValue` (*стандартные значения для этих типов*).

Использование типов, допускающих `null`, обычно является более предпочтительным подходом, поскольку в таком случае компилятор помогает предотвращать ошибки. Поле `DateTime.MinValue` полезно для обеспечения обратной совместимости с кодом, написанным до выхода версии C# 2.0 (где появились типы, допускающие `null`).



Вызов метода `ToUniversalTime` или `ToLocalTime` на `DateTime.MinValue` может дать в результате значение, не являющееся `DateTime.MinValue` (в зависимости от того, по какую сторону от Гринвича вы находитесь). Если вы находитесь прямо на Гринвиче (Англия в период, когда летнее время не действует), то данная проблема не возникает, потому что местное время и UTC совпадают. Считайте это компенсацией за английскую зиму.

Структуры `DateOnly` и `TimeOnly`

Структуры `DateOnly` и `TimeOnly` (появившиеся в .NET 6) необходимы, когда нужно представлять *только* дату или время.

Структура `DateOnly` похожа на `DateTime`, но без компонента времени. Кроме того, структура `DateOnly` не содержит значение перечисления `DateTimeKind`; по существу оно всегда равно `Unspecified`, а концепции `Local` или `Utc` отсутствуют. Исторически в качестве альтернативы `DateOnly` применялась структура `DateTime` с нулевым временем (полночью). Трудность такого подхода заключается в том, что сравнения на эквивалентность терпят неудачу при появлении ненулевого времени.

Структура `TimeOnly` аналогична `DateTime`, но без компонента даты. Она предназначена для регистрации времени суток и подходит для таких приложений, как фиксация времени будильника или часов работы.

Даты и часовые пояса

В данном разделе мы более детально исследуем влияние часовых поясов на типы `DateTime` и `DateTimeOffset`. Мы также рассмотрим типы `TimeZone` и `TimeZoneInfo`, которые предоставляют информацию по смещениям часовых поясов и переходу на летнее время.

Структура DateTime и часовые пояса

Структура DateTime обрабатывает часовые пояса упрощенным образом. Внутренне DateTime состоит из двух порций информации:

- 62-битное число, которое указывает количество тиков, прошедших с момента 1/1/0001;
- 2-битное значение перечисления DateTimeKind (Unspecified, Local или Utc).

При сравнении двух экземпляров DateTime сравниваются только их значения тиков, а значения DateTimeKind игнорируются:

```
DateTime dt1 = new DateTime (2000, 1, 1, 10, 20, 30, DateTimeKind.Local);
DateTime dt2 = new DateTime (2000, 1, 1, 10, 20, 30, DateTimeKind.Utc);
Console.WriteLine (dt1 == dt2);                                // True
DateTime local = DateTime.Now;
DateTime utc = local.ToUniversalTime();
Console.WriteLine (local == utc);                            // False
```

Методы экземпляра ToUniversalTime/ToLocalTime выполняют преобразование в универсальное/местное время. Они применяют текущие настройки часовогопояса компьютера и возвращают новый экземпляр DateTime со значением DateTimeKind, равным Utc или Local. При вызове ToUniversalTime на экземпляре DateTime, уже являющемся Utc, или ToLocalTime на экземпляре DateTime, который уже представляет собой Local, никакие преобразования не выполняются. Тем не менее, в случае вызова ToUniversalTime или ToLocalTime на экземпляре DateTime, являющемся Unspecified, преобразование произойдет.

С помощью статического метода DateTime.SpecifyKind можно конструировать экземпляр DateTime, который будет отличаться от других только значением поля Kind:

```
DateTime d = new DateTime (2020, 12, 12);    // Unspecified
DateTime utc = DateTime.SpecifyKind (d, DateTimeKind.Utc);
Console.WriteLine (utc);                      // 12/12/2020 12:00:00 AM
```

Структура DateTimeOffset и часовые пояса

Структура DateTimeOffset внутри содержит поле DateTime, значение которого всегда представлено как UTC, и 16-битное целочисленное поле Offset для смещения UTC, выраженного в минутах. Операции сравнения имеют дело только с полем DateTime (UTC); поле Offset используется главным образом для форматирования.

Методы ToUniversalTime/ToLocalTime возвращают экземпляр DateTimeOffset, представляющий один и тот же момент времени, но в UTC или местном времени. В отличие от DateTime эти методы не воздействуют на лежащее в основе значение даты/времени, а только на смещение:

```
DateTimeOffset local = DateTimeOffset.Now;
DateTimeOffset utc = local.ToUniversalTime();
Console.WriteLine (local.Offset);                  // -06:00:00 (в Центральной Америке)
```

```
Console.WriteLine (utc.Offset);           // 00:00:00
Console.WriteLine (local == utc);         // True
```

Чтобы включить в сравнение поле Offset, необходимо применить метод EqualsExact:

```
Console.WriteLine (local.EqualsExact (utc)); // False
```

Класс TimeZoneInfo

Класс TimeZoneInfo предоставляет информацию, касающуюся названий часовых поясов, смещений UTC и правил перехода на летнее время.

Класс TimeZone

Статический метод TimeZone.CurrentCulture возвращает объект TimeZone:

```
TimeZone zone = TimeZone.CurrentCulture;
Console.WriteLine (zone.StandardName); // Pacific Standard Time
                                         // (стандартное тихоокеанское время)
Console.WriteLine (zone.DaylightName); // Pacific Daylight Time (летнее
                                         // тихоокеанское время)
```

Метод GetDaylightChanges возвращает специфичную информацию о летнем времени для заданного года:

```
DaylightTime day = zone.GetDaylightChanges (2019);
Console.WriteLine (day.Start.ToString ("M")); // 10 March
Console.WriteLine (day.End.ToString ("M")); // 03 November
Console.WriteLine (day.Delta); // 01:00:00
```

Класс TimeZoneInfo

Статический метод TimeZoneInfo.Local возвращает объект TimeZoneInfo, основанный на текущих местных настройках. Ниже показаны результаты, которые получены для Калифорнии:

```
TimeZoneInfo zone = TimeZoneInfo.Local;
Console.WriteLine (zone.StandardName); // Pacific Standard Time
                                         // (стандартное тихоокеанское время)
Console.WriteLine (zone.DaylightName); // Pacific Daylight Time
                                         // (летнее тихоокеанское время)
```

Методы IsDaylightSavingTime и GetUtcOffset работают следующим образом:

```
DateTime dt1 = new DateTime (2019, 1, 1); // DateTimeOffset тоже работает
DateTime dt2 = new DateTime (2019, 6, 1);
Console.WriteLine (zone.IsDaylightSavingTime (dt1)); // True
Console.WriteLine (zone.IsDaylightSavingTime (dt2)); // False
Console.WriteLine (zone.GetUtcOffset (dt1)); // -08:00:00
Console.WriteLine (zone.GetUtcOffset (dt2)); // -07:00:00
```

Вызвав метод FindSystemTimeZoneById с идентификатором пояса, можно получить объект TimeZoneInfo для любого часового пояса в мире. Мы переключимся на Западную Австралию по причинам, которые вскоре станут ясны:

```
TimeZoneInfo wa = TimeZoneInfo.FindSystemTimeZoneById
    ("W. Australia Standard Time");
Console.WriteLine (wa.Id);           // W. Australia Standard Time
                                    // (стандартное время Западной Австралии)
Console.WriteLine (wa.DisplayName); // (GMT+08:00) Perth ((GMT+08:00) Перт)
Console.WriteLine (wa.BaseUtcOffset); // 08:00:00
Console.WriteLine (wa.SupportsDaylightSavingTime); // True
```

Свойство `Id` соответствует значению, которое передано методу `FindSystemTimeZoneById`. Статический метод `GetSystemTimeZones` возвращает все часовые пояса мира; следовательно, можно вывести список всех допустимых идентификаторов поясов:

```
foreach (TimeZoneInfo z in TimeZoneInfo.GetSystemTimeZones ())
    Console.WriteLine (z.Id);
```



Можно также создать специальный часовой пояс, вызвав метод `TimeZoneInfo.CreateCustomTimeZone`. Поскольку класс `TimeZoneInfo` неизменяемый, данному методу должны передаваться все существенные данные в качестве аргументов.

С помощью вызова метода `ToSerializedString` можно сериализовать предопределенный или специальный часовой пояс в (почти) читабельную для человека строку, а посредством вызова метода `TimeZoneInfo.FromSerializedString` десериализовать ее.

Статический метод `ConvertTime` преобразует экземпляр `DateTime` или `DateTimeOffset` из одного часового пояса в другой. Можно включить либо только целевой объект `TimeZoneInfo`, либо исходный и целевой объекты `TimeZoneInfo`. С помощью методов `ConvertTimeFromUtc` и `ConvertTimeToUtc` можно также выполнять преобразование прямо из времени UTC или в него.

Для работы с летним временем `TimeZoneInfo` предоставляет перечисленные ниже дополнительные методы.

- `IsInvalidTime` возвращает `true`, если значение `DateTime` находится в пределах часа (или дельты), который будет пропущен, когда часы переводятся вперед.
- `IsAmbiguousTime` возвращает `true`, если `DateTime` или `DateTimeOffset` находятся в пределах часа (или дельты), который будет повторен, когда часы переводятся назад.
- `GetAmbiguousTimeOffsets` возвращает массив из элементов `TimeSpan`, представляющий допустимые варианты смещения для неоднозначного `DateTime` или `DateTimeOffset`.

Вы не можете получить из объекта `TimeZoneInfo` простые даты, отражающие начало и конец летнего времени. Взамен понадобится вызвать метод `GetAdjustmentRules`, который возвращает декларативный список правил перехода на летнее время, применяемых ко всем годам. Каждое правило имеет свойства `DateStart` и `DateEnd`, указывающие диапазон дат, в рамках которого это правило допустимо:

```
foreach (TimeZoneInfo.AdjustmentRule rule in wa.GetAdjustmentRules())
    Console.WriteLine ("Rule: applies from " + rule.DateStart + " to " + rule.DateEnd);
```

В Западной Австралии переход на летнее время впервые был введен в межсезонье 2006 года (и затем в 2009 году отменен). Это требует специального правила для первого года; таким образом, существуют два правила:

```
Rule: applies from 1/01/2006 12:00:00 AM to 31/12/2006 12:00:00 AM
Rule: applies from 1/01/2007 12:00:00 AM to 31/12/2009 12:00:00 AM
```

Правило: применяется с 1/01/2006 12:00:00 AM по 31/12/2006 12:00:00 AM

Правило: применяется с 1/01/2007 12:00:00 AM по 31/12/2009 12:00:00 AM

Каждый экземпляр AdjustmentRule имеет свойство DaylightDelta типа TimeSpan (почти в каждом случае оно составляет один час), а также свойства DaylightTransitionStart и DaylightTransitionEnd. Последние два свойства относятся к типу TimeZoneInfo.TransitionTime, в котором определены следующие свойства:

```
public bool IsFixedDateRule { get; }
public DayOfWeek DayOfWeek { get; }
public int Week { get; }
public int Day { get; }
public int Month { get; }
public DateTime TimeOfDay { get; }
```

Время перехода несколько усложняется тем, что оно должно представлять и фиксированные, и плавающие даты. Примером плавающей даты может служить “последнее воскресенье марта месяца”. Ниже описаны правила интерпретации времени перехода.

Если для конечного перехода свойство IsFixedDateRule равно true, свойство Day — 1, свойство Month — 1 и свойство TimeOfDay — DateTime.MinValue, то в таком году летнее время не заканчивается (это может произойти только в южном полушарии после первоначального ввода перехода на летнее время в регионе).

В противном случае, если IsFixedDateRule равно true, тогда свойства Month, Day и TimeOfDay определяют начало или конец правила корректировки.

В противном случае, если IsFixedDateRule равно false, то свойства Month, DayOfWeek, Week и TimeOfDay определяют начало или конец правила корректировки. В последнем случае свойство Week ссылается на неделю месяца, причем 5 означает последнюю неделю. Мы можем проиллюстрировать сказанное путем перечисления правил корректировки для часового пояса wa:

```
foreach (TimeZoneInfo.AdjustmentRule rule in wa.GetAdjustmentRules())
{
    Console.WriteLine ("Rule: applies from " + rule.DateStart +
                      " to " + rule.DateEnd);           // применяется с ... по ...
    Console.WriteLine ("  Delta: " + rule.DaylightDelta);        // дельта
    Console.WriteLine ("  Start: " + FormatTransitionTime
                      (rule.DaylightTransitionStart, false)); // начало
    Console.WriteLine ("  End:   " + FormatTransitionTime
                      (rule.DaylightTransitionEnd, true));   // конец
    Console.WriteLine();
}
```

В методе FormatTransitionTime мы соблюдаем только что описанные правила:

```
static string FormatTransitionTime (TimeZoneInfo.TransitionTime tt,
                                    bool endTime)
{
    if (endTime && tt.IsFixedDateRule
        && tt.Day == 1 && tt.Month == 1
        && tt.TimeOfDay == DateTime.MinValue)
        return "-";

    string s;
    if (tt.IsFixedDateRule)
        s = tt.Day.ToString();
    else
        s = "The " +
            "first second third fourth last".Split() [tt.Week - 1] +
            " " + tt.DayOfWeek + " in";

    return s + " " + DateTimeFormatInfo.CurrentInfo.MonthNames [tt.Month-1]
           + " at " + tt.TimeOfDay.TimeOfDay;
}
```

Летнее время и структура DateTime

Если вы используете структуру DateTimeOffset или UTC-вариант DateTime, то сравнения эквивалентности свободны от влияния летнего времени. Однако с местными вариантами DateTime переход на летнее время может вызвать проблемы.

Подытожить правила можно следующим образом.

- Переход на летнее время оказывает воздействие на местное время, но не на время UTC.
- Когда часы переводят назад, тогда сравнения, основанные на том, что время движется вперед, дадут сбой, если (и только если) они применяют местные значения DateTime.
- Всегда можно надежно перемещаться между UTC и местным временем (на том же самом компьютере), даже когда часы переводят назад.

Метод IsDaylightSavingTime сообщает о том, относится ли заданное местное значение DateTime к летнему времени. В случае времени UTC всегда возвращается false:

```
Console.WriteLine(DateTime.Now.IsDaylightSavingTime());      // True или False
Console.WriteLine(DateTime.UtcNow.IsDaylightSavingTime());    // Всегда False
```

Предполагая, что dto имеет тип DateTimeOffset, следующее выражение делает то же самое:

```
dto.LocalDateTime.IsDaylightSavingTime
```

Конец летнего времени представляет особую сложность для алгоритмов, использующих местное время, потому что когда часы переводят назад, один и тот же час (точнее Delta) повторяется.



Любые два экземпляра `DateTime` можно надежно сравнивать, предварительно вызывая на каждом метод `ToUniversalTime`. Такая стратегия отказывает, если (и только если) в точности один из них имеет значение `DateTimeKind`, равное `Unspecified`. Возможность отказа является еще одной причиной отдавать предпочтение типу `DateTimeOffset`.

Форматирование и разбор

Форматирование означает преобразование в строку, а разбор — преобразование из строки. Потребность в форматировании и разборе во время программирования возникает часто, причем в самых разнообразных ситуациях. Для этого в .NET предусмотрено несколько механизмов.

- **ToString и Parse.** Данные методы предоставляют стандартную функциональность для многих типов.
- **Поставщики форматов.** Они проявляются в виде дополнительных методов `ToString` (и `Parse`), которые принимают форматную строку и/или поставщик формата. Поставщики форматов характеризуются высокой гибкостью и чувствительностью к культуре. В состав .NET входят поставщики форматов для числовых типов и типов `DateTime/DateTimeOffset`.
- **XmlConvert.** Это статический класс с методами, которые поддерживают форматирование и разбор с соблюдением стандартов XML. Класс `XmlConvert` также удобен при универсальном преобразовании, когда требуется обеспечить независимость от культуры либо избежать некорректного разбора. Класс `XmlConvert` поддерживает числовые типы, а также типы `bool`, `DateTime`, `DateTimeOffset`, `TimeSpan` и `Guid`.
- **Преобразователи типов.** Они предназначены для визуальных контроллеров и средств разбора XAML.

В текущем разделе мы обсудим первые два механизма, уделяя особое внимание поставщикам форматов. В следующем разделе мы опишем `XmlConvert` и преобразователи типов, а также другие механизмы преобразования.

Методы `ToString` и `Parse`

Метод `ToString` является простейшим механизмом форматирования. Он обеспечивает осмысленный вывод для всех простых типов значений (`bool`, `DateTime`, `DateTimeOffset`, `TimeSpan`, `Guid` и всех числовых типов). Для обратной операции в каждом из указанных типов определен статический метод `Parse`:

```
string s = true.ToString();    // s = "True"  
bool b = bool.Parse (s);      // b = true
```

Если разбор терпит неудачу, тогда генерируется исключение `FormatException`. Во многих типах также определен метод `TryParse`, который в случае отказа преобразования вместо генерации исключения возвращает `false`:

```
bool failure = int.TryParse ("qwerty", out int i1);
bool success = int.TryParse ("123", out int i2);
```

Если выходное значение вас не интересует и требуется лишь проверить, будет ли разбор успешным, то можете применить отбрасывание:

```
bool success = int.TryParse ("123", out int _);
```

Если вы предвидите ошибку, тогда вызов TryParse будет более быстрым и элегантным решением, чем вызов Parse в блоке обработки исключения.

Методы Parse и TryParse в DateTime (DateTimeOffset) и числовых типах учитывают местные настройки культуры, что можно изменить, указывая объект CultureInfo. Часто указание инвариантной культуры является удачной идеей. Например, разбор "1.234" в double дает 1234 для Германии:

```
Console.WriteLine (double.Parse ("1.234")); // 1234 (в Германии)
```

Причина в том, что символ точки в Германии используется в качестве разделителя тысяч, а не как десятичная точка. Проблему устраняет указание *инвариантной культуры*:

```
double x = double.Parse ("1.234", CultureInfo.InvariantCulture);
```

То же самое применимо и в отношении вызова ToString:

```
string x = 1.234.ToString (CultureInfo.InvariantCulture);
```



Начиная с версии .NET 8, числовые типы и типы даты/времени .NET (а также другие простые типы) позволяют напрямую форматировать и анализировать UTF-8 с помощью новых методов TryFormat и Parse/TryParse, которые работают с массивом байтов или Span<byte> (см. главу 23). В сценариях с высокой производительностью это может быть более эффективно, чем работа с обычными строками (UTF-16) и выполнение отдельного кодирования/декодирования UTF-8.

Поставщики форматов

Зачастую требуется больший контроль над тем, как происходит форматирование и разбор. Например, существуют десятки способов форматирования DateTime (DateTimeOffset). Поставщики форматов позволяют получить обширный контроль над форматированием и разбором и поддерживаются для числовых типов и типов даты/времени. Поставщики форматов также используются элементами управления пользовательского интерфейса для выполнения форматирования и разбора.

Интерфейсом для применения поставщика формата является IFormattable, который реализуют все числовые типы и тип DateTime (DateTimeOffset):

```
public interface IFormattable
{
    string ToString (string format, IFormatProvider formatProvider);
}
```

Методу `ToString` в первом аргументе передается *форматная строка*, а во втором — *поставщик формата*. Форматная строка предоставляет инструкции; поставщик формата определяет то, как инструкции транслируются. Например:

```
NumberFormatInfo f = new NumberFormatInfo();
f.CurrencySymbol = " $$";
Console.WriteLine (3.ToString ("C", f));           // $$ 3.00
```

Здесь "C" представляет собой форматную строку, которая указывает *денежное значение*, а объект `NumberFormatInfo` является поставщиком формата, определяющим то, каким образом должно визуализироваться денежное значение (и другие числовые представления). Такой механизм допускает глобализацию.



Все форматные строки для чисел и дат описаны в разделе “Стандартные форматные строки и флаги разбора” далее в главе.

Если для форматной строки или поставщика указать `null`, то будет применен стандартный вариант. Стандартный поставщик формата, `CultureInfo.CurrentCulture`, отражает настройки панели управления компьютера во время выполнения (если он не был переустановлен). Например:

```
Console.WriteLine (10.3.ToString ("C", null)); // $10.30
```

Ради удобства в большинстве типов метод `ToString` перегружен, так что `null` для поставщика можно не указывать:

```
Console.WriteLine (10.3.ToString ("C"));           // $10.30
Console.WriteLine (10.3.ToString ("F4")); //10.3000 (четыре десятичных позиции)
```

Вызов метода `ToString` без аргументов для типа `DateTime` (`DateTimeOffset`) или числового типа эквивалентен использованию стандартного поставщика формата с пустой форматной строкой.

В .NET определены три поставщика формата (все они реализуют интерфейс `IFormatProvider`):

```
NumberFormatInfo
DateTimeFormatInfo
CultureInfo
```



Все типы перечислений также поддерживают форматирование, хотя специальный класс, реализующий интерфейс `IFormatProvider`, для них не предусмотрен.

Поставщики форматов и `CultureInfo`

В рамках контекста поставщиков форматов тип `CultureInfo` действует как механизм косвенности для двух других поставщиков форматов, возвращая объект `NumberFormatInfo` или `DateTimeFormatInfo`, который может быть применен к региональным настройкам культуры.

В следующем примере мы запрашиваем специфическую культуру (английский (*english*) в Великобритании (*Great Britain*)):

```
CultureInfo uk = CultureInfo.GetCultureInfo ("en-GB");
Console.WriteLine (3.ToString ("C", uk));           // £3.00
```

Показанный код выполняется с использованием стандартного объекта `NumberFormatInfo`, применимого к культуре en-GB.

В приведенном ниже примере производится форматирование значения `DateTime` с использованием инвариантной культуры. Инвариантная культура всегда остается одной и той же независимо от настроек компьютера:

```
DateTime dt = new DateTime (2000, 1, 2);
CultureInfo iv = CultureInfo.InvariantCulture;
Console.WriteLine (dt.ToString (iv));                                // 01/02/2000 00:00:00
Console.WriteLine (dt.ToString ("d", iv));                            // 01/02/2000
```



Инвариантная культура основана на американской культуре с перечисленными ниже различиями:

- символом валюты является ₩, а не \$;
- дата и время форматируются с ведущими нулями (хотя месяц по-прежнему идет первым);
- для времени применяется 24-часовой формат, а не 12-часовой с указателем AM/PM.

Использование `NumberFormatInfo` или `DateTimeFormatInfo`

В следующем примере мы создаем экземпляр `NumberFormatInfo` и меняем разделитель групп цифр с запятой на пробел. После этого мы используем его для форматирования числа с тремя десятичными позициями:

```
NumberFormatInfo f = new NumberFormatInfo ();
f.NumberGroupSeparator = " ";
Console.WriteLine (12345.6789.ToString ("N3", f));    // 12 345.679
```

Начальные настройки для `NumberFormatInfo` или `DateTimeFormatInfo` основаны на инвариантной культуре. Тем не менее, иногда более удобно выбирать другую стартовую точку. Для этого с помощью `Clone` можно клонировать существующий поставщик формата:

```
NumberFormatInfo f = (NumberFormatInfo)
    CultureInfo.CurrentCulture.NumberFormat.Clone();
```

Клонированный поставщик формата всегда является записываемым, даже если исходный поставщик допускал только чтение.

Смешанное форматирование

Смешанные форматные строки позволяют комбинировать подстановку переменных с форматными строками. Статический метод `String.Format` принимает смешанную форматную строку (как иллюстрировалось в разделе “Метод `string.Format` и смешанные форматные строки” ранее в главе):

```
string composite = "Credit={0:C}";
Console.WriteLine (string.Format (composite, 500)); // Credit=$500.00
```

Сам класс `Console` перегружает свои методы `Write` и `WriteLine` для приема смешанных форматных строк, позволяя слегка сократить код примера:

```
Console.WriteLine ("Credit={0:C}", 500); // Credit=$500.00
```

Смешанную форматную строку можно также добавлять к `StringBuilder` (через `AppendFormat`) и к `TextWriter` для ввода-вывода (см. главу 15).

Метод `string.Format` принимает необязательный поставщик формата. Простым сценарием его применения является вызов `ToString` на произвольном объекте с передачей в то же время поставщика формата:

```
string s = string.Format (CultureInfo.InvariantCulture, "{0}", someObject);
```

Вот чему эквивалентен такой код:

```
string s;
if (someObject is IFormattable)
    s = ((IFormattable)someObject).ToString (null,
                                              CultureInfo.InvariantCulture);
else if (someObject == null)
    s = "";
else
    s = someObject.ToString();
```

Разбор с использованием поставщиков форматов

Стандартного интерфейса для выполнения разбора посредством поставщика формата не предусмотрено. Взамен каждый участвующий тип имеет перегруженную версию своего статического метода `Parse` (и `TryParse`), которая принимает поставщик формата и дополнительное значение перечисления `NumberStyles` или `DateTimeStyles`.

Перечисления `NumberStyles` и `DateTimeStyles` управляют работой разбора: они позволяют указывать аспекты наподобие того, могут ли встречаться во входной строке круглые скобки или символ валюты. (По умолчанию ни то, ни другое не разрешено.) Например:

```
int error = int.Parse ("(2)"); // Генерируется исключение
int minusTwo = int.Parse ("(2)", NumberStyles.Integer |
                           NumberStyles.AllowParentheses); // Нормально
decimal fivePointTwo = decimal.Parse ("£5.20", NumberStyles.Currency,
                                       CultureInfo.GetCultureInfo ("en-GB"));
```

В следующем разделе описаны все члены перечислений `NumberStyles` и `DateTimeStyles`, а также стандартные правила разбора для каждого типа.

IFormatProvider и ICustomeFormatter

Все поставщики форматов реализуют интерфейс `IFormatProvider`:

```
public interface IFormatProvider { object GetFormat (Type formatType); }
```

Цель заключается в обеспечении косвенности — именно это позволяет `CultureInfo` поручить выполнение работы соответствующему объекту `NumberFormatInfo` или `DateTimeFormatInfo`.

За счет реализации интерфейса `IFormatProvider` — наряду с `ICustomFormatter` — можно также построить собственный поставщик формата в сочетании с существующими типами.

В интерфейсе ICustomFormatter определен единственный метод:

```
string Format (string format, object arg, IFormatProvider formatProvider);
```

Следующий специальный поставщик формата записывает числа с помощью слов на английском языке:

```
public class WordyFormatProvider : IFormatProvider, ICustomFormatter
{
    static readonly string[] _numberWords =
        "zero one two three four five six seven eight nine minus point".Split();
    IFormatProvider _parent; // Позволяет потребителям строить
                            // цепочки поставщиков форматов

    public WordyFormatProvider () : this (CultureInfo.CurrentCulture) { }
    public WordyFormatProvider (IFormatProvider parent) => _parent = parent;
    public object GetFormat (Type formatType)
    {
        if (formatType == typeof (ICustomFormatter)) return this;
        return null;
    }
    public string Format (string format, object arg, IFormatProvider prov)
    {
        // Если это не наша форматная строка, тогда передать
        // ее родительскому поставщику:
        if (arg == null || format != "W")
            return string.Format (_parent, "{0:" + format + "}", arg);
        StringBuilder result = new StringBuilder();
        string digitList = string.Format (CultureInfo.InvariantCulture,
                                         "{0}", arg);
        foreach (char digit in digitList)
        {
            int i = "0123456789-".IndexOf (digit,
                                         StringComparison.InvariantCulture);
            if (i == -1) continue;
            if (result.Length > 0) result.Append (' ');
            result.Append (_numberWords[i]);
        }
        return result.ToString();
    }
}
```

Обратите внимание, что в методе Format для преобразования входного числа в строку мы применяем string.Format с указанием InvariantCulture. Было бы намного проще вызвать ToString на arg, но тогда использовалось бы свойство CurrentCulture. Причина потребности в инвариантной культуре становится очевидной дальше в коде:

```
int i = "0123456789-".IndexOf (digit, StringComparison.InvariantCulture);
```

Здесь критически важно, чтобы строка с числом содержала только символы 0123456789-. и никаких интернационализированных версий для них.

Ниже приведен пример, в котором задействован WordyFormatProvider:

```
double n = -123.45;
IFormatProvider fp = new WordyFormatProvider();
Console.WriteLine (string.Format (fp, "{0:C} in words is {0:W}", n));
// Выводит -$123.45 in words is minus one two three point four five
```

Специальные поставщики форматов могут применяться только в смешанных форматных строках.

Стандартные форматные строки и флаги разбора

Стандартные форматные строки управляют способом преобразования в строку числового типа или типа DateTime/DateTimeOffset. Существуют два вида форматных строк.

- **Стандартные форматные строки.** С их помощью обеспечивается общее управление. Стандартная форматная строка состоит из одиночной буквы и следующей за ней дополнительной цифры (смысл которой зависит от буквы). Примером может служить "C" или "F2".
- **Специальные форматные строки.** С их помощью контролируется каждый символ посредством шаблона. Примером может служить "0:#.000E+00".

Специальные форматные строки не имеют никакого отношения к специальным поставщикам форматов.

Форматные строки для чисел

В табл. 6.2 приведен список всех стандартных форматных строк для чисел.

Предоставление форматной строки, не предназначенной для чисел (либо null или пустой строки), эквивалентно применению стандартной форматной строки "G" без цифры. В таком случае поведение будет следующим.

- Числа меньше 10⁻⁴ или больше, чем точность типа, выражаются с использованием экспоненциальной (научной) записи.
- Две десятичные позиции на пределе точности float или double округляются, чтобы замаскировать неточности, присущие преобразованию в десятичный тип из лежащей в основе двоичной формы.



Только что описанное автоматическое округление обычно полезно и проходит незаметно. Тем не менее, оно может вызвать проблему, если необходимо вернуться обратно к числу; другими словами, преобразование числа в строку и обратно (возможно, многократно повторяемое) может нарушить равенство значений. По этой причине существуют форматные строки "R", "G17" и "G9", подавляющие такое неявное округление.

В табл. 6.3 представлен список специальных форматных строк для чисел.

Таблица 6.2. Стандартные форматные строки для чисел

Буква	Что означает	Пример ввода	Результат	Примечания
G или g	"Общий" формат	1.2345, "G" 0.00001, "G" 0.00001, "g" 1.2345, "G3" 12345, "G3"	1.2345 1E-05 1e-05 1.23 1.23E04	Переключается на экспоненциальную запись для очень малых или больших чисел. G3 ограничивает точность всего тремя цифрами (перед и после точки)
F	Формат с фиксированной точкой	2345.678, "F2" 2345.6, "F2"	2345.68 2345.60	F2 округляет до двух десятичных позиций
N	Формат с фиксированной точкой и разделителем групп ("числовой")	2345.678, "N2" 2345.6, "N2"	2,345.68 2,345.60	То же, что и выше, но с разделителем групп (тысяч); детали берутся из поставщика формата
D	Заполнение ведущими нулями	123, "D5" 123, "D1"	00123 123	Только для целочисленных типов. D5 дополняет слева до пяти цифр; усечение не производится
E или e	Принудительное применение экспоненциальной записи	56789, "E" 56789, "e" 56789, "E2"	5.678900E+004 5.678900e+004 5.68E+004	По умолчанию точность составляет шесть цифр
C	Денежное значение	1.2, "C" 1.2, "C4"	\$1.20 \$1.2000	С без цифры использует стандартное количество десятичных позиций, заданное поставщиком формата
P	Процент	.503, "P".503, "P0"	50.30 % 50 %	Использует символ и компоновку из поставщика формата.
X или x	Шестнадцатеричный формат	47, "X" 47, "x" 47, "X4"	2F 2f 002F	Десятичные позиции могут быть отброшены Х — для представления шестнадцатеричных цифр в верхнем регистре; x — для представления шестнадцатеричных цифр в нижнем регистре. Только для целочисленных типов
R или G9/G17	Округление	1f / 3f, "R"	0.333333343	R используется для типа BigInteger, G17 — для double или G9 — для float

Таблица 6.3. Специальные форматные строки для чисел

Специф- икатор	Что означает	Пример ввода	Результат	Примечания
#	Заполнитель для цифры	12.345, ".##" 12.345, ".####"	12.35 12.345	Ограничивает количество цифр после десятичной точки
0	Заполнитель для нуля	12.345, ".00" 12.345, ".0000" 99, "000.00"	12.35 12.3450 099.00	Как и выше, ограничивает количество цифр после десятичной точки, но также дополняет нулями до и после десятичных позиций
.	Десятичная точка			Отображает десятичную точку. Фактический символ берется из NumberFormatInfo
,	Разделитель групп	1234, "#,###,###" 1234, "0,000,000"	1,234 0,001,234	Символ берется из NumberFormatInfo
, (как и выше)	Коэффициент	1000000, "#," 1000000, "#,,"	1000 1	Когда запятая находится до или после десятичной позиции, она действует как коэффициент, разделяя результат на 1000, 1 000 000 и т.д.
%	Процентная запись	0.6, "00%"	60%	Сначала умножает на 100, а затем подставляет символ процента, полученный из NumberFormatInfo
E0, e0, E+0, e+0, E-0, e-0	Экспоненциальная запись	1234, "0E0" 1234, "0E+0" 1234, "0.00E00" 1234, "0.00e00"	1E3 1E+3 1.23E03 1.23e03	
\	Признак ли- терального сим- вола	50, @"\#0"	#50	Используется в сочета- нии с префиксом @ в строках — или же можно применять \\
'xx'	Признак ли- теральной строки	50, "0 '...'"	50 ...	
;	Разделитель секций	15, "#; (#); zero" -5, "#; (#); zero" 0, "#; (#); zero"	15 (5) zero	(Если положительное) (Если отрицательное) (Если ноль)
Любой другой символ	Литерал	35.2, "\$0 . 00c"	\$35 . 20c	

Перечисление NumberStyles

В каждом числовом типе определен статический метод Parse, принимающий аргумент типа NumberStyles. Перечисление флагов NumberStyles позволяет определить, каким образом строка читается при преобразовании в числовой тип. Перечисление NumberStyles имеет следующие комбинируемые члены:

AllowLeadingWhite, AllowTrailingWhite, AllowLeadingSign, AllowTrailingSign, AllowParentheses, AllowDecimalPoint, AllowThousands, AllowExponent, AllowCurrencySymbol, AllowHexSpecifier

В NumberStyles также определены составные члены:

None, Integer, Float, Number, HexNumber, Currency, Any

Все составные члены кроме None включают AllowLeadingWhite и AllowTrailingWhite. Остальные члены проиллюстрированы на рис. 6.1; три наиболее полезных выделены полужирным.

	AllowLeadingSign	AllowTrailingSign	AllowParenthesis	AllowDecimalPoint	AllowThousands	AllowExponent	AllowCurrencySymbol	AllowHexSpecifier
Integer	✓							
Float	✓		✓	✓				
Number	✓	✓	✓	✓				
HexNumber							✓	
Currency	✓	✓	✓	✓	✓	✓		
Any	✓	✓	✓	✓	✓	✓	✓	

Рис. 6.1. Составные члены NumberStyles

Когда метод Parse вызывается без указания флагов, применяются правила по умолчанию, как показано на рис. 6.2.

Целочисленные типы	Стандартные флаги разбора								
	AllowLeadingSign	AllowTrailingSign	AllowParenthesis	AllowDecimalPoint	AllowThousands	AllowExponent	AllowCurrencySymbol	AllowHexSpecifier	
Integer	✓								
double и float	✓		✓	✓	✓				
decimal	✓	✓	✓	✓					

Рис. 6.2. Стандартные флаги разбора для числовых типов

Если правила по умолчанию, представленные на рис. 6.2, не подходят, тогда значения NumberStyles должны указываться явно:

```
int thousand = int.Parse ("3E8", NumberStyles.HexNumber);
int minusTwo = int.Parse ("(2)", NumberStyles.Integer | 
                           NumberStyles.AllowParentheses);
double aMillion = double.Parse ("1,000,000", NumberStyles.Any);
decimal threeMillion = decimal.Parse ("3e6", NumberStyles.Any);
decimal fivePointTwo = decimal.Parse ("$5.20", NumberStyles.Currency);
```

Из-за того, что поставщик формата не указан, приведенный выше код работает с местным символом валюты, разделителем групп, десятичной точкой и т.д. В следующем примере для денежных значений жестко закодирован знак евро и разделитель групп в виде пробела:

```
NumberFormatInfo ni = new NumberFormatInfo();
ni.CurrencySymbol = "€";
ni.CurrencyGroupSeparator = " ";
double million = double.Parse ("€1 000 000", NumberStyles.Currency, ni);
```

Форматные строки для даты/времени

Форматные строки для DateTime/DateTimeOffset могут быть разделены на две группы, основываясь на том, учитывают ли они настройки культуры и поставщика формата. Форматные строки для даты/времени, чувствительные к культуре, описаны в табл. 6.4, а нечувствительные — в табл. 6.5. Пример вывода получен в результате форматирования следующего экземпляра DateTime (с инвариантной культурой в случае табл. 6.4):

```
new DateTime (2000, 1, 2, 17, 18, 19);
```

Таблица 6.4. Форматные строки для даты/времени, чувствительные к культуре

Форматная строка	Что означает	Пример вывода
d	Краткая дата	01/02/2000
D	Полная дата	Sunday, 02 January 2000
t	Краткое время	17:18
T	Полное время	17:18:19
f	Полная дата + краткое время	Sunday, 02 January 2000 17:18
F	Полная дата + полное время	Sunday, 02 January 2000 17:18:19
g	Краткая дата + краткое время	01/02/2000 17:18
G (по умолчанию)	Краткая дата + полное время	01/02/2000 17:18:19
m, M	Месяц и день	02 January
y, Y	Год и месяц	January 2000

Таблица 6.5. Форматные строки для даты/времени, нечувствительные к культуре

Форматная строка	Что означает	Пример вывода	Примечания
o	Возможность кругового преобразования	2000-01-02T 17:18:19.0000000	Будет присоединять информацию о часовом поясе, если только DateTimeKind не является Unspecified
r, R	Стандарт RFC 1123	Sun, 02 Jan 2000 17:18:19 GMT	Потребуется явно преобразовать в UTC с помощью DateTime.ToUniversalTime
s	Сортируемое; ISO 8601	2000-01-02T17:18:19	Совместимо с текстовой сортировкой
u	"Универсальное" сортируемое	2000-01-02 17:18:19Z	Подобно предыдущему; потребуется явно преобразовать в UTC
U	UTC	Sunday, 02 January 2000 17:18:19	Краткая дата плюс краткое время, преобразованное в UTC

Форматные строки "r", "R" и "u" выдают суффикс, который подразумевает UTC; пока что они не осуществляют автоматическое преобразование местной версии DateTime в UTC-версию (поэтому преобразование придется делать самостоятельно). По иронии судьбы "U" автоматически преобразует в UTC, но не записывает суффикс часового пояса! На самом деле "o" является единственным спецификатором формата в группе, который обеспечивает запись недвусмысленного экземпляра DateTime безо всякого вмешательства.

Класс `DateTimeFormatInfo` также поддерживает специальные форматные строки: они аналогичны специальным форматным строкам для чисел. Их полный список можно найти в документации от Microsoft. Ниже приведен пример специальной форматной строки:

```
уууу-ММ-дд HH:mm:ss
```

Разбор и некорректный разбор значений `DateTime`

Строки, в которых месяц или день помещен первым, являются неоднозначными и очень легко могут привести к некорректному разбору, в частности при наличии глобальных пользователей. Это не будет проблемой в элементах управления пользовательского интерфейса, т.к. при разборе и форматировании принудительно применяются одни и те же настройки. Но при записи в файл, например, некорректный разбор дня/месяца может стать реальной проблемой. Существуют два решения:

- при формировании и разборе всегда придерживаться одной и той же явной культуры (например, инвариантной);
- форматировать `DateTime` и `DateTimeOffset` в независимой от культуры манере.

Второй подход более надежен — особенно если выбран формат, в котором первым указывается год из четырех цифр: такие строки намного реже некорректно разбираются другой стороной. Кроме того, строки, сформированные с использованием *соответствующего стандартам* формата с годом в самом начале (такого как "о"), могут корректно разбираться вместе с локально сформированными строками. (Даты, сформированные посредством "s" или "u", обеспечивают дополнительное преимущество, будучи сортируемыми.)

В целях иллюстрации предположим, что сгенерирована следующая нечувствительная к культуре строка DateTime по имени s:

```
string s = DateTime.Now.ToString ("o");
```



Форматная строка "о" включает в вывод миллисекунды. Приведенная ниже специальная форматная строка дает тот же результат, что и "о", но без миллисекунд:

```
yyyy-MM-ddTHH:mm:ss K
```

Повторно разобрать строку s можно двумя способами. Метод ParseExact требует строгого соответствия с указанной форматной строкой:

```
DateTime dt1 = DateTime.ParseExact (s, "о", null);
```

(Достичь похожего результата можно с помощью методов ToString и ToDateTime класса XmlConvert.)

Тем не менее, метод Parse неявно принимает как формат "о", так и формат CurrentCulture:

```
DateTime dt2 = DateTime.Parse (s);
```

Прием работает и для DateTime, и для DateTimeOffset.



Метод ParseExact обычно предпочтительнее, когда вам известен формат разбираемой строки. Это означает, что если строка сформирована некорректно, тогда сгенерируется исключение — что обычно лучше, нежели риск получения неправильно разобранной даты.

Перечисление DateTimeStyles

Перечисление флагов DateTimeStyles предоставляет дополнительные инструкции при вызове метода Parse на DateTime (DateTimeOffset). Ниже приведены его члены:

```
None,  
AllowLeadingWhite, AllowTrailingWhite, AllowInnerWhite,  
AssumeLocal, AssumeUniversal, AdjustToUniversal,  
NoCurrentDateDefault, RoundTripKind
```

Имеется также составной член AllowWhiteSpaces:

```
AllowWhiteSpaces = AllowLeadingWhite | AllowTrailingWhite | AllowInnerWhite
```

Стандартным значением является `None`. Таким образом, лишние пробельные символы обычно запрещены (что не касается пробельных символов, являющихся частью стандартного шаблона `DateTime`).

Флаги `AssumeLocal` и `AssumeUniversal` применяются, если строка не имеет суффикса часового пояса (вроде `Z` или `+9:00`). Флаг `AdjustToUniversal` учитывает суффиксы часовых поясов, но затем выполняет преобразование в UTC с использованием текущих региональных настроек.

При разборе строки, содержащей время и не включающей дату, по умолчанию берется сегодняшняя дата. Однако если применен флаг `NoCurrentDateDefault`, то будет использоваться 1 января 0001 года.

Форматные строки для перечислений

В разделе “Перечисления” главы 3 было описано форматирование и разбор перечислимых значений. В табл. 6.6 приведен список форматных строк для перечислений и результаты их применения в следующем операторе:

```
Console.WriteLine (System.ConsoleColor.Red.ToString (formatString));
```

Таблица 6.6. Форматные строки для перечислений

Форматная строка	Что означает	Пример вывода	Примечания
G или g	“Общий” формат	Red	Используется по умолчанию
F или f	Трактуется, как если бы присутствовал атрибут <code>Flags</code>	Red	Работает на составных членах, даже если перечисление не имеет атрибута <code>Flags</code>
D или d	Десятичное значение	12	Извлекает лежащее в основе целочисленное значение
X или x	Шестнадцатеричное значение	0000000C	Извлекает лежащее в основе целочисленное значение

Другие механизмы преобразования

В предшествующих двух разделах рассматривались поставщики форматов — основной механизм .NET для форматирования и разбора. Прочие важные механизмы преобразования разбросаны по различным типам и пространствам имен. Есть механизмы, преобразующие в тип `string` и из него, а есть такие, которые осуществляют другие виды преобразований. В настоящем разделе мы обсудим следующие темы.

- Класс `Convert` и его функции:
 - преобразования вещественных чисел в целые, которые производят округление, а не усечение;
 - разбор чисел в системах счисления с основаниями 2, 8 и 16;
 - динамические преобразования;
 - преобразования Base-64.

- Класс `XmlConvert` и его роль в форматировании и разборе для XML.
- Преобразователи типов и их роль в форматировании и разборе для визуальных конструкторов и XAML.
- Класс `BitConverter`, предназначенный для двоичных преобразований.

Класс `Convert`

Перечисленные типы .NET называются *базовыми типами*:

- `bool`, `char`, `string`, `System.DateTime` и `System DateTimeOffset`;
- все *числовые типы C#*.

В статическом классе `Convert` определены методы для преобразования каждого базового типа в любой другой базовый тип. К сожалению, большинство таких методов бесполезны: они либо генерируют исключения, либо избыточны из-за доступности неявных приведений. Тем не менее, среди этого беспорядка есть несколько полезных методов, которые рассматриваются в последующих разделах.



Все базовые типы (явно) реализуют интерфейс `IConvertible`, в котором определены методы для преобразования во все другие базовые типы. В большинстве случаев реализация каждого из таких методов просто вызывает какой-то метод из класса `Convert`. В редких ситуациях может оказаться удобным создать метод, принимающий аргумент типа `IConvertible`.

Округляющие преобразования вещественных чисел в целые

В главе 2 было показано, что неявные и явные приведения позволяют выполнять преобразования между числовыми типами. Подведем итоги:

- неявные приведения работают для преобразований без потери (например, `int` в `double`);
- явные приведения обязательны для преобразований с потерей (например, `double` в `int`).

Приведения оптимизированы для обеспечения эффективности, поэтому они *усекают* данные, которые не умещаются. В результате может возникнуть проблема при преобразовании вещественного числа в целое, т.к. часто требуется не *усечение*, а *округление*. Упомянутую проблему решают методы числового преобразования `Convert`; они всегда производят *округление*:

```
double d = 3.9;  
int i = Convert.ToInt32 (d); // i == 4
```

Класс `Convert` использует *округление, принятое в банках*, при котором серединные значения привязываются к четным целым (это позволяет избегать положительного или отрицательного отклонения). Если округление, принятое в банках, становится проблемой, тогда для вещественного числа необходимо вызвать метод `Math.Round`: он принимает дополнительный аргумент, позволяющий управлять округлением серединного значения.

Разбор чисел в системах счисления с основаниями 2, 8 и 16

Среди методов `ToЦелочисленный`-типа скрываются перегруженные версии, которые разбирают числа в системах счисления с другими основаниями:

```
int thirty = Convert.ToInt32 ("1E", 16);    // Разобрать в шестнадцатеричной
                                                // системе счисления
uint five  = Convert.ToUInt32 ("101", 2);    // Разобрать в двоичной системе
                                                // счисления
```

Во втором аргументе задается основание системы счисления. Допускается указывать 2, 8, 10 или 16.

Динамические преобразования

Иногда требуется преобразовывать из одного типа в другой, но точные типы не известны вплоть до времени выполнения. Для таких целей класс `Convert` предлагает метод `ChangeType`:

```
public static object ChangeType (object value, Type conversionType);
```

Исходный и целевой типы должны относиться к “базовым” типам. Метод `ChangeType` также принимает необязательный аргумент `IFormatProvider`. Ниже представлен пример:

```
Type targetType = typeof (int);
object source = "42";

object result = Convert.ChangeType (source, targetType);

Console.WriteLine (result);                // 42
Console.WriteLine (result.GetType());      // System.Int32
```

Примером, когда это может оказаться полезным, является написание десериализатора, который способен работать с множеством типов. Он также может преобразовывать любое перечисление в его целочисленный тип (как было показано в разделе “Перечисления” главы 3).

Ограничение метода `ChangeType` заключается в том, что для него нельзя указывать форматную строку или флаг разбора.

Преобразования Base-64

Временами возникает необходимость включать двоичные данные вроде растрового изображения в текстовый документ, такой как XML-файл или сообщение электронной почты. Base-64 — повсеместно применяемое средство кодирования двоичных данных в виде читабельных символов, которое использует 64 символа из набора ASCII.

Метод `ToBase64String` класса `Convert` осуществляет преобразование байтового массива в код Base-64, а метод `FromBase64String` выполняет обратное преобразование.

Класс `XmlConvert`

Класс `XmlConvert` (из пространства имен `System.Xml`) предлагает наиболее подходящие методы для форматирования и разбора данных, которые поступают из XML-файла или направляются в него. Методы в `XmlConvert` учитыва-

ют нюансы XML-форматирования, не требуя указания специальных форматных строк. Например, значение `true` в XML выглядит как `true`, но не `True`. Класс `XmlConvert` широко применяется внутри библиотеки .NET BCL. Он также хорошо подходит для универсальной и не зависящей от культуры сериализации. Все методы форматирования в `XmlConvert` доступны в виде перегруженных версий методов `ToString`; методы разбора называются `ToBoolean`, `ToDateTime` и т.д.:

```
string s = XmlConvert.ToString (true);           // s = "true"  
bool isTrue = XmlConvert.ToBoolean (s);
```

Методы, выполняющие преобразование в и из `DateTime`, принимают аргумент типа `XmlDateTimeSerializationMode` — перечисление со следующими значениями:

`Unspecified`, `Local`, `Utc`, `RoundtripKind`

Значения `Local` и `Utc` вызывают преобразование во время форматирования (если `DateTime` еще не находится в нужном часовом поясе). Часовой пояс затем добавляется к строке:

2010-02-22T14:08:30.9375	// Unspecified
2010-02-22T14:07:30.9375+09:00	// Local
2010-02-22T05:08:30.9375Z	// Utc

Значение `Unspecified` приводит к отбрасыванию перед форматированием любой информации о часовом поясе, встроенной в `DateTime` (т.е. `DateTimeKind`). Значение `RoundtripKind` учитывает `DateTimeKind` из `DateTime`, так что при восстановлении результирующая структура `DateTime` будет в точности такой же, какой была изначально.

Преобразователи типов

Преобразователи типов предназначены для форматирования и разбора в средах, используемых во время проектирования. Преобразователи типов вдобавок поддерживают разбор значений в документах XAML (Extensible Application Markup Language — расширяемый язык разметки приложений), которые применяются в инфраструктуре Windows Presentation Foundation (WPF).

В .NET существует свыше 100 преобразователей типов, охватывающих такие аспекты, как цвета, изображения и URI. В отличие от них поставщики форматов реализованы только для небольшого количества простых типов значений.

Преобразователи типов обычно разбирают строки разнообразными путями, не требуя подсказок. Например, если при создании в Visual Studio приложения WPF присвоить элементу управления цвет фона, введя "Beige" в поле соответствующего свойства, то преобразователь типа `Color` определит, что производится ссылка на имя цвета, а не на строку RGB или системный цвет. Подобная гибкость иногда может делать преобразователи типов удобными в контекстах, выходящих за рамки визуальных конструкторов и XAML-документов.

Все преобразователи типов являются подклассами класса `TypeConverter` из пространства имен `System.ComponentModel`. Для получения экземпляра `TypeConverter` необходимо вызвать метод `TypeDescriptor.GetConverter`.

Следующий код получает экземпляр TypeConverter для типа Color (из пространства имен System.Drawing):

```
TypeConverter cc = TypeDescriptor.GetConverter (typeof (Color));
```

Среди многих других методов в классе TypeConverter определены методы ConvertToString и ConvertFromString. Их можно вызывать так, как показано ниже:

```
Color beige = (Color) cc.ConvertFromString ("Beige");
Color purple = (Color) cc.ConvertFromString ("#800080");
Color window = (Color) cc.ConvertFromString ("Window");
```

По соглашению преобразователи типов имеют имена, заканчивающиеся на Converter, и обычно находятся в том же самом пространстве имен, что и тип, для которого они предназначены. Тип ссылается на свой преобразователь через атрибут TypeConverterAttribute, позволяя визуальным конструкторам автоматически выбирать преобразователи.

Преобразователи типов могут также предоставлять службы стадии проектирования, такие как генерация списков стандартных значений для заполнения раскрывающихся списков в визуальном конструкторе или для помощи в написании кода сериализации.

Класс BitConverter

Большинство базовых типов может быть преобразовано в байтовый массив путем вызова метода BitConverter.GetBytes:

```
foreach (byte b in BitConverter.GetBytes (3.5))
    Console.Write (b + " ");
// 0 0 0 0 0 12 64
```

Класс BitConverter также предоставляет методы для преобразования в другом направлении, например, ToDouble.

Типы decimal и DateTime (DateTimeOffset) не поддерживаются классом BitConverter. Однако значение decimal можно преобразовать в массив int, вызвав метод decimal.GetBits. Для обратного направления тип decimal предлагает конструктор, принимающий массив int.

В случае типа DateTime можно вызывать метод ToBinary на экземпляре — в результате возвращается значение long (в отношении которого затем можно использовать BitConverter). Статический метод DateTime.FromBinary выполняет обратное действие.

Глобализация

С интернационализацией приложения связаны два аспекта: глобализация и локализация.

Глобализация занимается следующими тремя задачами (указанными в порядке убывания важности).

1. Обеспечение работоспособности программы при запуске на машине с другой культурой.

2. Соблюдение правил форматирования локальной культуры — например, при отображении дат.
3. Проектирование программы таким образом, чтобы она выбирала специфичные к культуре данные и строки из подчиненных сборок, которые можно написать и развернуть позже.

Локализация означает решение последней задачи за счет написания подчиненных сборок для специфических культур. Вы можете заняться этим *после* написания своей программы — мы раскроем соответствующие детали в разделе “Ресурсы и подчиненные сборки” главы 17.

.NET содействует решению второй задачи, применяя специфичные для культуры правила по умолчанию. Мы уже показывали, что вызов метода `ToString` на `DateTime` или числе учитывает локальные правила форматирования. К сожалению, в таком случае очень легко допустить ошибку при решении первой задачи и нарушить работу программы, поскольку ожидается, что даты или числа должны быть сформатированы в соответствии с предполагаемой культурой. Как уже было продемонстрировано, решение предусматривает либо указание культуры (такой как инвариантная культура) при форматировании и разборе, либо использование независимых от культуры методов вроде тех, которые определены в классе `XmlConvert`.

Контрольный перечень глобализации

Мы уже рассмотрели в текущей главе важные моменты, связанные с глобализацией. Ниже приведен сводный перечень основных требований.

- Освойте Unicode и текстовые кодировки (см. раздел “Кодировка текста и Unicode” ранее в главе).
- Помните о том, что такие методы, как `ToUpper` и `ToLower` в типах `char` и `string`, чувствительны к культуре: если чувствительность к культуре не нужна, тогда применяйте методы `ToUpperInvariant`/`ToLowerInvariant`.
- Отдавайте предпочтение независимым от культуры механизмам форматирования и разбора для `DateTime` и `DateTimeOffset`, таким как `ToString("o")` и `XmlConvert`.
- В других обстоятельствах указывайте культуру при форматировании/разборе чисел или даты/времени (если только вас не интересует поведение локальной культуры).

Тестирование

Проводить тестирование для различных культур можно путем переустановки свойства `CurrentCulture` класса `Thread` (из пространства имен `System.Threading`). В представленном далее коде текущая культура изменяется на турецкую:

```
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo ("tr-TR");
```

Турецкая культура является очень хорошим тестовым сценарием по следующим причинам.

- "i".ToUpper() != "I" и "I".ToLower() != "i".
- Даты форматируются как день.месяц.год (обратите внимание на разделитель в виде точки).
- Символом десятичной точки является запятая, а не сама точка.

Можно также поэкспериментировать, изменяя настройки форматирования чисел и дат в панели управления Windows: они отражены в стандартной культуре (CultureInfo.CurrentCulture).

Метод CultureInfo.GetCultures возвращает массив всех доступных культур.



Классы Thread и CultureInfo также поддерживают свойство CurrentUICulture. Оно больше связано с локализацией, которую мы рассмотрим в главе 17.

Работа с числами

Преобразования

Числовые преобразования были описаны в предшествующих главах и разделах; все доступные варианты подытожены в табл. 6.7.

Таблица 6.7. Обзор числовых преобразований

Задача	Функции	Примеры
Разбор десятичных чисел	Parse TryParse	double d = double.Parse ("3.5"); int i; bool ok = int.TryParse ("3", out i);
Разбор чисел в системах счисления с основаниями 2, 8 или 16	Convert. ToInt32 ("1E", 16);	int i = Convert.ToInt32 ("1E", 16);
Форматирование в шестнадцатеричную запись	ToString ("X")	string hex = 45.ToString ("X");
Числовое преобразование без потерь	Нев явное приведение	int i = 23; double d = i;
Числовое преобразование с усечением	Явное приведение	double d = 23.5; int i = (int) d;
Числовое преобразование с округлением (вещественных чисел в целые)	Convert. ToInt32 (d);	double d = 23.5; int i = Convert.ToInt32 (d);

Класс Math

В табл. 6.8 представлен список основных членов статического класса Math. Тригонометрические функции принимают аргументы типа double; другие методы наподобие Max перегружены для работы со всеми числовыми типами.

В классе `Math` также определены математические константы `E` (e) и `PI` (π).

Таблица 6.8. Методы статического класса Math

Категория	Методы
Округление	Round, Truncate, Floor, Ceiling
Максимум и минимум	Max, Min
Абсолютное значение и знак	Abs, Sign
Квадратный корень	Sqrt
Возведение в степень	Pow, Exp
Логарифм	Log, Log10
Тригонометрические функции	Sin, Cos, Tan, Sinh, Cosh, Tanh, Asin, Acos, Atan

Метод Round позволяет указывать количество десятичных позиций для округления, а также способ обработки серединных значений (от нуля или посредством округления, принятого в банках). Методы Floor и Ceiling округляют до ближайшего целого: Floor всегда округляет в меньшую сторону, а Ceiling — в большую, даже отрицательные числа.

Методы Max и Min принимают только два аргумента. Для массива или последовательности чисел следует применять расширяющие методы Max и Min из класса System.Linq.Enumerable.

Структура BigInteger

Структура BigInteger — это специализированный числовой тип, который находится в новом пространстве имен System.Numerics и позволяет представлять произвольно большие целые числа без потери точности.

В языке C# отсутствует собственная поддержка BigInteger, так что нет никакого способа записи литералов типа BigInteger. Тем не менее, можно выполнять неявное преобразование в BigInteger из любого другого целочисленного типа:

```
BigInteger twentyFive = 25; //Неявное преобразование из целочисленного типа
```

Чтобы представить крупное число вроде одного гугола (10100), можно воспользоваться одним из статических методов BigInteger, например, Pow (возведение в степень):

```
BigInteger googol = BigInteger.Pow (10, 100);
```

Или же можно применить метод Parse для разбора строки:

```
BigInteger googol = BigInteger.Parse ("1".PadRight (101, '0'));
```

Вызов метода `ToString` в данном случае приводит к выводу на экран всех цифр:

Между BigInteger и стандартными числовыми типами можно выполнять преобразования с потенциальной потерей точности, используя явную операцию приведения:

```
double g2 = (double) googol;           // Явное приведение
BigInteger g3 = (BigInteger) g2;       // Явное приведение
Console.WriteLine (g3);
```

Вывод демонстрирует потерю точности:

```
999999999999999673361688041166912...
```

В структуре BigInteger перегружены все арифметические операции, включая получение остатка от деления (%), а также операции сравнения и эквивалентности.

Структуру BigInteger можно также конструировать из байтового массива. Следующий код генерирует 32-байтовое случайное число, подходящее для криптографии, и затем присваивает его переменной BigInteger:

```
// Здесь используется пространство имен System.Security.Cryptography:
RandomNumberGenerator rand = RandomNumberGenerator.Create();
byte[] bytes = new byte [32];
rand.GetBytes (bytes);
var bigRandomNumber = new BigInteger (bytes); // Преобразовать в BigInteger
```

Преимущество хранения таких чисел в структуре BigInteger по сравнению с байтовым массивом связано с тем, что вы получаете семантику типа значения. Вызов ToByteArray осуществляет преобразование BigInteger обратно в байтовый массив.

Структура Half

Структура Half представляет собой 16-битный тип с плавающей точкой; она появилась в версии .NET 5. Данная структура предназначена главным образом для взаимодействия с процессорами графических плат и не имеет собственной поддержки в большинстве центральных процессоров.

Выполнять преобразования между Half и float или double можно через явное приведение:

```
Half h = (Half) 123.456;
Console.WriteLine (h); // 123.44 (обратите внимание на потерю точности)
```

Для данного типа не определены арифметические операции, поэтому выполнение вычислений требует его преобразования в другой тип, такой как float или double.

Структура Half имеет диапазон от -65 500 до 65 500:

```
Console.WriteLine (Half.MinValue); // -65500
Console.WriteLine (Half.MaxValue); // 65500
```

Обратите внимание на потерю точности при значениях, близких к максимальному значению диапазона:

```
Console.WriteLine ((Half)65500); // 65500
Console.WriteLine ((Half)65490); // 65500
Console.WriteLine ((Half)65480); // 65470
```

Структура Complex

Структура Complex является еще одним специализированным числовым типом, который предназначен для представления комплексных чисел с помощью вещественных и мнимых компонентов типа double. Структура Complex находится в том же пространстве имен, что и BigInteger.

Для применения Complex необходимо создать экземпляр структуры, указав вещественное и мнимое значения:

```
var c1 = new Complex (2, 3.5);
var c2 = new Complex (3, 0);
```

Существуют также неявные преобразования в Complex из стандартных числовых типов.

Структура Complex предлагает свойства для вещественного и мнимого значений, а также для фазы и амплитуды:

```
Console.WriteLine (c1.Real);           // 2
Console.WriteLine (c1.Imaginary);       // 3.5
Console.WriteLine (c1.Phase);          // 1.05165021254837
Console.WriteLine (c1.Magnitude);      // 4.03112887414927
```

Конструировать число Complex можно путем указания амплитуды и фазы:

```
Complex c3 = Complex.FromPolarCoordinates (1.3, 5);
```

Стандартные арифметические операции перегружены для работы с числами Complex:

```
Console.WriteLine (c1 + c2);           // (5, 3.5)
Console.WriteLine (c1 * c2);           // (6, 10.5)
```

Структура Complex предоставляет статические методы для выполнения более сложных функций, включая:

- тригонометрические (Sin, Asin, Sinh, Tan и т.д.);
- логарифмы и возведения в степень;
- сопряженное число (Conjugate).

Класс Random

Класс Random генерирует псевдослучайную последовательность значений byte, int или double.

Чтобы использовать класс Random, сначала понадобится создать его экземпляр, дополнительно предоставив начальное значение для инициализации последовательности случайных чисел. Применение одного и того же начального значения гарантирует получение той же самой последовательности чисел (при запуске под управлением одной и той же версии CLR), что иногда полезно, когда нужна воспроизводимость:

```
Random r1 = new Random (1);
Random r2 = new Random (1);
Console.WriteLine (r1.Next (100) + ", " + r1.Next (100)); // 24, 11
Console.WriteLine (r2.Next (100) + ", " + r2.Next (100)); // 24, 11
```

 Если воспроизводимость не требуется, тогда можно конструировать Random без начального значения — в качестве такого значения будет использоваться текущее системное время.

Поскольку системные часы имеют ограниченный квант времени, два экземпляра Random, созданные в близкие друг к другу моменты времени (обычно в пределах 10 мс), будут выдавать одинаковые последовательности значений. Указанную проблему в общем случае решают путем создания нового объекта Random каждый раз, когда требуется случайное число, вместо повторного использования *того же самого объекта*.

Удачный прием предусматривает объявление единственного статического экземпляра Random. Однако в многопоточных сценариях это может привести к проблемам, т.к. объекты Random не являются безопасными в отношении потоков. В разделе “Локальное хранилище потока” главы 21 будет описан обходной способ.

Вызов метода Next (n) генерирует случайное целочисленное значение между 0 и n-1. Вызов метода NextDouble генерирует случайное значение типа double между 0 и 1. Вызов метода NextBytes заполняет случайными значениями байтовый массив.

Начиная с версии .NET 8, класс Random имеет метод GetItems, который выбирает n случайных элементов из коллекции. С помощью следующего кода из коллекции, содержащей пять чисел, выбираются два случайных числа:

```
int[] numbers = { 10, 20, 30, 40, 50 };
int[] randomTwo = new Random().GetItems (numbers, 2);
```

В .NET 8 также появился метод Shuffle, позволяющий рандомизировать порядок следования элементов внутри массива или диапазона.

Класс Random не считается в достаточной степени случайным для приложений, предъявляющих высокие требования к безопасности, таким как криптографические программы. Для них в .NET предлагается *криптографически строгий* генератор случайных чисел, находящийся в пространстве имен System.Security.Cryptography. Вот как его применять:

```
var rand = System.Security.Cryptography.RandomNumberGenerator.Create();
byte[] bytes = new byte [32];
rand.GetBytes (bytes); // Заполнить байтовый массив случайными числами
```

Недостаток класса RandomNumberGenerator в том, что он менее гибкий: заполнение байтового массива является единственным средством получения случайных чисел. Чтобы получить целое число, потребуется использовать BitConverter:

```
byte[] bytes = new byte [4];
rand.GetBytes (bytes);
int i = BitConverter.ToInt32 (bytes, 0);
```

Класс BitOperations

Класс `System.Numerics.BitOperations` (появившийся в .NET 6) предоставляет следующие методы, которые помогают выполнять операции в двоичной системе счисления.

`IsPow2`

Возвращает `true`, если число является степенью 2.

`LeadingZeroCount/TrailingZeroCount`

Возвращает количество ведущих нулей в целом числе, представленном в 32- или 64-битном двоичном формате без знака.

`Log2`

Возвращает двоичный логарифм целого числа без знака.

`PopCount`

Возвращает количество бит, установленных в 1, в целом числе без знака.

`RotateLeft/RotateRight`

Выполняет побитовое вращение влево/вправо.

`RoundUpToPowerOf2`

Округляет целое число без знака до ближайшей степени 2.

Перечисления

В главе 3 был описан тип `enum`, а также показано, каким образом комбинировать его члены, проверять эквивалентность, применять логические операции и выполнять преобразования. Инфраструктура .NET расширяет поддержку перечислений в C# через тип `System.Enum`, исполняющий две роли:

- обеспечение унификации для всех типов `enum`;
- определение статических служебных методов.

Унификация типов означает возможность неявного приведения любого члена перечисления к экземпляру `System.Enum`:

```
Display (Nut.Macadamia);           // Nut.Macadamia
Display (Size.Large);              // Size.Large
void Display (Enum value)
{
    Console.WriteLine (value.GetType().Name + "." + value.ToString());
}
enum Nut { Walnut, Hazelnut, Macadamia }
enum Size { Small, Medium, Large }
```

Статические служебные методы `System.Enum` относятся главным образом к выполнению преобразований и получению списков членов.

Преобразования для перечислений

Представить значение перечисления можно тремя способами:

- как член перечисления;
- как лежащее в основе целочисленное значение;
- как строку.

В этом разделе будет показано, каким образом осуществлять преобразования между всеми представлениями.

Преобразования члена перечисления в целое значение

Вспомните, что явное приведение выполняет преобразование между членом перечисления и его целочисленным значением. Явное приведение будет корректным подходом, если тип enum известен на этапе компиляции:

```
[Flags] public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
int i = (int) BorderSides.Top;           // i == 4
BorderSides side = (BorderSides) i;      // side == BorderSides.Top
```

Таким же способом можно приводить экземпляр System.Enum к его целочисленному типу. Трюк заключается в приведении сначала к object, а затем к целочисленному типу:

```
static int GetIntegralValue (Enum anyEnum)
{
    return (int) (object) anyEnum;
}
```

Код полагается на то, что вам известен целочисленный тип: приведенный выше метод потерпит неудачу, если ему передать член перечисления, целочисленным типом которого является long. Чтобы написать метод, работающий с перечислением любого целочисленного типа, можно воспользоваться одним из трех подходов. Первый из них предусматривает вызов метода Convert.ToDecimal:

```
static decimal GetAnyIntegralValue (Enum anyEnum)
{
    return Convert.ToDecimal (anyEnum);
}
```

Данный подход работает, потому что каждый целочисленный тип (включая ulong) может быть преобразован в десятичный тип без потери информации. Второй подход предполагает вызов метода Enum.GetUnderlyingType для получения целочисленного типа перечисления и затем вызов метода Convert.ChangeType:

```
static object GetBoxedIntegralValue (Enum anyEnum)
{
    Type integralType = Enum.GetUnderlyingType (anyEnum.GetType ());
    return Convert.ChangeType (anyEnum, integralType);
}
```

Как показано в следующем примере, в результате предохраняется исходный целочисленный тип:

```
object result = GetBoxedIntegralValue (BorderSides.Top);
Console.WriteLine (result);           // 4
Console.WriteLine (result.GetType()); // System.Int32
```



Наш метод `GetBoxedIntegralType` фактически не выполняет никаких преобразований значения; взамен он *переупаковывает* то же самое значение в другой тип. Он транслирует целочисленное значение внутри оболочки *типа перечисления* в целое значение внутри оболочки *целочисленного типа*. Мы рассмотрим это более подробно в разделе “Как работают перечисления” далее в главе.

Третий подход заключается в вызове метода `Format` или `ToString` с указанием форматной строки "d" или "D". В итоге получается целочисленное значение перечисления в виде строки, что удобно при написании специальных форматеров сериализации:

```
static string GetIntegralValueAsString (Enum anyEnum)
{
    return anyEnum.ToString ("D"); // Возвращает что-то наподобие "4"
```

Преобразования целочисленного значения в член перечисления

Метод `Enum.ToObject` преобразует целочисленное значение в член перечисления заданного типа:

```
object bs = Enum.ToObject (typeof (BorderSides), 3);
Console.WriteLine (bs); // Left, Right
```

Это динамический эквивалент следующего кода:

```
BorderSides bs = (BorderSides) 3;
```

Метод `ToObject` перегружен для приема всех целочисленных типов, а также типа `object`. (Последняя перегруженная версия работает с любым упакованым целочисленным типом.)

Строковые преобразования

Для преобразования перечисления в строку можно вызвать либо статический метод `Enum.Format`, либо метод `ToString` на экземпляре. Оба метода принимают форматную строку, которой может быть "G" для стандартного поведения форматирования, "D" для выдачи лежащего в основе целочисленного значения в виде строки, "X" для выдачи лежащего в основе целочисленного значения в виде шестнадцатеричной записи или "F" для форматирования комбинированных членов перечисления без атрибута `Flags`. Примеры приводились в разделе “Стандартные форматные строки и флаги разбора” ранее в главе.

Метод `Enum.Parse` преобразует строку в перечисление. Он принимает тип `enum` и строку, которая может содержать множество членов:

```
BorderSides leftRight = (BorderSides) Enum.Parse (typeof (BorderSides),
"Left, Right");
```

Необязательный третий аргумент позволяет выполнять разбор, нечувствительный к регистру. Если член не найден, тогда генерируется исключение `ArgumentException`.

Перечисление значений перечисления

Метод `Enum.GetValues` возвращает массив, содержащий все члены указанного типа перечисления:

```
foreach (Enum value in Enum.GetValues (typeof (BorderSides)))
    Console.WriteLine (value);
```

В массив включаются и составные члены, такие как `LeftRight=Left | Right`.

Метод `Enum.GetNames` выполняет то же самое действие, но возвращает массив строк.



Внутренне среда CLR реализует методы `GetValues` и `GetNames`, выполняя рефлексию полей в типе перечисления. В целях эффективности результаты кешируются.

Как работают перечисления

Семантика типов перечислений в значительной степени поддерживается компилятором. В среде CLR во время выполнения не делается никаких отличий между экземпляром перечисления (когда он не упакован) и лежащим в его основе целочисленным значением. Более того, определение `enum` в CLR является просто подтиповом `System.Enum` со статическими полями целочисленного типа для каждого члена. В итоге обычное применение `enum` становится высокоэффективным, приводя к затратам во время выполнения, которые соответствуют затратам, связанным с целочисленными константами.

Недостаток данной стратегии состоит в том, что типы перечислений могут обеспечивать *статическую*, но не *строгую* безопасность типов. Мы приводили пример в главе 3:

```
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
BorderSides b = BorderSides.Left;
b += 1234; // Ошибка не возникает!
```

Когда компилятор не имеет возможности выполнить проверку достоверности (как в показанном примере), то нет никакой подстраховки со стороны исполняющей среды в форме генерации исключения.

Может показаться, что утверждение об отсутствии разницы между экземпляром перечисления и его целочисленным значением во время выполнения вступает в противоречие со следующим кодом:

```
[Flags] public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
Console.WriteLine (BorderSides.Right.ToString()); // Right
Console.WriteLine (BorderSides.Right.GetType().Name); // BorderSides
```

Учитывая природу экземпляра перечисления во время выполнения, можно было бы ожидать вывода на экран 2 и `Int32`! Причина такого поведения кроется в определенных уловках, предпринимаемых компилятором. Компилятор C# явно *упаковывает* экземпляр перечисления перед вызовом его виртуальных ме-

тодов, таких как `ToString` и `GetType`. И когда экземпляр перечисления упакован, он получает оболочку времени выполнения, которая ссылается на его тип перечисления.

Структура Guid

Структура `Guid` представляет глобально уникальный идентификатор: 16-байтовое значение, которое после генерации является почти наверняка уникальным в мире. Идентификаторы `Guid` часто используются для ключей различных видов — в приложениях и базах данных. Количество уникальных идентификаторов `Guid` составляет 2128, или $3,4 \times 10^{38}$. Статический метод `Guid.NewGuid` генерирует уникальный идентификатор `Guid`:

```
Guid g = Guid.NewGuid();  
Console.WriteLine(g.ToString()); // 0d57629c-7d6e-4847-97cb-9e2fc25083fe
```

Для создания идентификатора `Guid` с применением существующего значения предназначено несколько конструкторов. Вот два наиболее полезных из них:

```
public Guid (byte[] b); // Принимает 16-байтовый массив  
public Guid (string g); // Принимает форматированную строку
```

В случае представления в виде строки идентификатор `Guid` форматируется как шестнадцатеричное число из 32 цифр с необязательными символами — после 8-й, 12-й, 16-й и 20-й цифры. Вся строка также может быть дополнительно помещена в квадратные или фигурные скобки:

```
Guid g1 = new Guid ("{0d57629c-7d6e-4847-97cb-9e2fc25083fe}");  
Guid g2 = new Guid ("0d57629c7d6e484797cb9e2fc25083fe");  
Console.WriteLine (g1 == g2); // True
```

Будучи структурой, `Guid` поддерживает семантику типа значения; следовательно, в показанном выше примере операция эквивалентности работает нормально.

Метод `ToByteArray` преобразует `Guid` в байтовый массив.

Статическое свойство `Guid.Empty` возвращает пустой идентификатор `Guid` (со всеми нулями), который часто используется вместо `null`.

Сравнение эквивалентности

До сих пор мы предполагали, что все применяемые операции `==` и `!=` выполняли сравнение эквивалентности. Однако проблема эквивалентности является более сложной и тонкой, временами требуя использования дополнительных методов и интерфейсов. В настоящем разделе мы исследуем стандартные протоколы C# и .NET для определения эквивалентности, уделяя особое внимание следующим двум вопросам.

- Когда операции `==` и `!=` адекватны (либо неадекватны) для сравнения эквивалентности и какие существуют альтернативы?
- Как и когда должна настраиваться логика эквивалентности типа?

Но перед тем как погрузиться в исследование протоколов эквивалентности и способов их настройки мы должны взглянуть на вводные концепции эквивалентности значений и ссылочной эквивалентности.

Эквивалентность значений и ссылочная эквивалентность

Различают два вида эквивалентности.

- **Эквивалентность значений.** Два значения эквивалентны в некотором смысле.
- **Ссылочная эквивалентность.** Две ссылки ссылаются в точности на один и тот же объект.

Если только не было переопределено, то:

- типы значений применяют **эквивалентность значений**;
- ссылочные типы используют **ссылочную эквивалентность** (это переопределяется в анонимных типах и записях).

На самом деле типы значений могут применять *только* эквивалентность значений (если они не упакованы). Сравнение двух чисел служит простой демонстрацией эквивалентности значений:

```
int x = 5, y = 5;
Console.WriteLine (x == y);           // True (в силу эквивалентности значений)
```

Более сложная демонстрация предусматривает сравнение двух структур `DateTimeOffset`. Приведенный ниже код выводит на экран `True`, т.к. две структуры `DateTimeOffset` относятся к *одной и той же точке во времени*, а потому считаются эквивалентными:

```
var dt1 = new DateTimeOffset (2010, 1, 1, 1, 1, 1, TimeSpan.FromHours(8));
var dt2 = new DateTimeOffset (2010, 1, 1, 2, 1, 1, TimeSpan.FromHours(9));
Console.WriteLine (dt1 == dt2);      // True
```



Тип `DateTimeOffset` — это структура, семантика эквивалентности которой была настроена. По умолчанию структуры поддерживают специальный вид эквивалентности значений, называемый *структурной эквивалентностью*, при которой два значения считаются эквивалентными, если эквивалентны все их члены. (Вы можете удостовериться в сказанном, создав структуру и вызвав ее метод `Equals`; позже будут приведены более подробные обсуждения.)

Ссылочные типы по умолчанию поддерживают ссылочную эквивалентность. В следующем примере ссылки `f1` и `f2` не являются эквивалентными — несмотря на то, что их объекты имеют идентичное содержимое:

```
class Foo { public int X; }
...
Foo f1 = new Foo { X = 5 };
Foo f2 = new Foo { X = 5 };
Console.WriteLine (f1 == f2);      // False
```

Напротив, f3 и f1 эквивалентны, поскольку они ссылаются на тот же самый объект:

```
Foo f3 = f1;
Console.WriteLine (f1 == f3); // True
```

Позже в этом разделе мы объясним, каким образом можно настраивать ссылочные типы для обеспечения эквивалентности значений. Примером будет служить класс Uri из пространства имен System:

```
Uri uri1 = new Uri ("http://www.linqpad.net");
Uri uri2 = new Uri ("http://www.linqpad.net");
Console.WriteLine (uri1 == uri2); // True
```

Класс string обладает похожим поведением:

```
var s1 = "http://www.linqpad.net";
var s2 = "http://" + "www.linqpad.net";
Console.WriteLine (s1 == s2); // True
```

Стандартные протоколы эквивалентности

Существуют три стандартных протокола, которые типы могут соблюдать при сравнении эквивалентности:

- операции == и !=;
- виртуальный метод Equals в классе object;
- интерфейс IEquatable<T>.

Вдобавок есть *подключаемые* протоколы и интерфейс IStructuralEquatable, который мы рассмотрим в главе 7.

Операции == и !=

Вы уже видели в многочисленных примерах, каким образом стандартные операции == и != производят сравнения равенства/неравенства. Тонкости с == и != возникают из-за того, что они являются *операциями*, а потому распознаются статически (на самом деле они реализованы в виде статических функций). Следовательно, когда вы используете операцию == или !=, то решение о том, какой тип будет выполнять сравнение, принимается на этапе компиляции, и виртуальное поведение в игру не вступает. Обычно подобное и желательно. В показанном далее примере компилятор жестко привязывает операцию == к типу int, т.к. переменные x и y имеют тип int:

```
int x = 5;
int y = 5;
Console.WriteLine (x == y); // True
```

Но в представленном ниже примере компилятор привязывает операцию == к типу object:

```
object x = 5;
object y = 5;
Console.WriteLine (x == y); // False
```

Поскольку `object` — класс (т.е. ссылочный тип), операция `==` типа `object` применяет для сравнения `x` и `y` *ссылочную эквивалентность*. Результатом будет `false`, потому что `x` и `y` ссылаются на разные упакованные объекты в куче.

Виртуальный метод `Object.Equals`

Для корректного сравнения `x` и `y` в предыдущем примере мы можем использовать виртуальный метод `Equals`. Метод `Equals` определен в классе `System.Object`, поэтому он доступен всем типам:

```
object x = 5;
object y = 5;
Console.WriteLine (x.Equals (y));      // True
```

Метод `Equals` распознается во время выполнения — согласно действительному типу объекта. В данном случае вызывается метод `Equals` типа `Int32`, который применяет к operandам *эквивалентность значений*, возвращая `true`. Со ссылочными типами метод `Equals` по умолчанию выполняет сравнение ссылочной эквивалентности; для структур метод `Equals` производит сравнение структурной эквивалентности, вызывая `Equals` на каждом их поле.

Чем объясняется подобная сложность?

У вас может возникнуть вопрос, почему разработчики C# не попытались избежать проблемы, сделав операцию `==` виртуальной и таким образом функционально идентичной `Equals`? На то было несколько причин.

- Если первый operand равен `null`, то метод `Equals` терпит неудачу с генерацией исключения `NullReferenceException`, а статическая операция — нет.
- Поскольку операция `==` распознается статически, она выполняется очень быстро. Это означает, что вы можете записывать код с интенсивными вычислениями без нанесения ущерба производительности — и без необходимости в изучении другого языка, такого как C++.
- Иногда полезно обеспечить для операции `==` и метода `Equals` разные определения эквивалентности. Мы опишем такой сценарий позже в разделе.

По существу сложность реализованного проектного решения отражает сложность самой ситуации: концепция эквивалентности охватывает большое число сценариев.

Следовательно, метод `Equals` подходит для сравнения двух объектов в независимой от типа манере. Показанный ниже метод сравнивает два объекта любого типа:

```
public static bool AreEqual (object obj1, object obj2)
=> obj1.Equals (obj2);
```

Тем не менее, есть один сценарий, при котором такой подход не работает. Если первый аргумент равен null, тогда генерируется исключение NullReferenceException. Ниже приведена исправленная версия метода:

```
public static bool AreEqual (object obj1, object obj2)
{
    if (obj1 == null) return obj2 == null;
    return obj1.Equals (obj2);
}
```

Или более лаконично:

```
public static bool AreEqual (object obj1, object obj2)
    => obj1 == null ? obj2 == null : obj1.Equals (obj2);
```

Статический метод Object.Equals

В классе System.Object определен статический вспомогательный метод, который выполняет работу метода AreEqual из предыдущего примера. Он имеет имя Equals (точно как у виртуального метода), но никаких конфликтов не возникает, потому что данный метод принимает *два* аргумента:

```
public static bool Equals (object objA, object objB)
```

Статический метод Object.Equals предоставляет алгоритм сравнения эквивалентности, безопасный к null, который предназначен для ситуаций, когда типы не известны на этапе компиляции:

```
object x = 3, y = 3;
Console.WriteLine (object.Equals (x, y)); // True
x = null;
Console.WriteLine (object.Equals (x, y)); // False
y = null;
Console.WriteLine (object.Equals (x, y)); // True
```

Метод полезен при написании обобщенных типов. Приведенный ниже код не скомпилируется, если вызов Object.Equals заменить операцией == или !=:

```
class Test <T>
{
    T _value;
    public void SetValue (T newValue)
    {
        if (!object.Equals (newValue, _value))
        {
            _value = newValue;
            OnValueChanged();
        }
    }
    protected virtual void OnValueChanged() { ... }
}
```

Операции здесь запрещены, потому что компилятор не может выполнить связывание со статическим методом неизвестного типа.



Более аккуратный способ реализации такого сравнения предусматривает использование класса EqualityComparer<T>. Преимущество заключается в том, что тогда удается избежать упаковки:

```
if (!EqualityComparer<T>.Default.Equals (newValue, _value))
```

Мы обсудим класс EqualityComparer<T> более подробно в разделе “Подключение протоколов эквивалентности и порядка” главы 7.

Статический метод Object.ReferenceEquals

Иногда требуется принудительно применять сравнение ссылочной эквивалентности. Статический метод Object.ReferenceEquals делает именно это:

```
Widget w1 = new Widget();
Widget w2 = new Widget();
Console.WriteLine (object.ReferenceEquals (w1, w2)); // False
class Widget { ... }
```

Поступать так может быть необходимо из-за того, что в классе Widget допускается переопределение виртуального метода Equals, в результате чего w1.Equals(w2) будет возвращать true. Более того, в Widget возможна перегрузка операции == и сравнение w1==w2 также будет давать true. В подобных случаях вызов Object.ReferenceEquals гарантирует использование нормальной семантики ссылочной эквивалентности.



Еще один способ обеспечения сравнения ссылочной эквивалентности предусматривает приведение значений к object с последующим применением операции ==.

Интерфейс IEquatable<T>

Последствием вызова метода Object.Equals является упаковка типов значений. В сценариях с высокой критичностью к производительности это не желательно, т.к. по сравнению с действительным сравнением упаковка будет относительно затратной в плане ресурсов. Решение данной проблемы появилось в C# 2.0 — интерфейс IEquatable<T>:

```
public interface IEquatable<T>
{
    bool Equals (T other);
}
```

Идея заключается в том, что реализация интерфейса IEquatable<T> обеспечивает такой же результат, как и вызов виртуального метода Equals из object, но только быстрее. Интерфейс IEquatable<T> реализован большинством базовых типов .NET. Интерфейс IEquatable<T> можно использовать как ограничение в обобщенном типе:

```
class Test<T> where T : IEquatable<T>
{
```

```
public bool IsEqual (T a, T b)
{
    return a.Equals (b); // Упаковка с обобщенным типом T не происходит
}
```

Если убрать ограничение обобщенного типа, то класс по-прежнему скомпилируется, но `a.Equals(b)` будет связываться с более медленным методом `object.Equals` (более медленным, исходя из предположения, что `T` — тип значения).

Когда метод Equals и операция == не эквивалентны

Ранее мы упоминали, что временами для операции `==` и метода `Equals` полезно применять разные определения эквивалентности. Например:

```
double x = double.NaN;
Console.WriteLine (x == x); // False
Console.WriteLine (x.Equals (x)); // True
```

Операция `==` в типе `double` гарантирует, что значение `NaN` никогда не может быть равным чему-либо еще — даже другому значению `NaN`. Это наиболее естественно с математической точки зрения и отражает внутреннее поведение центрального процессора. Однако метод `Equals` обязан использовать *рефлексивную* эквивалентность; другими словами, вызов `x.Equals(x)` должен всегда возвращать `true`.

На такое поведение `Equals` полагаются коллекции и словари; в противном случае они не смогут найти ранее сохраненный в них элемент.

Обеспечение отличающегося поведения эквивалентности в `Equals` и `==` для типов значений предпринимается довольно редко. Более распространенный сценарий относится к ссылочным типам и возникает, когда разработчик настраивает метод `Equals` так, что тот реализует эквивалентность значений, оставляя операцию `==` выполняющей (стандартную) ссылочную эквивалентность. Именно это делает класс `StringBuilder`:

```
var sb1 = new StringBuilder ("foo");
var sb2 = new StringBuilder ("foo");
Console.WriteLine (sb1 == sb2); // False (ссылочная эквивалентность)
Console.WriteLine (sb1.Equals (sb2)); // True (эквивалентность значений)
```

Теперь давайте посмотрим, как настраивать эквивалентность.

Эквивалентность и специальные типы

Вспомним стандартное поведение сравнения эквивалентности:

- типы значений применяют *эквивалентность значений*;
- ссылочные типы используют *ссылочную эквивалентность*, если это не переопределено (как в случае анонимных типов и записей).

Кроме того:

- по умолчанию метод `Equals` структуры применяет *структурную эквивалентность значений* (т.е. сравниваются все поля в структурах).

При написании типа такое поведение иногда имеет смысл переопределять. Существуют две ситуации, когда требуется переопределение:

- для изменения смысла эквивалентности;
- для ускорения сравнений эквивалентности в структурах.

Изменение смысла эквивалентности

Изменять смысл эквивалентности необходимо тогда, когда стандартное поведение операции `==` и метода `Equals` неестественно для типа и *не является тем поведением, которое будет ожидать потребитель*. Примером может служить `DateTimeOffset` — структура с двумя закрытыми полями: UTC-значение `DateTime` и целочисленное смещение. При написании подобного типа может понадобиться сделать так, чтобы сравнение эквивалентности принимало во внимание только поле с UTC-значением `DateTime` и не учитывало поле смещения. Другим примером являются числовые типы, поддерживающие значения `NaN`, вроде `float` и `double`. Если вы реализуете типы такого рода самостоятельно, то наверняка захотите обеспечить поддержку логики сравнения с `NaN` в сравнениях эквивалентности.

В случае классов иногда более естественно по умолчанию предлагать поведение эквивалентности значений, а не ссылочной эквивалентности. Это часто встречается в небольших классах, которые хранят простую порцию данных — например, `System.Uri` (или `System.String`).

С записями компилятор автоматически реализует структурную эквивалентность (сравнивая каждое поле). Тем не менее, иногда в процесс могут быть вовлечены поля, которые сравнивать нежелательно, или объекты, которые требуют специальной логики сравнения, такие как коллекции. Процесс переопределения эквивалентности с записями слегка отличается, поскольку записи следуют особому шаблону, который разработан, чтобы хорошо сочетаться с их правилами для наследования.

Ускорение сравнений эквивалентности для структур

Стандартный алгоритм сравнения *структурной эквивалентности* для структур является относительно медленным. Взяв на себя контроль над таким процессом за счет переопределения метода `Equals`, можно улучшить производительность в пять раз. Перегрузка операции `==` и реализация интерфейса `IEquatable<T>` делают возможными неупаковывающие сравнения эквивалентности, что также может ускорить производительность в пять раз.



Переопределение семантики эквивалентности для ссылочных типов не дает преимуществ в плане производительности. Стандартный алгоритм сравнения ссылочной эквивалентности уже характеризуется высокой скоростью, т.к. он просто сравнивает две 32- или 64-битные ссылки.

На самом деле существует другой, довольно специфический случай для настройки эквивалентности, который касается совершенствования алгоритма хеширования структуры в целях достижения лучшей производительности в

хеш-таблице. Это происходит из того факта, что сравнение эквивалентности и хеширование объединены в общий механизм. Хеширование рассматривается чуть позже в главе.

Как переопределить семантику эквивалентности

Чтобы переопределить эквивалентность для классов и структур, понадобится выполнить следующие шаги.

1. Переопределить методы GetHashCode и Equals.
2. (Дополнительно) перегрузить операции != и ==.
3. (Дополнительно) реализовать интерфейс IEquatable<T>.

Для записей процесс отличается (и он проще), потому что компилятор уже переопределяет методы и операции сравнения эквивалентности в соответствии с собственным специальным шаблоном. Если вы хотите вмешаться, то должны следовать этому шаблону, т.е. записывать метод Equals с сигнатурой следующего вида:

```
record Test (int X, int Y)
{
    public virtual bool Equals (Test t) => t != null && t.X == X && t.Y == Y;
}
```

Обратите внимание, что метод Equals определен как virtual (не override) и принимает фактический тип записи (в данном случае Test, а не object). Компилятор обнаружит, что ваш метод имеет “корректную” сигнатуру и вставит его.

Как и в случае классов или структур, вам также потребуется переопределить метод GetHashCode. Перегружать операции != и == или реализовывать интерфейс IEquatable<T> не нужно (и не следует), т.к. это уже сделано за вас.

Переопределение метода GetHashCode

Может показаться странным, что в классе System.Object — с его небольшим набором членов — определен метод специализированного и узкого назначения. Такому описанию соответствует виртуальный метод GetHashCode в Object, который существует главным образом для извлечения выгоды следующими двумя типами:

```
System.Collections.Hashtable
System.Collections.Generic.Dictionary< TKey, TValue >
```

Они представляют собой *хеш-таблицы* — коллекции, в которых каждый элемент имеет ключ, используемый для сохранения и извлечения. В хеш-таблице применяется очень специфичная стратегия для эффективного выделения памяти под элементы на основе их ключей. Она требует, чтобы каждый ключ имел число типа Int32, или *хеш-код*. Хеш-код не обязан быть уникальным для каждого ключа, но должен быть насколько возможно разнообразным, чтобы обеспечить хорошую производительность хеш-таблицы. Хеш-таблицы считаются настолько важными, что метод GetHashCode определен в классе System.Object, а потому любой тип может выдавать хеш-код.



Хеш-таблицы детально описаны в главе 7.

Сылочные типы и типы значений имеют стандартные реализации метода `GetHashCode`, т.е. переопределять данный метод не придется, если только не переопределяется метод `Equals`. (И если вы переопределяете метод `GetHashCode`, то почти наверняка захотите также переопределить метод `Equals`.)

Существуют и другие правила, касающиеся переопределения метода `Object.GetHashCode`.

- Он обязан возвращать одно и то же значение на двух объектах, для которых метод `Equals` возвращает `true` (поэтому `GetHashCode` и `Equals` переопределяются вместе).
- Он не должен генерировать исключения.
- Он должен возвращать одно и то же значение при многократных вызовах на том же самом объекте (если только объект не изменился).

Для достижения максимальной производительности в хеш-таблицах метод `GetHashCode` должен быть написан так, чтобы минимизировать вероятность того, что два разных значения получат один и тот же хеш-код. Такое требование порождает третью причину переопределения методов `Equals` и `GetHashCode` в структурах — предоставление алгоритма хеширования, который эффективнее стандартного. Стандартная реализация для структур возлагается на исполняющую среду и может основываться на каждом поле в структуре.

Напротив, стандартная реализация метода `GetHashCode` для классов основана на внутреннем маркере объекта, который уникален для каждого экземпляра в текущей реализации, предлагаемой CLR.



Если хеш-код объекта изменяется после того, как он был добавлен как ключ в словарь, то объект больше не будет доступен в словаре. Ситуацию подобного рода можно предотвратить за счет базирования вычислений хеш-кодов на неизменяемых полях.

Вскоре мы рассмотрим завершенный пример, иллюстрирующий переопределение метода `GetHashCode`.

Переопределение метода `Equals`

Ниже перечислены постулаты, касающиеся метода `Object.Equals`.

- Объект не может быть эквивалентен `null` (если только он не относится к типу, допускающему значение `null`).
- Эквивалентность *рефлексивна* (объект эквивалентен самому себе).
- Эквивалентность *симметрична* (если `a.Equals(b)`, то `b.Equals(a)`).
- Эквивалентность *транзитивна* (если `a.Equals(b)` и `b.Equals(c)`, то `a.Equals(c)`).
- Операции эквивалентности повторяемы и надежны (они не генерируют исключений).

Перегрузка операций == и !=

В дополнение к переопределению метода Equals можно необязательно перегрузить операции == и !=. Так почти всегда поступают для структур, потому что в противном случае операции == и != просто не будут работать для типа.

В случае классов возможны два пути:

- оставить операции == и != незатронутыми — тогда они будут использовать ссылочную эквивалентность;
- перегрузить == и != вместе с Equals.

Первый подход чаще всего применяется в специальных типах — особенно в изменяемых типах. Он гарантирует ожидаемое поведение, заключающееся в том, что операции == и != должны обеспечивать ссылочную эквивалентность со ссылочными типами, и позволяет избежать путаницы у потребителей специальных типов. Пример уже приводился ранее:

```
var sb1 = new StringBuilder ("foo");
var sb2 = new StringBuilder ("foo");
Console.WriteLine (sb1 == sb2);           // False (ссылочная эквивалентность)
Console.WriteLine (sb1.Equals (sb2));    // True (эквивалентность значений)
```

Второй подход имеет смысл использовать с типами, для которых потребителю никогда не понадобится ссылочная эквивалентность. Такие типы обычно неизменяемы — как классы string и System.Uri — и временами являются хорошими кандидатами на реализацию в виде структур.



Хотя возможна перегрузка операции != с приданием ей смысла, отличающегося от !(==), на практике так поступают редко. Примером могут служить типы, определенные в пространстве имен System.Data.SqlTypes, которые представляют собственные типы столбцов в SQL Server. Они соответствуют логике сравнения значений null в базах данных, при которой операции = и <> (== и != в C#) возвращают значение null, если любой из operandов имеет значение null.

Реализация интерфейса IEquatable<T>

Для полноты при переопределении метода Equals также неплохо реализовать интерфейс IEquatable<T>. Его результаты должны всегда соответствовать результатам переопределенного метода Equals класса Object. Реализация IEquatable<T> не требует никаких усилий по программированию, если реализация метода Equals структурирована, как демонстрируется в показанном далее примере.

Пример: структура Area

Предположим, что необходима структура для представления области, ширина и высота которой взаимозаменяемы. Другими словами, 5×10 эквивалентно 10×5 . (Такой тип был бы подходящим в алгоритме упорядочения прямоугольных форм.)

Ниже приведен полный код:

```
public struct Area : IEquatable <Area>
{
    public readonly int Measure1;
    public readonly int Measure2;

    public Area (int m1, int m2)
    {
        Measure1 = Math.Min (m1, m2);
        Measure2 = Math.Max (m1, m2);
    }

    public override bool Equals (object other)
        => other is Area a && Equals (a); // Вызывает метод, определенный ниже
    public bool Equals (Area other) // Реализует IEquatable<Area>
        => Measure1 == other.Measure1 && Measure2 == other.Measure2;
    public override int GetHashCode ()
        => HashCode.Combine (Measure1, Measure2);

    // Обратите внимание, что мы вызываем статический метод Equals в классе
    // object: это выполняет проверку на null перед вызовом нашего метода
    // (экземпляра) Equals.
    public static bool operator == (Area a1, Area a2) => Equals (a1, a2);
    public static bool operator != (Area a1, Area a2) => !(a1 == a2);
}
```



Начиная с версии C# 10, процесс можно сократить с помощью записей. Объявляя это как структуру типа записи, можно избавиться от всего кода, следующего за конструктором.

В реализации метода GetHashCode мы применяли функцию HashCode.Combine из .NET для получения составного хеш-кода. (До появления указанной функции популярный подход предусматривал умножение каждого значения на некоторое простое число и затем их сложение.)

Вот как можно использовать структуру Area:

```
Area a1 = new Area (5, 10);
Area a2 = new Area (10, 5);
Console.WriteLine (a1.Equals (a2)); // True
Console.WriteLine (a1 == a2); // True
```

Подключаемые компараторы эквивалентности

Если необходимо, чтобы тип задействовал другую семантику эквивалентности только в конкретном сценарии, то можно воспользоваться подключаемым компаратором эквивалентности IEqualityComparer. Он особенно удобен в сочетании со стандартными классами коллекций, и мы рассмотрим такие вопросы в разделе “Подключение протоколов эквивалентности и порядка” главы 7.

Сравнение порядка

Помимо определения стандартных протоколов для эквивалентности в C# и .NET также имеются стандартные протоколы для определения порядка, в котором один объект соотносится с другим. Базовые протоколы таковы:

- интерфейсы `IComparable` (`IComparable` и `IComparable<T>`);
- операции `>` и `<`.

Интерфейсы `IComparable` применяются универсальными алгоритмами сортировки. В следующем примере статический метод `Array.Sort` работает из-за того, что класс `System.String` реализует интерфейсы `IComparable`:

```
string[] colors = { "Green", "Red", "Blue" };
Array.Sort (colors);
foreach (string c in colors) Console.Write (c + " ");      // Blue Green Red
```

Операции `<` и `>` более специализированы и предназначены в основном для числовых типов. Поскольку эти операции распознаются статически, они могут транслироваться в высокоэффективный байт-код, подходящий для алгоритмов с интенсивными вычислениями.

В .NET также предлагаются подключаемые протоколы упорядочения через интерфейсы `IComparer`. Мы опишем их в финальном разделе главы 7.

Интерфейсы `IComparable`

Интерфейсы `IComparable` определены следующим образом:

```
public interface IComparable { int CompareTo (object other); }
public interface IComparable<in T> { int CompareTo (T other); }
```

Указанные два интерфейса представляют ту же самую функциональность. Для типов значений обобщенный интерфейс, безопасный к типам, оказывается быстрее, чем необобщенный интерфейс. В обоих случаях метод `CompareTo` работает так, как описано ниже:

- если `a` находится после `b`, то `a.CompareTo(b)` возвращает положительное число;
- если `a` такое же, как и `b`, то `a.CompareTo(b)` возвращает 0;
- если `a` находится перед `b`, то `a.CompareTo(b)` возвращает отрицательное число.

Например:

```
Console.WriteLine ("Beck".CompareTo ("Anne"));           // 1
Console.WriteLine ("Beck".CompareTo ("Beck"));           // 0
Console.WriteLine ("Beck".CompareTo ("Chris"));          // -1
```

Большинство базовых типов реализуют оба интерфейса `IComparable`. Эти интерфейсы также иногда реализуются при написании специальных типов. Вскоре мы рассмотрим пример.

IComparable или Equals

Предположим, что в некотором типе переопределен метод Equals и тип также реализует интерфейсы IComparable. Вы ожидаете, что когда метод Equals возвращает true, то метод CompareTo должен возвращать 0. И будете правы. Но вот в чем загвоздка:

- когда Equals возвращает false, метод CompareTo может возвращать то, что ему заблагорассудится (до тех пор, пока это внутренне непротиворечиво)!

Другими словами, эквивалентность может быть “придирчивее”, чем сравнение, но не наоборот (стоит нарушить правило и алгоритмы сортировки перестанут работать). Таким образом, метод CompareTo может сообщать: “Все объекты равны”, тогда как метод Equals сообщает: “Но некоторые из них ‘более равны’, чем другие”.

Великолепным примером может служить класс System.String. Метод Equals и операция == в String используют *ординальное* сравнение, при котором сравниваются значения кодовых точек Unicode каждого символа. Однако метод CompareTo применяет сравнение, зависящее от культуры, что иногда приводит к помещению более одного символа в одну и ту же позицию сортировки.

В главе 7 мы обсудим подключаемый протокол упорядочения IComparer, который позволяет указывать альтернативный алгоритм упорядочения при сортировке или при создании экземпляра сортированной коллекции. Специальная реализация IComparer может и дальше расширять разрыв между CompareTo и Equals — например, нечувствительный к регистру компаратор строк будет возвращать 0 в качестве результата сравнения “A” и “a”. Тем не менее, обратное правило по-прежнему применимо: метод CompareTo никогда не может быть придирчивее, чем Equals.



При реализации интерфейсов IComparable в специальном типе нарушения данного правила можно избежать, поместив в первую строку метода CompareTo следующий оператор:

```
if (Equals (other)) return 0;
```

После этого можно возвращать то, что нравится, при условии соблюдения согласованности!

Операции > и <

В некоторых типах определены операции > и <, например:

```
bool after2010 = DateTime.Now > new DateTime (2010, 1, 1);
```

Можно ожидать, что операции > и <, когда они реализованы, должны быть функционально согласованными с интерфейсами IComparable. Такая стандартная практика повсеместно соблюдается в .NET.

Также стандартной практикой является реализация интерфейсов IComparable всякий раз, когда операции > и < перегружаются, хотя обратное утверждение неверно. В действительности большинство типов .NET, реализующих