Министерство труда и социальной защиты
Республики Беларусь

Государственное учреждение образования
«Республиканский институт повышения квалификации
и переподготовки работников Министерства труда
и социальной защиты Республики Беларусь»

П. С. Карпович

C# в примерах

Практикум

В двух частях

Часть 1

Минск
Минский государственный ПТК полиграфии
2019

УДК 004(075,9)
ББК 32,973,26-018,2я75
К26

Рекомендовано к изданию Советом института государственного учреждения образования «Республиканский институт повышения квалификациии переподготовки работников Министерства труда и социальной защиты Республики Беларусь», протокол от 31.10.2019 № 4.

А в т о р : *П. С. Карпович*, преп. кафедры информационных технологий РИПК Минтруда и соцзащиты.

Р е ц е н з е н т ы : старший преподаватель кафедры защиты информации учреждения образования «Белорусский государственный университет информатики и радиоэлектроники» *Г. А. Пухир*, программист ООО «Легат Бай» *А. О. Шашков.*

Карпович, П. С.
К26 C# в примерах : практикум. В 2 ч. Ч. 1 / П. С. Карпович. – Минск : Минский государственный ПТК полиграфии, 2019. – 144 с.
ISBN 978-985-7249-05-3.

Практикум «C# в примерах» рекомендовано использовать для слушателей системы повышения квалификации по информационным технологиям, а также при проведении занятий по дисциплинам переподготовки по специальности «Программное обеспечение информационных систем»

УДК 004(075,9)
ББК 32,973,26-018,2я75

ISBN 978-985-7249-05-3 (ч. 1)
ISBN 978-985-7249-07-7

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Практикум «C# в примерах» предназначен для слушателей системы повышения квалификации по информационным технологиям, а также может быть использован при проведении занятий по дисциплинам переподготовки по специальности «Программное обеспечение информационных систем».

Основными задачами практикума являются:

– ознакомление слушателей с синтаксисом и семантикой языка программирования C#;

– описание особенностей архитектуры .NET;

– формирование навыков разработки приложений в рамках парадигмы объектно-ориентированного программирования.

В языке C#, созданном компанией Microsoft для поддержки среды .NET Framework, проверенные временем средства программирования усовершенствованы с помощью самых современных технологий. C# предоставляет очень удобный и эффективный способ написания программ для современной среды вычислительной обработки данных, которая включает операционную систему Windows, Интернет, компоненты и пр.

C# – это язык, разработанный Эндерсом Хейлсбергом в корпорации Microsoft в качестве основной среды разработки для .Net Framework и всех будущих продуктов Microsoft. C# основан на других языках, в основном, C++, Java, Delphi, Modula-2 и Smalltalk. Характеризуя основные особенности языка, отметим, что с одной стороны, для C# в еще большей степени, чем для упомянутых выше языков, характерна внутренняя объектная ориентация; с другой стороны, в нем реализована концепция упрощения объектов, что существенно облегчает освоение мира объектно-ориентированного программирования.

Практикум состоит из двух частей и ряда приложений.

Первая часть практикума содержит шесть глав и пять приложений.

Первая глава первой части представляет собой введение, где разбираются особенности архитектуры .NET, кратко описываются ее основные компоненты.

Во второй главе изложены основы языка C#.

Третья и четвертая главы посвящены принципам объектно-ориетированного программирования, приводится понятие классов и объектов.

В пятой и шестой главах первой части рассматриваются реализация наследования и полиморфизма в языке программирования.

В приложениях представлены примеры реализации программ на C#.

Глава 1. ЛЕГКОЕ ПОГРУЖЕНИЕ В СРЕДУ .NET

Введение

Язык программирования в первую очередь представляет собой инструмент решения определенных задач. И для того, чтобы в полной мере понять язык программирования, следует изучить и осознать, какие проблемы он был призван решать, с какими целями создавался, для чего сотни людей по всему миру трудились много лет, не покладая рук, чтобы его разработать.

Чтобы понять принципы, на которые опирается платформа .NET и язык программирования C#, нужно, как минимум, начать с событий, произошедших за десятилетие до появления этих продуктов. А именно привести историю появления и становления языка программирования Java и всей прилегающей к нему платформы.

Это происходило в лихие девяностые. Все программировали, как могли. Персональные компьютеры только начинали своё внедрение в повседневную жизнь, были широко распространены большие мощные мейнфремы для серьёзных вычислений. Каждый производитель компьютеров делал что-то своё оригинальное и не совместимое со всем остальным. Действовали десятки различных операционных систем и архитектур процессоров, и для каждой из них требовалось создать писать свою версию программы.

Программа, написанная для одной архитектуры, не работала на другой. И с этим ничего нельзя было сделать, только создать требуемую программу заново для новой платформы.

Разработка кроссплатформенных и универсальных программ была очень сложным делом. Если разработчикам было нужно, чтобы их продукт мог использовать значительный круг пользователей, то сложности разработки и поддержки программного продукта могли увеличиваться в разы: одновременно нужно было поддерживать несколько десятков версий одного проекта. Вместо одной программы, разработчики были вынуждены работать сразу с десятком. И добавляя новую функциональность, делать это для каждой версии продукта. Причем эти изменения далеко не всегда можно было вносить единообразно, похожим способом (рис.1.1.).

Это требовало значительных трудозатрат и было кошмаром для кроссплатформенных разработчиков. В результате команда талантливых разработчиков из компании Sun Microsystems решила взять на себя непростую задачу и кардинальным образом изменить принцип построения программ.

После долгих лет работы, в 1995 году, Sun Microsystems выпустили первую версию Java. И это был настоящий прорыв для того времени, инновация, которая навсегда изменила технологию разработки программного обеспечения.

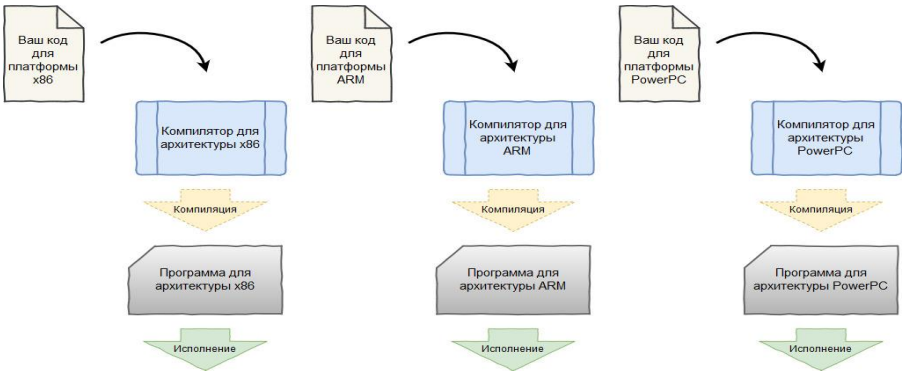

Рисунок 1.1 – Классическая компиляция программы

Концепции

Идея заключалась в том, чтобы обеспечить выполнение программы сразу на всех возможных аппаратных платформах, без необходимости поддерживать несколько версий одного и того же исходного кода. Чтобы единожды написанный код был универсальным и мог запуститься где угодно: на домашних компьютерах, вычислительных мейнфреймах, телефонах, даже на кофемашине или умном холодильнике.

Для этого разработчики из Sun Microsystems придумали дополнительный промежуточный слой между исходным кодом и конечной скомпилированной программой (рис.1.2).

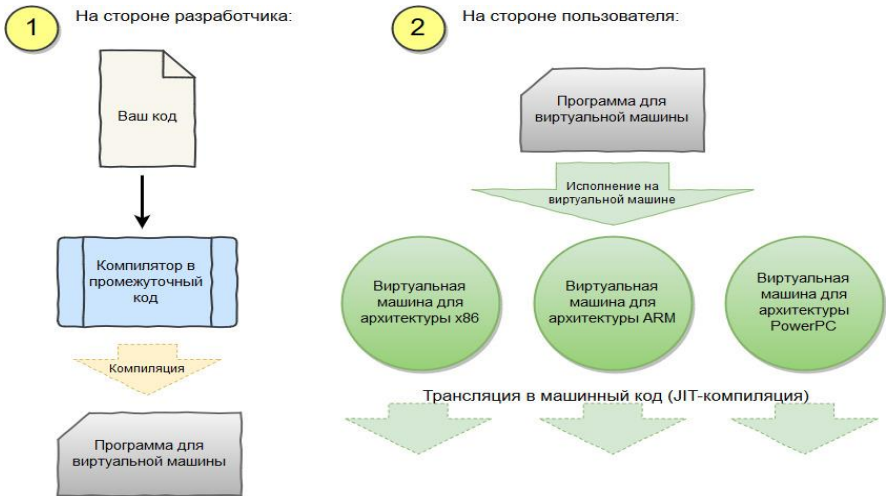

Рисунок 1.2 – Компиляция с виртуальной машиной

Виртуальная машина

Ключевым элементом новой концепции выступает *виртуальная машина*. Под виртуальной машиной здесь подразумевается запущенный фоновый процесс, эмулирующий искусственную аппаратную платформу, не для некоторой реальной архитектуры процессора, а для архитектуры этой искусственной виртуальной машины.

Когда на виртуальной машине запускается какая-либо программа, инструкции из этого промежуточного «как-бы машинного кода» преобразуются к инструкциям для конкретной процессорной архитектуры, которая используется на физической машине, и исполняются.

Такой подход полностью исключает необходимость для разработчика создавать код для нескольких разных платформ – создается всего одна версия кода для виртуальной машины.

Отдельно для каждой поддерживаемой платформы остается написать только саму эту виртуальную машину – и то лишь один раз, а дальше она будет доступна для использования всеми обычными разработчиками.

JIT-компиляция

Just-in-Time компиляция (JIT-компиляция) или динамическая трансляция – это способ выполнения программы, когда машинный код компилируется прямо во время ее работы.

На практике предкомпиляция обычно происходит во время запуска приложения, а во время выполнения осуществляется перекомпиляция выбранных участков кода для их оптимизации.

Преимущества и недостатки

Неоспоримым преимуществом JIT-компиляции является постоянная оптимизация получаемого машинного кода, что может серьезно повысить производительность программы, а также отсутствие необходимости в самостоятельном управлении памятью.

Самый значимый недостаток – длительность запуска программы, т.е. время, необходимое на предварительную компиляцию. Также на компьютере должен быть запущен процесс JIT-компилятора.

Продолжение истории

Язык Java сразу начал набирать огромную популярность и захватывать рынок.

В Microsoft в это время разрабатывался новый набор библиотек для C++.

И, наблюдая за подъемом Java, Microsoft решил вместо разработки для C++ создать собственную платформу и новый язык программирования по аналогии с Java, со своей виртуальной машиной и своим промежуточным языком.

И так появился .NET

Microsoft взяла все основные концепции Java, улучшил и построил вокруг них свою платформу. В 2002 году вышла в свет первая версия .NET Framework.

Однако эта версия не была кроссплатформенной, а работала только под Windows.

.NET

.NET–Framework представляет собой набор библиотек (фреймворк) для создания разносторонних приложений под управлением общеязыковой среды исполнения (CLR).

Все компоненты фреймворка .NET написаны для промежуточного слоя виртуальной машины CLR, что отстраняет их от конкретного языка программирования.

.NETCore

.NETCore – более современная версия .NET Framework, с улучшенным компилятором и переработанной средой CLR.

Основные языки для .NET

Специально для работы с .NET, Microsoft был разработан объектно-ориентированный язык программирования C#, функциональный язык F#, а также адаптирован старичок VisualBasic .NET.

И прочие, и прочие

Поскольку платформа .NET является прослойкой на уровне виртуальной машины, любой язык программирования может полноценно работать с ней. Необходим лишь специальный компилятор в ее промежуточный язык.

И таких реализаций языков программирования для .NET целое множество, начиная с древнейшего COBOL, до популярных Python и Ruby. Имеется даже версия Java от Microsoft для .NET – J#.

С полным списком можно ознакомиться здесь[1].

Каркас платформы

В основе .NET Framework лежат две главные составляющие:

−общеязыковая среда исполнения (CLR);

−библиотека классов .NET(FCL).

CLR

Общеязыковая среда исполнения CLR (CommonLanguageRuntime) является основой платформы .NET.

Это та самая виртуальная машина, которая исполняет промежуточный код (JIT-компиляция), управляет распределением памяти, обеспечивает безопасность типов, обработку исключений и сборку мусора.

В общем, это то, что отвечает за выполнение программ на .NET.

[1] https://en.wikipedia.org/wiki/List_of_CLI_languages

Промежуточный язык

Промежуточный язык, используемый в CLR, описан в спецификации Common Language Infrastructure (CLI)[2].

Чтобы язык программирования был совместим с платформой .NET, он должен компилироваться в этот промежуточный язык.

Управляемый код

Управляемым кодом называют исходный код программы, который будет выполняться в рамках CLR, т.е. код для среды .NET.

Однако в рамках программы для .NET можно использовать и другие вызовы, например, функции операционной системы (Win32 API), которые под управление CLR не подпадают. Такие части кода называются неуправляемым кодом.

Сборка мусора

За счет того, что программы для .NET выполняются внутри другого процесса (CLR), исчезает необходимость в самостоятельном управлении памятью. Этим занимается непосредственно среда CLR.

CLR выполняет такой процесс, как сборка мусора – автоматическое освобождение неиспользуемой памяти.

Она отслеживает ссылки на выделенную память в созданной программе, и, когда они перестают существовать, освобождает ресурсы.

FCL

Библиотека классов FCL (Framework Class Library) – это огромный набор библиотек .NET, включающий масштабную объектно-ориентированную систему типов, компоненты и средства для всевозможных приложений, начиная с консольных программ, и заканчивая корпоративными веб-приложениями.

BCL

В основе всех программных компонентов .NET лежит стандартная система типов, которые представляет собой набор базовых классов.

Этот набор классов называется Базовой библиотекой классов (Base Class Library).

Компоненты FCL / Технологии .NET

Библиотеки FCL разделяются на компоненты, предназначенные для разных технологий в рамках .NET (рис. 1.3):

– ADO.NET;
– ASP.NET;
– WCF;
– WPF.

2 http://www.ecma-international.org/publications/standards/Ecma-335.htm

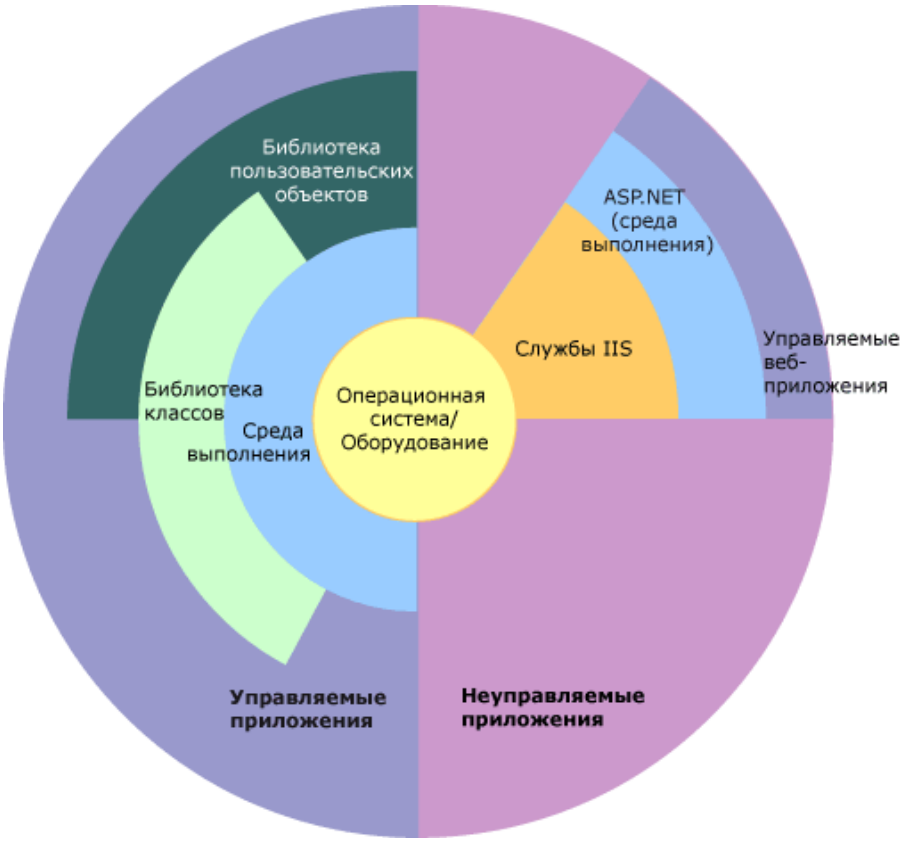

Рисунок 1.3 – Платформа .NET и её окружение

.NETCore

Если ранее .NET был ориентирован только на Windows, то сейчас Microsoft развивает кроссплатформенный .NET Core, а обычный .NET потихоньку устаревает.

Перспективные направления развития .NET

– ML.NET;
– Unity;
– UWP;
– Xamarin.

Глава 2. БЫСТРЫЙ ЭКСКУРС ПО C#

Как отмечалось выше, C# был разработан специально для платформы .NET, и является основным языком для работы с ней.

C# является чисто объектно-ориентированным языком. В объектно-ориентированном программировании (ООП) ход выполнения программы определяется объектами.

Объекты – это экземпляры класса.

Класс – абстрактный тип данных, определяемый пользователем (программистом). Класс может содержать данные (поля) и функции (методы).

```
class имя_класса
{
// члены класса (поля и методы)
}
```

ООП в C#

Все функции в C# должны быть обязательно определены внутри класса.

Точка входа программы в C# Main (тут с прописной буквы) также должна являться статическим методом класса, как правило, класса Program.

```
classProgram
{
staticvoid Main(string[] args)
{
// Вы пишите здесь
    }
}
```

Начало работы

При создании пустого консольного приложения в среде разработки Visual Studio будет сгенерирован вот такой шаблон приложения:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestCon1
{
classProgram
    {
staticvoid Main(string[] args)
{
        }
    }
}
```

Директива using

В начале шаблона идут директивы using, после которых записываются названия подключаемых пространств имен.

Для каждого приложения по умолчанию создается свое уникальное пространство имен, в котором располагаются все создаваемые программистом элементы.

Далее

Однако, прежде чем изучать принципы написания программ, следует углубиться в основы самого языка**.**

Типы данных .NET

Общая структура типов данных, поддерживаемых CLR, описана в спецификации CTS, как общая система типов, иерархическое дерево которой можно увидеть на схеме ниже (рис. 2.1):

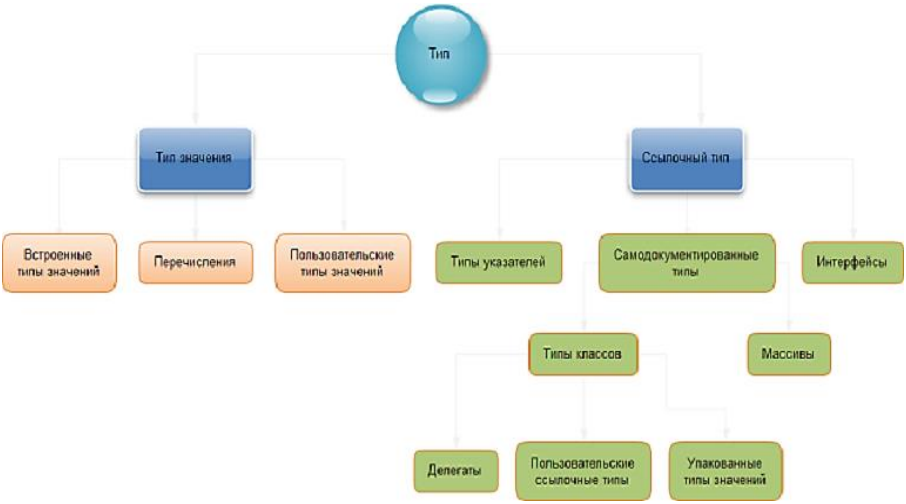

Рисунок 2.1 – Система типов .NET

Таблица 2.1

Типы данных C#

Название	Значения
bool	значение true или false
byte	целое число от 0 до 255 и занимает 1 байт
sbyte	целое число от -128 до 127 и занимает 1 байт
short	целое число от -32768 до 32767 и занимает 2 байта
ushort	целое число от 0 до 65535 и занимает 2 байта
int	целое число от -2147483648 до 2147483647 и занимает 4 байта
uint	целое число от 0 до 4294967295 и занимает 4 байта
long	целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт
ulong	целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт
float	число с плавающей точкой от -3.4*1038 до 3.4*1038 и занимает 4 байта
double	число с плавающей точкой от ±5.0*10-324 до ±1.7*10308 и занимает 8 байта
decimal	16 байтное вещественное число с высокой точностью
char	одиночный символ в кодировке Unicode и занимает 2 байта
string	набор символов Unicode
object	может хранить ссылку на значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе

Типы данных в C#

Все базовые типы данных являются синонимами специальных классов из пространства имен System:System.Byte, System.Int16, System.Int32, System.UInt32, System.Single, System.Double, System.Boolean и т.д.

Объявление переменных

Общий способ объявления переменных имеет C-подобный синтаксис:

```
bool isTrue = true;
int x;
double y = 3.14;
string hello = "Hello Invader!";
char c = 's';
x = 3;
int y = x * 5;
```

Системные типы

Вместо указания ключевых слов для определения типа можно использовать системные типы:

```
int a = 4;
System.Int32 b = 4;
```

Две эти записи будут идентичны.

Неявная типизация

C# относится к языкам со строгой типизацией, но допускается автоматическое присвоение типа через использование ключевого слова var.

```
var str = "string";
var c = 32;
Console.WriteLine(str.GetType().ToString());
Console.WriteLine(c.GetType().ToString());
```

Ограничения var

Во-первых, нельзя объявить неявно типизируемую переменную раздельно с инициализацией.

Во-вторых, нельзя проинициализировать неявно типизированную переменную значением null.

```
var c;          // ошибка 1
c = 144;

var nl = null;      // ошибка 2
```

Ломаем систему, или динамическая типизация

В строго статически типизированном языке программирования есть возможность динамического выведения типов.

Для объявления динамически типизированной переменной используется ключевое слово dynamic вместо указания типа.

Это позволяет использовать одну переменную для хранения разных типов данных.

dynamic

```
dynamic x = 3;          // здесь x - целочисленное int
Console.WriteLine(x);

x = "Привет мир";       // x - строка
Console.WriteLine(x);

x = 354.4f;             // Теперь x - float
Console.WriteLine(x);

Console.ReadLine();
```

dynamic в методах. Настоящий динамит

Динамические переменные можно использовать как параметры и возвращаемые значения в методах (функциях).

```
dynamic GetSalary(dynamic value, string format)
{
if (format == "string")
   {
return value + " рублей";
   }
elseif (format == "int")
   {
return value;
}
else
   {
return 0.0;
   }
}
```

Арифметика

По арифметическим операциям все стандартно:

```
int x = 13;
int z = x + 3;
x = 3 * z - 4;
double y = x / z;
x++;
```

Логическая арифметика

Операторы булевой алгебры также работают по общепринятым правилам:

```
int x = 3;          // 011
int y = 5;      // 101
int and = x & amp; y;    // 1
int or = x | y;     // 111
int xor = x ^ y;    // 110
uint not = ~(uint)x;   // много
int l = x << 1;       // 110
int r = x >> 1;       // 001
```

Преобразования типов

По аналогии с C++ существуют явные и неявные преобразования типов.

```
double d = 143.234;
int c = d;          // неявное приведение типов
int c = (int)d;     // явное
```

Проверка диапазона

Есть ключевое слово checked, которое проверяет диапазоны переполнения, и в случае ошибки генерирует исключение.

```
int a = 53;
int b = 724;
byte c = checked((byte)(a + b));
Console.WriteLine(c);
```

Класс Convert

Для более корректного преобразования типов можно использовать методы класса Convert, которые позволяют приводить практически любые базовые типы данных друг к другу.

```
int x = 34;
string sd = "31.45326";
ulong l = 235;

byte b = Convert.ToByte(l);
double d = Convert.ToDouble(sd);
string s = Convert.ToString(x);
```

Массивы

Написание объявлений массивов имеет свои нюансы: квадратные скобки стоят не после имени переменной, а после типа.

```
int[] mas1;
int[] mas2 = newint[5];
int[] mas3 = newint[5] { 1, 2, 3, 4, 5 };
int[] mas3 = newint[] { 1, 2, 3, 4, 5 };
int[] mas3 = new[] { 1, 2, 3, 4, 5 };
int[] mas3 = { 1, 2, 3, 4, 5 };
```

Многомерные массивы

```
int[,,] cube = newint[2, 3, 4];
int[,] matrix = newint[2, 3] { { 1, 2, 3 }, { -1, -2, -3 } };
```

Массивы массивов

Нужно отличать многомерные массивы от массивов массивов.

Многомерные массивы являются «прямоугольными», а каждый массив внутри массива массивов может иметь свой собственный размер.

```
int[][] mas = newint[3][];
mas[0] = newint[3];
mas[1] = newint[12];
mas[2] = newint[5];
```

Цикл foreach

Помимо стандартных трех циклов for, while, do-while в C# присутствует цикл foreach, разработанный в целях упрощения перебора элементов каких-либо последовательностей.

Цикл имеет следующую форму записи:

```
foreach (тип_данных имя in контейнер)
{
    действия
}
```

Пример

Пример перебора элементов в одномерном и двухмерном массивах:

```
int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int element in arr)
{
    Console.WriteLine(element);
}

int[][] mtr = newint[3][];
mtr[0] =  new[] { 1, 2, 3 };
mtr[1] =  new[] { 4, 5, 6, 7};
mtr[2] =  new[] { 8, 9 };
foreach (int[] array in mtr)
{
foreach (int element in array)
{
        Console.WriteLine(element);
    }
}
```

Методы массива

Массив представляет собой объект агрегирующего класса. Как любой класс, он содержит методы. Их существует огромное количество, некоторые из которых указаны в таблице 2.2.

Таблица 2.2

Основные методы класса Array

Название	Описание
Average()	Высчитывает среднее значение для элементов массива
CopyTo()	Копирует все элементы этого массива в другой
GetType()	Позволяет получить тип текущего массива
GetLength()	Получить количество элементов в массиве
Rank()	Получает количество измерений этого массива
Min()	Выдает минимальное значение элемента массива
Max()	Выдает максимальное значение элемента массива

Программа сортировки массива

```
using System;
namespace SortApp
{
classProgram
    {
staticvoid Main(string[] args)
        {
int[] nums = newint[7];
            Console.WriteLine("Введите семь чисел");
for (int i = 0; i < nums.Length; i++)
            {
                Console.Write("{0}-е число: ", i + 1);
                nums[i] = Int32.Parse(Console.ReadLine());
            }

int temp;
for (int i = 0; i < nums.Length - 1; i++)
            {
for (int j = i + 1; j < nums.Length; j++)
                {
if (nums[i] > nums[j])
                    {
                        temp = nums[i];
                        nums[i] = nums[j];
                        nums[j] = temp;
                    }
                }
            }

            Console.WriteLine("Вывод отсортированного массива");
for (int i = 0; i < nums.Length; i++)
            {
                Console.WriteLine(nums[i]);
            }
            Console.ReadLine();
}
    }
}
```

Строки

В C# строки являются объектами класса String. Содержимое объекта String не подлежит изменению, однако это не сказывается на удобстве его использования.

Конкатенация (объединение) строк представляет из себя тривиальную операцию с оператором +.

```
string str1 = " is ";
string res = "This" + str1 + "string" + "!";
```

Формирование строк

Для создания строк из переменных можно воспользоваться специальным префиксным знаком $:

```
int[] mas = { 1, 2, 3 };
string str = mas[1].ToString();
string st = $"Element {mas[1]} has type: {str} !\n";
Console.Write(st);
```

Использование строк

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
classProgram
    {
staticvoid Main(string[] args)
{
// Сравним первые две строки
string s1 = "это строка";
string s2 = "это текст, а это строка";

if (String.CompareOrdinal(s1, s2) != 0)
                Console.WriteLine("Строки s1 и s2 неравны");

if (String.Compare(s1, 0, s2, 13, 10, true) == 0)
Console.WriteLine("При этом в них есть одинаковый текст");

// Конкатенация строк
            Console.WriteLine(String.Concat("\n" + "Один, два ", "три,
четыре"));

// Поиск в строке
// Первое вхождение подстроки
if (s2.IndexOf("это") != -1)
                Console.WriteLine("Слово \"это\" найдено в строке, оно "
+
"находится на: {0} позиции", s2.IndexOf("это"));

// Последнее вхождение подстроки
if (s2.LastIndexOf("это") != -1)
                Console.WriteLine("Последнее вхождение слова \"это\"
находится "
                    + "на {0} позиции", s2.LastIndexOf("это"));

// Поиск из массива символов
char[] myCh = { 'ы', 'х', 'т' };
if (s2.IndexOfAny(myCh) != -1)
                Console.WriteLine("Один из символов из массива ch " +
```

```
  "найден в текущей строке на позиции {0}", s2.IndexOfAny(myCh));

  // Определяем начинается ли строка с заданной подстроки
  if (s2.StartsWith("это текст") == true)
  Console.WriteLine("Подстрока найдена!");

  // Определяем содержится ли в строке подстрока
  // на примере определения ОС пользователя
  string myOS = Environment.OSVersion.ToString();
  if (myOS.Contains("NT 5.1"))
                  Console.WriteLine("Ваша операционная система Windows
XP");
  elseif (myOS.Contains("NT 6.1"))
  Console.WriteLine("Ваша операционная система Windows 7");

              Console.ReadLine();
          }

      }
  }
```

Комментарии

В C# комментарии бывают трех типов:

– однострочные;

– многострочные;

– XML-документация.

Комментарии для документации

Это специальные однострочные комментарии, начинающиеся с трех слешей (///) вместо двух. В таких комментариях можно указывать XML-дескрипторы, содержащие документацию по написанному коду.

```
  namespace ConsoleApplication1
  {
  ///<summary>
  ///Класс Program
  ///основной класс программы
  ///выводящий текст "Hello world"
  ///</summary>
  classProgram
      {
  ///<summary>
  ///Метод Main() является
  /// входной точкой работы программы
  ///</summary>
  staticvoid Main(string[] args)
  {
  /* содержимое */
          }
      }
  }
```

Параметры функций

Существует два основных способа передачи параметров в метод (функцию):

– по ссылке;

– по значению.

Объекты классов по умолчанию передаются по ссылке, а базовые типы данных и структуры – по значению.

Модификатор ref

При передаче параметров по ссылке перед параметрами используется модификатор ref:

```
staticvoid Main(string[] args)
{
int x = 10;
int y = 15;
    Addition(ref x, y); // вызов метода
    Console.WriteLine(x);

    Console.ReadLine();
}

// определение метода
staticvoid Addition(refint x, int y)
{
    x += y;
}
```

Модификатор out

Модификатор out позволяет через параметры метода возвращать значение. Такое часто применяется, когда требуется вернуть более одного значения различных типов.

```
staticvoid Main(string[] args)
{
int x = 10;

int z;

    Sum(x, 15, out z);

    Console.WriteLine(z);

    Console.ReadLine();
}

staticvoid Sum(int x, int y, outint a)
{
    a = x + y;
}
```

Необязательные и именованные параметры

Необязательные параметры имеют выданное значение по умолчанию.

При вызове метода можно задавать параметры по внутренним именам, что позволит обойти необходимость очередности передачи.

```
staticint OptionalParam(int x, int y, int z = 5, int s = 4)
{
return x + y + z + s;
}

staticvoid Main(string[] args)
{
    OptionalParam(x: 2, y: 3);
//Необязательный параметр z использует значение по умолчанию
OptionalParam(y: 2, x: 3, s: 10);
    Console.ReadLine();
}
```

Переменное число параметров

Используя ключевое слово params в методы можно передавать любое количество параметров заданного типа:

```
staticvoid Addition(paramsint[] integers)
{
int result = 0;
for (int i = 0; i < integers.Length; i++)
    {
        result += integers[i];
    }
    Console.WriteLine(result);
}

staticvoid Main(string[] args)
{
    Addition(1, 2, 3, 4, 5);

int[] array = newint[] { 1, 2, 3, 4 };
    Addition(array);

    Addition();
    Console.ReadLine();
}
```

Указатели и небезопасный код

C# поддерживает работу с указателями и памятью по типу C++, то есть режим работы с памятью напрямую считает небезопасным, и не рекомендуется к использованию без чрезвычайной необходимости.

По умолчанию в Visual Studio запрещено использовать указатели.

Но если хочется неприятностей...

Для того, чтобы разрешить использование небезопасного кода, необходимо зайти в свойства проекта, и на вкладке Построение поставить галочку напротив поля «Разрешить небезопасный код».

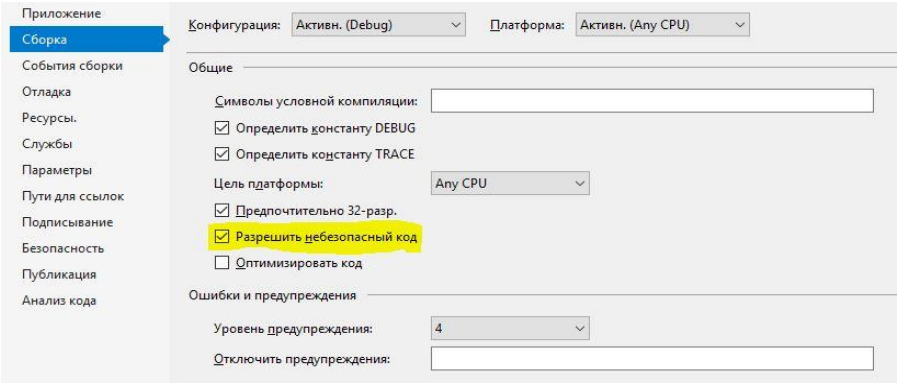

Рисунок 2.2 – Разрешение небезопасного кода

unsafe код

Теперь компилятор разрешит использовать в коде указатели, однако только в местах, помеченных ключевым словом unsafe.

```
staticvoid Main(string[] args)
{
int m = 1;
unsafe
    {
int* pm1 = & m;
        *pm1 += 2;
int* pm2 = pm1;
    }
}

unsafechar* Method(int* ch)
{
char* nc = (char*)ch;
return nc;
}
```

Консоль

Как и все прочее, консоль в среде .NET и конкретно в C# представлена в виде класса Console.

Выше был использован вывод на консоль с помощью метода WriteLine, теперь же рассмотрим этот способ подробнее.

Таблица 2.3

Методы класса Console

Название метода	Описание
Beep()	Подача звукового сигнала
Clear()	Очистка консоли
WriteLine()	Вывод строки текста с автоматическим переходом на следующую строку
Write()	Вывод строки текста, но без символа конца строки
ReadLine()	Считывание строки текста со входного потока
Read()	Считывание введенного символа в виде числового кода данного символа
ReadKey()	Считывание нажатой клавиши клавиатуры (Console KeyInfo key= Console.ReadKey();)

Таблица 2.4

Свойства класса Console

Название свойства	Описание
BackgroundColor	Цвет фона консоли
ForegroundColor	Цвет шрифта консоли
BufferHeight	Высота буфера консоли
BufferWidth	Ширина буфера консоли
Title	Заголовок консоли
WindowHeight	Высота окна консоли
WindowWidth	Ширина окна консоли

Пример работы с консолью

```
classProgram
{
staticvoid Main(string[] args)
    {
// установка зеленого цвета шрифта
        Console.ForegroundColor = ConsoleColor.DarkGreen;

try
        {
do
            {
                Console.WriteLine("Введите первое число");
int num1 = Int32.Parse(Console.ReadLine());
```

```
  Console.WriteLine("Введите второе число");
  int num2 = Int32.Parse(Console.ReadLine());

                  Console.WriteLine("Сумма чисел {0} и {1} равна {2}",
num1, num2, num1 + num2);

  Console.WriteLine("Для выхода нажмите Escape; для продолжения - любую
другую клавишу");
  }
  while (Console.ReadKey().Key != ConsoleKey.Escape);
          }
  catch (Exception ex)
          {
              Console.WriteLine(ex.Message);
  Console.ReadLine();
          }
      }
  }
```

Обработка исключений

Иногда при выполнении программы возникают ошибки, которые трудно предусмотреть или предвидеть, называемые исключениями.

Для обработки таких ситуаций в C# (да и во многих других языках) предназначена конструкция try...catch...finally

try-catch-finally

Блок try охватывает код, в котором предполагается возможность получения исключительной ситуации.

В блоке catch размещаются обработчики исключений. Сюда выполнение программы переходит после исключения в блоке try.

Блок finally предназначен для обязательно выполняющегося кода, как при генерации исключения, так и без него.

Структура обработки исключений

```
try
{
/* код программы */
}
catch
{
/* обработка исключений */
}
finally
{
/* обязательный код */
}
```

Пример

```
staticvoid Main(string[] args)
{
int[] a = newint[4];
try
    {
a[5] = 4; // тут возникнет исключение, так как у нас в массиве только 4
элемента
        Console.WriteLine("Завершение блока try");
    }
catch (Exception ex)
    {
        Console.WriteLine("Ошибка: " + ex.Message);
    }
finally
    {
        Console.WriteLine("Блок finally");
}
    Console.ReadLine();
}
```

Блок catch

Как можно заметить, после ключевого слова catch в скобках записывается тип перехватываемых исключений.

Для разных исключений можно записывать несколько блоков catch.

```
try
{

}
catch (AccessViolationException e)
{
// Обработка исключения, возникшего при отсутствии файла
}
catch (IOException e)
{
// Обработка исключений ввода-вывода
}
catch
{
// Обработка любых других исключений
}
```

Фильтры исключений

Существует возможность обрабатывать исключения в зависимости от каких-то заданных условий при помощи ключевого слова when:

```
int x = 1;
int y = 0;
```

```
try
{
int result = x / y;
}
catch (Exception ex) when (y == 0)
{
    Console.WriteLine("y не должен быть равен 0");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Оператор throw

Чтобы самому прописывать случаи возникновения исключений, используется оператор throw.

```
staticvoid Main(string[] args)
{
try
    {
string message = Console.ReadLine();
if (message.Length > 6)
        {
thrownew Exception("Длина строки больше 6 символов");
        }
    }
catch (Exception e)
    {
        Console.WriteLine("Ошибка: " + e.Message);
}
    Console.ReadLine();
}
```

Механизмы встроенной проверки

Не во всех случаях оптимально применять конструкции try...catch. В классах.NET Framework имеется множество методов для выполнения различных проверок, например метод TryParse() классаInt32.

```
void f1()
{
    Console.WriteLine("Введите число");
int x = Int32.Parse(Console.ReadLine());    // исключение

    x *= x;
    Console.WriteLine("Квадрат числа: " + x);
    Console.Read();
}
```

```
  void f2()
  {
      Console.WriteLine("Введите число");
  int x;
  string input = Console.ReadLine();
  if (Int32.TryParse(input, out x))       // встроенная проверка
исключения
      {
          x *= x;
          Console.WriteLine("Квадрат числа: " + x);
      }
  else
      {
          Console.WriteLine("Некорректный ввод");
      }
      Console.Read();
  }
```

Вложенность обработчиков исключений

Возникшее в программе исключение в обязательном порядке должно быть передано на обработку блоку catch.

Поиск обработчиков выполняется по уровню вызова методов: сначала в локальной функции, если не находится – в функции, откуда вызвана эта функция, и т.д. пока не выйдет за пределы главной функции Main. Если до этого момента не было обнаружено подходящего обработчика, исключение передается реде CLR, которая сама обрабатывает исключение и сообщает пользователю о произошедшей ошибке.

Кортежи (C# 7.0+)

В одной из последних версий язык C# пополнился встроенной возможностью использования кортежей.

Кортеж (tuple) – это просто набор значений.

Кортеж можно считать "неявной структурой" или массивом из элементов разных типов.

Объявление кортежа

Переменная-кортеж определяется через символы круглых скобок:

```
  (int, int) tuple1;
  (int, int) tuple2 = (5, 12);
  tuple1 = tuple2;

  var tuple3 = (1, "Roger Smith", 45);
```

Доступ к элементам кортежа

Для доступа к элементам кортежа по умолчанию используется имена формата ItemN, где N - номер элемента в кортеже.

```
var some = (43, "name", true);
Console.WriteLine(some.Item1);      // 43
Console.WriteLine(some.Item2);      // name
some.Item3 = false;
Console.WriteLine(some.Item3);      // False
```

Кортежи являются строго типизированными, и элемент одного типа нельзя заменить элементом другого типа.

Собственные имена

Вместо ItemN можно назначать элементам кортежа свои наименования, используя нотацию двоеточия.

Причем это работает только при использовании автоматического выведения типов с var.

```
// (string, int, double) person = ("Tom", 25, 81.23);   // Так бы
неработало
var person = (name: "Tom", age: 25, weight: 81.23);
person.age++;
person.weight = 88.3;
person.name = Console.ReadLine();
```

Раскрытие кортежей

Кортеж можно присвоить группе переменных, чтобы получить разделить его значения.

```
var (name, age) = ("Tom", 23);
Console.WriteLine(name);    // Tom
Console.WriteLine(age);     // 23
Console.Read();
```

Применение

Для чего предназначены кортежи?

Их основное применение - возврат сразу нескольких значений из метода (функции).

Кортежи и методы

Как и любой тип, кортежи могут использоваться в методах в качестве параметров и возвращаемых значений.

Например:

```
staticvoid Main(string[] args)
{
var tuple = GetValues();
    Console.WriteLine(tuple.Item1); // 1
    Console.WriteLine(tuple.Item2); // 3

    Console.Read();
```

```
}
privatestatic (int, int) GetValues()
{
var result = (1, 3);
return result;
}
```

Еще один пример

```
staticvoid Main(string[] args)
{
var tuple = GetNamedValues(newint[] { 1, 2, 3, 4, 5, 6, 7 });
    Console.WriteLine(tuple.count);
    Console.WriteLine(tuple.sum);

    Console.Read();
}
privatestatic (int sum, int count) GetNamedValues(int[] numbers)
{
var result = (sum: 0, count: 0);
for (int i = 0; i < numbers.Length; i++)
    {
        result.sum += numbers[i];
        result.count++;
}
return result;
}
```

Камень-Ножницы-Бумага % 3

```
using System;
using System.Collections.Generic;
usingstatic System.Console;

classProgram
{
staticvoid Main(string[] args)
    {
var variants = new List<string>() { "Бумага", "Ножницы", "Камень" };
while (true)
        {
            Write("You: ");
string userChoice = ReadLine();
int userNum = variants.IndexOf(userChoice);

var rand = new Random();
int compNum = rand.Next(3);
            WriteLine("Computer: " + variants[compNum]);

if (userNum == compNum)
            {
                WriteLine("Draw!");
```

```
            }
elseif ((compNum + 1) % 3 == userNum)
            {
                WriteLine("You win!");
            }
else
            {
                WriteLine("Comp win!");
}
        }

    }
}
```

История обновлений

Список всех новых «штук» можно найти здесь:
Официальная документация[3].

Глава 3. ВВЕДЕНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

ООП – подход к написанию программ, когда оперируют не базовыми элементами – числами, строками, символами, а построенными на их основе абстрактными высокоуровневыми сущностями.

Архитектура

Главное в ООП – архитектура, т.е. проектирование кода.

Вы не просто пишите программу как последовательность действий и набор функций, а проектируете элементы системы, объединяете логически связанные компоненты, описываете принципы их взаимодействия.

Моделирование

ООП – это не просто написание программ. Это их моделирование и структурирование.

Например

Пусть требуется разработать сервис для онлайн заказов такси. Вместо того, чтобы писать функции «Свободно Ли Это Такси», «Поиск Ближайшей Машины», «Заказать Такси», «Получить Время До Прибытия» и использовать целый набор глобальных переменных, будет создан набор сущностей «Клиент», «Такси», «МенеджерТакси», «Сервис», которые будут связываться и общаться между собой.

3 https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-6#null-conditional-operators

Принципы ООП

– абстракция;

– инкапсуляция;

– наследование;

– полиморфизм.

Абстракция

Абстракция данных (рис. 3.1) – это принцип описания программных компонентов с привязкой к конкретной предметной области и четким определением их концептуальных границ.

Элементы программы разделяется на уровни абстракции, что позволяет работать с объектами, не вдаваясь в особенности их реализации, и упрощает их использование в прикладном коде.

Рисунок 3.1 – Принципы ООП: абстракция

А по-человечески?

Например.

Вместо того, чтобы обращаться с консолью напрямую через низкоуровневые механизмы ввода-вывода, весь низкоуровневый сложный код спрятан внутри класса Console, который имеет простой и понятный интерфейс для чтения и записи.

Строка, по сути, это массив символов, с которым очень неудобно работать. Поэтому вокруг него построен класс String, который прячет внутри себя всю работу с посимвольным перебором, и предоставляет готовый набор основных методов для работы со строкой.

Или же, допустим, у вас сервис по продаже автомобилей. И чтобы было удобнее работать на уровне целого автомобиля, вы описываете абстракцию машины, внутри которой уже будет содержаться информация о всех его комплектующих: двигателе, коробке передач, электронике, тормозной системе, амортизации и т.д.

Инкапсуляция

Инкапсуляцией (рис. 3.2) называется объединение логически связанных данных и функций для их обработки.

Также в C# под инкапсуляцией понимается механизм, позволяющий ограничить доступ одних компонентов программы к другим.

Рисунок 3.2 – Принципы ООП: инкапсуляция

А по-человечески?

Все функции для работы со строкой находятся внутри строки String.

Все функции для работы с массивом находятся внутри класса массива Array.

Все функции для работы с датой находятся внутри класса даты DateTime.

Вне зоны доступа

Сущность может состоять из элементов, доступ к которым не должен получить никто кроме самой этой сущности.

Например, хоть строка и состоит из набора символов, но перезаписать значения этих символов при работе со строкой нельзя, эта операция скрыта.

А теперь представим, что вы создаете систему для учета книг в библиотеке. У каждой книги есть набор характеристик: название, автор,

ISBN-номер, состояние. Эти характеристики должны быть объединены в одну сущность. Причем все перечисленные свойства, кроме состояния, должны быть скрыты от изменения извне.

Например, имеется база книг библиотеки, в которой хранится информация обо всех подотчетных книгах. Для каждой книги должна храниться история держателей, и данные текущего носителя. Также должен быть функционал назначения нового держателя, или удаление и перенос текущего в историю. Все эти данные и функции могут быть размещены вместе в одной сущности База Книг.

Наследование

Наследованием (рис. 3.3.) называется возможность абстрактной сущности перенимать данные и функциональность некоторой другой сущности, способствуя повторному использованию программного кода.

Так, начиная с самых абстрактных понятий и сущностей, может формироваться иерархия классов, переходя от полной абстракции к частным случаям реализации.

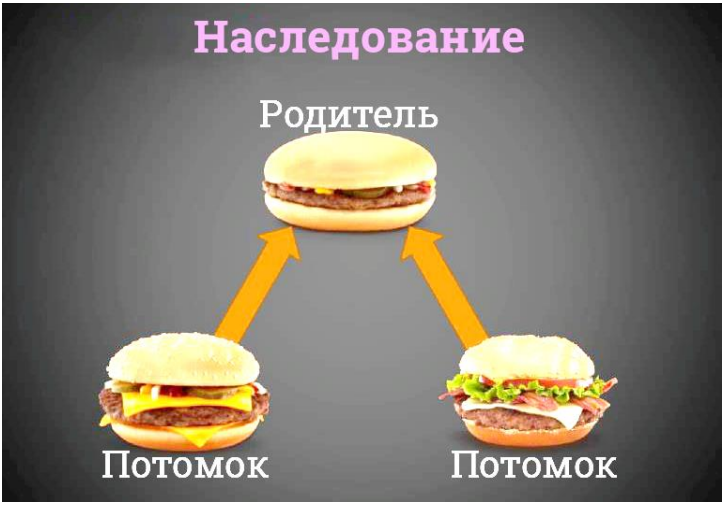

Рисунок 3.3 – Принципы ООП: наследование

А по-человечески?

Можно создавать классы по шаблону других классов, т.е. чтобы они сразу были не пустыми, а уже содержали некую начальную логику, от которой начиналась выстраиваться их собственная.

Например, базовый класс фигуры, у которого будут свойства координат, масштаба и вращения.

И от этого класса будут наследоваться более конкретные классы прямоугольника, эллипса и параллелограмма, у которых останутся их «свойства по наследству», а также добавятся свои собственные.

Библиотека BCL

Все классы из всех библиотек .NET составляют единую гигантскую иерархию классов.

Базовым классом для всего является класс Object.

Например, все простые типы данных – int, char, double и т.д. (структуры), наследуются от класса ValueType, который уже наследуется от класса Object.

А классы String и Array наследуются от Object напрямую.

Теперь пример. Представим, что вы разрабатываете программу для формирования отчетности по сертификации оборудования на промышленном предприятии. У вас в коде фигурируют классы «Сертификат соответствия ГОСТ», «Декларация соответствия» и «Сертификат на серийный выпуск». Все эти сущности имеют некоторый набор одинаковых характеристик, поэтому логично выделить их общую часть в отдельный класс «Базовый сертификат2, от которого будут наследоваться все три, и добавлять к основным характеристикам что-то сугубо свое.

Полиморфизм – способность функции обрабатывать данные разных типов (рис. 3.4.), имеющих одинаковый интерфейс (базовый класс).

Рисунок 3.4 – Принципы ООП: полиморфизм

А если по-человечески?

Если функция принимает объект типа «Музыкальный инструмент», то она может принять и объект типа «Фортепиано», и объект типа «Труба». Потому что все эти типы являются потоками класса Музыкальный инструмент.

Функция, которая работает с потоком вывода, может работать с потоком вывода на консоль, с потоком вывода в файл, а также с любым другим потоком вывода – с каким угодно классом, который является производным от базового потока вывода.

Например, у нас есть приложение интернет-магазина смешанной продукции. Пусть здесь будут продаваться корма для животных, косметика и канцтовары. Вместо того, чтобы писать методы "Купить Корм", "Купить Косметику" и "Купить Канцтовары" следует определить всего один метод, который будет работать с базовым классом всех наших товаров: "Купить Товар".

На вырост

Изучать следующие темы необходимо только зрелым ООП программистам.

S.O.L.I.D.

Принципы объектно-ориентированного дизайна систем.

Пять принципов правильного дизайна программ[4].

Паттерны проектирования

Устоявшиеся и общепризнанные архитектурные конструкции, разрешающие часто возникающие проблемы проектирования сложных систем.

Книга: Head First. Design Pattern[5][EN] [RU].

Хорошие разъяснения на YouTube[6]: Ссылка [EN].

Сайт с кратким описанием всех патернов[7]: Ссылка [EN].

4 https://habr.com/ru/company/mailru/blog/412699/

5 https://www.oreilly.com/library/view/head-first-design/0596007124/

6 https://www.youtube.com/watch?list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc&v=v9ejT8FO-7I

7 https://sourcemaking.com/

Глава 4. КЛАССЫ И ОБЪЕКТЫ

Объектно-ориентированное программирование

Что важно понять для работы с ООП, – это то, что все типы являются классами.

Ваша программа описана в классе программы, для работы с консолью используется класс консоли, для работы с окнами – класс окна, для работы с лохматым померанским шпицем – класс собачки.

Классы и объекты

Класс – это абстрактный тип данных, определяемый программистом.

Объекты – это экземпляры класса, т.е. переменные типа класса.

Члены класса – это данные и функции для работы с этими данными.

Имя класса можно считать именем нового типа данных.

```
// Объявление класса
classИмяКласса
{
// члены класса
}
```

Класс и его экземпляры

Класс можно считать "чертежом", "схемой" какого-либо объекта. Класс сам по себе является просто описанием какой-либо сущности.

Объект (экземпляр) же является фактическим представлением этой сущности.

Например, Boeing777 – класс, представляющий одноименный самолет. Но это еще не какой-либо конкретный самолет этой модели. Можно создать экземпляры этого класса boeing1, boeing2 и boeing3, которые уже будут представляться реальные воплощения, три разных самолета этой модели.

Создание экземпляра класса

Создание объекта класса осуществляется с помощью оператора new:

```
ИмяКласса имяОбъекта = new ИмяКласса();
```

Например:

```
Random rand = new Random();
DateTime dt = new DateTime(1991, 10, 23);
string[] mas = newstring[4];
List<int> list = new List<int>();
```

КлассDog

```
// Описание класса Dog
classDog
{
```

```
// Поля этого класса
publicstring name;
publicstring breed;
publicint age;
publicstring owner;

// Открытыйметод
publicvoid Walk()
    {
        Console.WriteLine("The {0} walks!", this.name);
        Random randGen = new Random();
int rndNumber = randGen.Next(4);
if (rndNumber == 3)
        {
this.Pee();
        }
    }

// Скрытыйметод
privateint Pee()
    {
        Console.WriteLine("The {0} has urinated..", this.name);
        Random randGen = new Random();
int strenth = randGen.Next(1, 11);
return strenth;
    }
}

staticvoid Main(string[] args)
{
// Создание экземпляра класса Dog с именем dg1
    Dog dg1 = new Dog();
// Присваивание значений полям
    dg1.name = "Альфред";
    dg1.breed = "Бернский зенненхунд";

// Создание массива под 4 объекта класса Dog
    Dog[] flock = new Dog[4];

// Во второй элемент массива записываем ранее созданный объект
    flock[2] = dg1;

// Вызываем для него метод Walk
    flock[2].Walk();
}
```

Ключевое слово this

Для доступа к членам текущего экземпляра класса можно использовать ключевое слово this. Это ключевое слово предоставляет ссылку на текущий объект класса.

В принципе, использовать this не обязательно, это бывает нужно при совпадении имен. Например:

```
  // Допишем новый метод ChangeOwner
  // который принимает параметр: строку owner
  publicbool ChangeOwner(string owner)
  {
  // Сравнивается поле owner в текущем (this) объекте класса со значением
параметра owner
  if (this.owner == owner)
      {
  returnfalse;
  }
  // В поле записывается значение из параметра
  this.owner = owner;
  returntrue;
  }
```

Спецификаторы доступа

Уровнем доступности каждого элемента класса можно управлять.

Это делается с помощью спецификаторов доступа:

– public – общедоступный член класса;

– private – член класса доступен только внутри данного класса;

– protected – член класса доступен только внутри данного класса и его производных классов;

– internal – член класса доступен только внутри данной сборки (~программы).

КлассBook

```
  // Описание класса Book
  classBook
  {
  // Открытые поля
  publicstring name;
  publicstring author;
  publicint year;

  // Закрытые поля
  protectedint id;
  privatebool isAllowed;

  // Метод
  publicvoid CreateEmptyBook()
  {
  // Всем полям задаются какие-то значения
  this.name = "неизвестно";
  this.author = "неизвестно";
  this.year = 0;
  this.id = 0x8FFA;
  this.isAllowed = false;
      }
```

```
  // Метод
  publicvoid FillBook(string name, string author, int year)
  {
  // Заполнение полей через параметры
  this.name = name;
  this.author = author;
  this.year = year;
  this.id = 0x031AE;
  this.isAllowed = true;
      }

  // Метод
  publicvoid GetInformation()
      {
  if (this.name == null)
  {
  // В методе вызывается другой метод
  this.CreateEmptyBook();
          }
          Console.WriteLine($"Книга '{name}' (автор {author}) была издана
в {year} году");
      }
  }
```

Содержимое класса

Все, что может быть внутри класса подразделяется на:

– элементы данных;

– функциональные элементы;

– вложенные типы.

Элементы данных в классах

Элементы данных подразделяются на:

– поля;

– константы;

– события.

Поле – основной член класса – переменная внутри класса, содержащая некоторое значение.

Пример

Можно, например, объявить класс, членами которого будут являться только поля:

```
  // Просто класс с полями
  classSimpleClass
  {
  publicint x;
  publicfloat y;
  publicconstdouble z = 0.9134;
  publicconstchar c = 'A';
  publicstring s;
  }
```

Константы

Константы – это неизменяемые поля. Объявляются с помощью ключевого слова const.

```
classCalendar1
{
// Константное поле
publicconstint months = 12;
}
```

Константа не может быть объявлена без указания значения.

Поля только для чтения

Readonly поля – похожие на константы поля, с той лишь особенностью, что значение им может быть присвоено позже (не сразу). Но после инициализации менять их, как и константы, нельзя. Объявляются с помощью ключевого слова readonly.

```
classAge
{
// Поле только для чтения
readonlyint year;

// Метод
void Initialize(int year)
{
// Записываем значение в поле только для чтения
// Будет работать только один раз
this.year = year;
    }

}
```

Функциональные элементы в классах

Функциональные элементы подразделяются на:

– методы;

– свойства;

– конструкторы;

– финализаторы;

– индексаторы;

– операторы.

Методы

Основой любого класса являются методы – функции внутри класса, выполняющие некоторые действия над полями этого класса.

Например, здесь описаны два метода: Sum и Sub:

```
// Класс
classCalculator
{
// Закрытые поля класса
privateint x;
privateint y;

// Метод для подсчёта суммы значений этих полей
publicint Sum()
    {
returnthis.x + this.y;
}

// Метод для подсчёта разности значекний этих полей
publicint Sub()
    {
returnthis.x - this.y;
}
}
```

Статические члены класса

Обычные методы и вообще все, описанное внутри класса, существует в контексте какого-либо объекта этого класса. Т.е. для каждого экземпляра класса действуют свои копии методов и полей.

Можно создать элементы класса, которые будут относиться непосредственно к самому классу, а не к объектам. Такие элементы класса называется статическими, являются общими для всех объектов и существуют в единственном экземпляре.

Чтобы объявить такой элемент, нужно к его описанию добавить ключевое слово static.

Статические методы

Например, как происходит работа с консолью в C#?

Что-то наподобие этого:

```
Console.WriteLine("message");
string input1 = Console.ReadLine();
Console.Write("end;\n");
```

Обратите внимание, здесь нигде не создается объект класса Console, мы напрямую обращаемся с именем класса.

Именно так используются статические методы.

Пример

Посмотрим на различия между вызовом статических и обыкновенных методов:

```
// Класс
classMinMax
{
// Статический метод
```

```
publicstaticint Min(int x, int y)
{
int z = (x < y) ? x : y; // Тернарный (условный) оператор
return z;
    }

// Обычный метод
publicint Max(int x, int y)
    {
int z = (x > y) ? x : y;
return z;
    }
}

classProgrma
{
staticvoid Main(string[] args)
    {
int a = -2;
int b = 23;
        Console.WriteLine("a={0}, b={1}", a, b);
// Вызов статического метода
// для класса MinMax
int k = MinMax.Min(a, b);
        Console.WriteLine("Min = {0}", k);

        MinMax instance = new MinMax();
// Вызов обычного метода
// для объекта instance класса MinMax
        instance.Max(a, b);
    }
}
```

Область применения статики

Статические методы применяются повсеместно, и используются в тех случаях, когда нет необходимости создавать объект для работы с этим методом.

Статическими, например, могут быть какие-нибудь константы внутри класса, общие для всех объектов. Или методы, никак не затрагивающие всей основной структуры класса и работающие обособленно.

Свойства

Для создания большей гибкости и безопасности внутренних полей, были разработаны такие элементы класса как Свойства.

Во многих языках программирования свойствами называют обычные поля класса, но в C# есть разграничение между этими понятиями. Фактически они очень схожи между собой и имеют границу только для разработчика класса.

Свойства в C#

Свойством в C# называются специальные методы доступа (чтения/записи) к полю класса (get и set).

Описание свойства имеет следующий синтаксис:

```
// Закрытое поле
privateint name;

// Открытое свойство для этого поля
publicint Name
{
// Геттер - чтобы получить значение
get
    {
returnthis.name;
    }
// Сеттер - чтобы присвоить значение
set
    {
this.name = value; // value - специальное ключевое слово, описывающее
присваемое значение
    }
}
```

Описание свойства

Свойство состоит из двух частей:

– метод-аксессор (геттер) get;

– метод-аксессор (сеттер) set.

Метод get обязательно должен возвращать значение.

В методе set существует неявный параметр value, в котором хранится присваиваемое значение.

Работа со свойствами

Поле и связанное с ним свойство – разные элементы, поэтому должны иметь разные имена.

Общепринято называть Свойство так же как называется Поле, только с заглавной буквы.

Работа со свойствами аналогична работе с обычными полями.

```
// Класс
classTimePeriod
{
// Закрытое поле
privatedouble minutes;

// Свойство для него
publicdouble Minutes
    {
```

```
get { returnthis.minutes; }
set { this.minutes = value; }
}

// Ещё одно закрытое поле
privatedouble hours;

// И для второго поля свойство
publicdouble Hours
    {
get { returnthis.hours; }
set { this.hours = value; }
    }
}
```

Предназначение

Поля в классе принято всегда объявлять приватными, т.е. скрытыми.

А для того, чтобы предоставить доступ к полю, используется Свойство – публичный элемент, через который пользователь читает (get) или записывает (set) значения в приватное поле.

Т.е. свойство является посредником между пользователем и данными, и когда требуется обращаться к полю, всегда следует делать это через свойство.

В свойствах можно описывать необходимую логику или форматирование для значений.

Доступ только для чтения/записи

Один из аксессоров может быть опущен.

Если написать Свойство только с методом get, то это свойство будет только для чтения: из него можно будет получить значение, а записать нельзя.

И менее распространенный вариант, только с методом set свойство будет использоваться только для записи: в него можно записывать значение, но нельзя получить его.

Только чтение

Такой вариант свойства применяется довольно часто:

```
// Класс Person
publicclassPerson
{
// Закрытые поля
privatestring firstName;
privatestring lastName;
privateint age;

// Метод для инициализации значений полей
publicvoid CreatePerson(string first, string last, int age)
    {
```

```
this.firstName = first;
this.lastName = last;
this.age = age;
    }

// Свойство только для получения значения
// причём выдаваемое и хранимые значения различаются
publicstring Name
    {
get
        {
returnthis.firstName + " " + this.lastName;
}
    }

// Обычное свойство
publicint Age
    {
get
        {
returnthis.age;
        }
    }
}
```

Примеры

```
// Класс SomeClass
classSomeClass
{
// Закрытое поле и открытое свойство для него
privateint mode;
publicint Mode
    {
get { returnthis.mode; }
set { this.mode = value; }
}

// Закрытое поле и открытое свойство для него
privatestring name;
publicstring Name
    {
get { returnthis.name; }
set { this.name = value; }
}
}
```

Примеры

Вместо того, чтобы опускать аксессор, можно описать его как приватный:

```
  // Класс Person
  classPerson
  {
  // Ничего не обычного
  privatestring name;
  publicstring Name
      {
  get { return name; }
  set { name = value; }
      }

  // Закрытое поле
  privatechar gender;
  // Открытое свойство
  publicchar Gender
      {
  // Обычный геттер
  get { return gender; }

  // Закрытый сеттер - доступен только внутри класса
  privateset { gender = value; }
      }

  private DateTime birth;
  // Свойство вовсе без сеттера
  public DateTime Birth
      {
  get { return birth; }
      }

  // Снаружи между этими двумя вариантами разницы не будет никакой - ни
там, ни там нет сеттера.
  // Но внутри, в первом случае, сеттер будет работать, а во втором - нет
  }
```

Свойства со встроенной проверкой

При необходимости, внутри свойства можно задавать некоторые проверки, или в принципе написать любой нужный код.

```
  // Ещё один класс
  classTryPerson
  {
  // Закрытое поле
  privatestring name;
  // Открытое свойство для него
  publicstring Name
      {
  // Геттер с логикой
  get
          {
  // Выдаёт не просто значение из поля, но с припиской "Name is "
  string res = "Name is " + this.name;
```

```
  return res;
          }
  // Сеттер с логикой
  set
  {
  // Записывает значение только если оно не равно пустой строке или
нулевой ссылке
  if (value != ""&& value != null)
                  name = value;
          }
      }

  privatechar gender;
  publicchar Gender
      {
  get { return gender; }
  // Закрытый сеттер с логикой
  privateset
          {
  // Разрешает записывать только значения 'м' и 'ж'
  if (value == 'м' || value == 'ж')
  gender = value;
          }
      }

  private DateTime birth;
  public DateTime Birth
      {
  get { return birth; }
  // Безсеттера
      }
  }
```

Примеры

```
  // Да сколько можно
  publicclassDate
  {
  // Закрытое поле с начальным значением (вообще, так лучше не делать)
  privateint month = 7;

  // Открытое свойство для этого поля
  publicint Month
      {
  // Стандартный геттер
  get
          {
  returnthis.month;
          }
  // Сеттер с логикой
  set
          {
```

```
// Не пропускает значения вне диапазона номеров месяцев
if ((value < 0) && (value < 13))
            {
this.month = value;
}
        }
    }
}
```

Примеры

Свойство (как и все остальное) может быть статическим. Тогда оно может ссылаться только на статические поля:

```
// Остановитесь, классы!
publicclassEmployee
{
// Открытое статическое поле
publicstaticint NumberOfEmployees;
// Закрытое статическое поле
privatestaticint counter;

// Закрытое обычное поле
privatestring name;

// Свойство для него
publicstring Name
    {
get { return name; }
set { name = value; }
}

// Статическое свойство для статического класса
publicstaticint Counter
{
get { return counter; }
// Без сеттера
    }

// Конструктор(?)
publicEmployee()
    {
// Увеличение счётчика рабочих
counter = ++NumberOfEmployees;
    }
}

classTestEmployee
{
staticvoid Main()
{
// Записываем значение в открытое статическое поле класса Employee
Employee.NumberOfEmployees = 107;
```

```
// Создаём объект класса Employee
Employee e1 = new Employee();
// Записываем значение в его свойство
        e1.Name = "Claude Vige";

// Выводим значение из статического свойства класса
System.Console.WriteLine("Employee number: {0}", Employee.Counter);
// и из свойства объекта
        System.Console.WriteLine("Employee name: {0}", e1.Name);
}
}
```

Конструкторы

Конструктор – особый метод, вызываемый при создании экземпляра класса. Он называется так же, как класс, и не имеет типа возвращаемого значения.

В классе всегда должен быть хотя бы один конструктор (если его нет – он неявно создается сам).

Выделяют 4 основных типа конструкторов:

– конструктор по умолчанию;

– конструктор с аргументами;

– конструктор копирования;

– статический конструктор.

Конструктор по умолчанию

Конструктор, объявленный без аргументов, называется конструктором по умолчанию и должен присутствовать в любом классе.

Если в классе не указать ни одного конструктора, компилятор сам создаст конструктор по умолчанию на этапе компиляции.

Пример класса с конструктором

```
// Класс без конструктора (явного)
classA
{
// С открытыми полями
publicint x, y, z;
}

// Класс с конструктором
classB
{
// С открытыми полями
publicint x, y, z;

// Конструктор по умолчанию - без параметров
public B()
    {
```

```
  // Инициализация полей
          x = 3;
          y = 4;
          z = 5;
  }
  }

  classProgram
  {
  publicstaticvoid Main(string[] args)
  {
  // При создании объекта через new вызывается конструктор
  // Для класса A вызывается автоматически созданный конструктор по
умолчанию, x = y = z = 0
          A a1 = new A();
  // Для класса B вызывается написанный нами конструктор по умолчанию, x =
3, y = 4, z = 5
          B b1 = new B();
      }
  }
```

Конструкторы с аргументами

Конструкторы могут принимать различные аргументы, с помощью которых объекты классов инициализируются различными способами.

Пример использования конструкторов с аргументами

```
  // Класс D
  classD
  {
  // Открытые поля
  publicint x, y, z;

  // Конструктор по умолчанию
  public D()
      {
  // Инициализация всех полей
          x = y = z = 0;
      }

  // Конструктор с двумя параметрами
  public D(int x, int y)
      {
  // Инициализация всех полей
  this.x = x;
  this.y = y;
  this.z = x + y;
  }

  // Конструктор с тремя параметрами
  public D(int x, int y, int z)
      {
```

```
  // Инициализация всех полей
  this.x = x;
  this.y = y;
  this.z = z;
      }
  }

  classProgram
  {
  publicstaticvoid Main(string[] args)
  {
  // Создаём объект класса D, используя конструктор по умолчанию, x = y =
z = 0
          D did = new D();
  // Создаём объект класса D, используя конструктор с 2 параметрами, x =
5, y = 8, z = 13
          D opo = new D(5, 8);
  // Создаём объект класса D, используя конструктор с 3 параметрами, x =
7, y = 2, z = 6
          D fer = new D(7, 2, 6);
      }
  }
```

Конструктор копирования

Конструктор копирования принимает в качестве параметра экземпляр своего класса, и нужен затем, чтобы создать текущий объект как копию переданного.

```
  // Очередной класс
  classClassC
  {
  // Три открытых поля
  publicint x, y, z;

  // Конструктор по умолчанию
  public ClassC()
      {
  // Инициализирует все поля значением -1
          x = y = z = -1;
      }

  // Конструктор копирования
  public ClassC(ClassC obj_c)
      {
  // Копирует все данные из объекта, переданного параметром obj_c, в
текущий объект this
  this.x = obj_c.x;
  this.y = obj_c.y;
  this.z = obj_c.z;
  }
  }
```

Применение конструктора копирования

Необходимость в конструкторе копирования обуславливается тем, что класс является объектом ссылочного типа, а это означает, что после операции присваивания между двумя объектами класса произойдет копирование ссылки на один и тот же объект, а не создание объекта-копии.

```
  // Создаём объект класса ClassC obj1
  ClassC obj1 = new ClassC();
  // Здесь не создаётся новый объект obj2
  ClassC obj2 = obj1;
  // obj1 и obj2 указывают на один и тот же объект класса

  obj2.x = -10;       // obj1.x == -10

  // А вот здесь создаётся новый объект - копия объекта obj2
  ClassC obj3 = new C(obj2);
  // obj3 имеет такие же значения полей как и obj2, но является другим
объектом

  obj3.x = 2000;      // obj2.x == -10
```

Статический конструктор

Статический конструктор используется для инициализации статических элементов класса, и не может иметь параметров. Он неявно вызывается средой CLR, до момента первого использования класса.

```
  // Класс
  classSimpleClass
  {
  // Статическое поле только для чтения
  staticreadonlylong baseline;

  // Статический конструктор
  static SimpleClass()
  {
  // Который инициализирует статическое поле
         baseline = DateTime.Now.Ticks;
     }
  }
```

Пример

```
  // Класс
  publicclassBus
  {
  // Статическая переменная, используемая всеми экземплярами класса Bus
  // Представляет время, когда первый за день автобус начал свой маршрут
  protectedstatic DateTime globalStartTime;
```

```
  // Свойство для номера маршрута каждого автобуса
  // Сокращённая форма записи
  protectedint RouteNumber { get; set; }

  // Статический конструктор инициализирует статическое поле
  // Это происходит до создания первого экземпляра класса
  static Bus()
      {
          globalStartTime = DateTime.Now;

  Console.WriteLine("Статический конструктор установил время старта на
{0}",
  globalStartTime.ToLongTimeString());
      }

  // Конструкторэкземпляракласса
  public Bus(int routeNum)
      {
          RouteNumber = routeNum;
          Console.WriteLine("Автобус #{0} отправляетсявпуть.",
RouteNumber);
      }

  // Метод экземпляра класса
  publicvoid Drive()
      {
          TimeSpan elapsedTime = DateTime.Now - globalStartTime;

  // Симуляция времени поездки автобуса
          Console.WriteLine("Автобус {0} начал своё движение через {1:N2}
минут после общего времени старта {2}.",
  this.RouteNumber,
              elapsedTime.Milliseconds,
              globalStartTime.ToShortTimeString()
  );
      }
  }

  classTestBus
  {
  staticvoid Main()
  {
  // Создание экземпляра предварительно активирует статический
конструктор.
  Bus bus1 = new Bus(71);

  // Создание второго маршрута
          Bus bus2 = new Bus(72);

  // Автобус 1 отправляется в путь
          bus1.Drive();

  // Делаем паузу перед отправкой второго автобуса
```

```
        System.Threading.Thread.Sleep(25);

// Автобус 2 отправляется
        bus2.Drive();

        System.Console.WriteLine("Нажмите любую клавишу для выхода...");
        System.Console.ReadKey();
    }
}
```

Рекомендации

Для любого класса желательно всегда определять конструктор по умолчанию и конструктор копирования, так как они являются универсальными и могут требоваться некоторыми встроенными алгоритмами.

Финализатор

Финализатор – специальный метод, который вызывается перед тем, как Сборщик Мусора соберется удалить ваш объект. Он должен называться так же, как и сам класс, только перед именем должен стоять знак тильды ~.

```
classCar
{
// Финализатор
    ~Car()
    {
// Освобождаем ресурсы...
// Прощаемся с объектом...
    }
}
```

Пример работы финализатора

```
// Класс
classFinalizeObject
{
// Свойство и поле в сокращённой форме записи
publicint Id { get; set; }

// Открытое статическое поле
publicstaticint identity;

// Статический конструктор
static FinalizeObject()
    {
// Присваиваем начальное значение статическому полю
        identity = 0;
    }

// Конструктор по умолчанию
public FinalizeObject()
    {
```

```
  // Присваиваем полю текущего объекта значение из статического поля
  this.Id = FinalizeObject.identity++;
  // Затем увеличиваем значение в статическом поле на 1
  // Таким образом, Id каждого последующего создающегося объекта будет на
1 больше,
  // чем у предыдущего созданного объекта
      }

  // Финализатор
      ~FinalizeObject()
      {
  // Перед удалением объекта из памяти на консоль выведется его Id
          Console.WriteLine("Объект №{0} уничтожен", Id);
  }
  }

  classProgram
  {
  staticvoid Main(string[] args)
  {
  // Создаётся массив для 100 экземпляров класса FinalizeObject
  FinalizeObject[] obj = new FinalizeObject[100];
  // По циклу создаём эти 100 объектов
  for (int i = 0; i < 100; i++)
              obj[i] = new FinalizeObject();

  // Main заканчивается, и все переменные, созданные в нём, удаляются
  // Т.е. удаляются и наши экземпляры класса
      }
  }
```

Индексаторы

Индексаторы позволяют обращаться с объектом класса так, как будто он является массивом, т.е. применять оператор квадратных скобок.

Индексатор по синтаксису напоминает свойство. Он также состоит из методов-аксессоров get и set, только в дополнение принимает параметр – индекс.

```
  // Пример индексатора
  public тип_возвращаемого_значения this[тип_индекса j] // Принимает
"параметр" - индекс
  {
  get
      {
  // Действия с return
      }
  set
      {
  // Установка значения с value
      }
  }
```

Индексатор в классе

Обычно индексаторы используются в классах, которые содержат массив или другую коллекцию, для простого доступа к их элементам.

```
  // Класс
  classMyArray
  {
  // Приватное поле - массив байт
  privatebyte[] innerArr = newbyte[100];

  // Индексатор к текущему классу, для доступа к элементам закрытого
массива
  publicbytethis[int i]
  {
  // Геттер, возвращающий элемент массива по переданному индексу i
  get
          {
  return innerArr[i];
          }
  // Сеттер, записывающий значение в элемент массива по индексу i
  set
          {
              innerArr[i] = value;
          }
      }
  }

  classProgram
  {
  staticvoid Main()
      {
  // Создаём экземпляр класса MyArray
  var arr = new MyArray();

  // Записываем значение 255 по индексу 0 во внутреннее поле-массив класса
через индексатор
          arr[0] = 255;

  // Обращаемся к нулевому элементу массив через индексатор
          Console.WriteLine(arr[0]);
      }
  }
```

Несколько индексаторов

Как и методы, индексаторы могут быть перегружены. Т.е. в одном классе может быть сразу несколько индексаторов, различающихся типами индексов и их количеством.

```
  // Класс Human
  classHuman
  {
  // Закрытые поля
```

```
  protectedstring name;
  protectedstring surname;
  protectedstring patronymic;

  // Индексатор с типом параметра string
  publicstringthis[string s]
  {
  // Возвращает значение поля класса по переданному имени поля
  get
          {
  switch (s)
              {
  case"name": returnthis.name;
  case"surname": returnthis.surname;
  case"patronymic": returnthis.patronymic;
  default: return"none";
              }
          }
  // Записывает значение в поле, по переданному имени
  set
          {
  switch (s)
              {
  case"name": this.name = value; break;
  case"surname": this.surname = value; break;
  case"patronymic": this.patronymic = value; break;
  }
          }
      }

  // Индексатор для текущего класса с типом параметра int
  publicstringthis[int n]
  {
  // Каждое поле класса ассоциированно с индексом-числом
  // В зависимости от числа в индексе, возвращаем значение из какого-либо
поля
  get
          {
  switch (n)
              {
  case 0: returnthis.name;
  case 1: returnthis.surname;
  case 2: returnthis.patronymic;
  default: return"none";
              }
          }
  // То же самое, только с присваиванием значения
  set
          {
  switch (n)
              {
  case 0: this.name = value; break;
  case 1: this.surname = value; break;
  case 2: this.patronymic = value; break;
```

```
            }
        }
    }
}

publicclassBlaBlaClass
{
publicstaticvoid Main(string[] args)
{
// Создание объекта класса Human
        Human slave1 = new Human();
// Обращение к объекту через индексатор, принимающий строку
        slave1["name"] = "Gregore";
// Записываем значения в поля, обращаясь к ним через квадратные скобки
slave1["surname"] = "O'Cammel";
        slave1["patronymic"] = "What does it mean?";

// Или так, используя второй индексатор, который принимает индексы -
числа
        slave1[2] = "Silence!";

// Можно перебрать свойства через цикл
for (int i = 0; i < 3; ++i)
        {
            Console.WriteLine($"Field #{i,2}: {slave1[i]}");
}
    }
}
```

Индексаторы с несколькими индексами

Индексатор может принимать не только один индекс, но и сразу несколько, как при использовании многомерных массивов.

```
// Класс ячейки шахматного поля
classChessBoardCell
{
// ...

}
// Класс шахматной доски
classChessBoard
{
// Закрытое поле для хранения информации о клетках поля
// Двумерный массив 8x8 из объектов класса CheshBoardCell
private ChessBoardCell[,] _matrix;

public ChessBoard(bool fill = true)
{
        _matrix = new ChessBoardCell[8, 8];
    }

// Индексатор с двумя индексами - символом и числом
```

```
  public ChessBoardCell this[char row, int column]
  {
  // Из символа получаем число, соответствующее индексу во внутренней
матрице
  // И по полученному числу и второму индексу возвращаем значение
  get
          {
  if ((int)row >= (int)'A'&& (int)row <= (int)'h')
              {
  return _matrix[(char)((int)row - (int)'A'), column];
              }
  elseif ((int)row >= (int)'a'&& (int)row <= (int)'h')
              {
  return _matrix[(char)((int)row - (int)'a'), column];
  }
  else
              {
  returnnull;
              }
          }
  // То же самое, только с присваиванием значения
  set
          {
  if ((int)row >= (int)'A'&& (int)row <= (int)'h')
              {
                  _matrix[(char)((int)row - (int)'A'), column] = value;
              }
  elseif ((int)row >= (int)'a'&& (int)row <= (int)'h')
              {
                  _matrix[(char)((int)row - (int)'a'), column] = value;
  }
          }
      }
  }

  // Перечисление игровых сторон
  enumChessSide
  {
      WHITE,
      BLACK
  };

  // Перечисление видов шахматных фигур
  enumChessPieceType
  {
      KING,
      QUEEN,
      BISHOP,
      KNIGHT,
      ROOK,
      PAWN
  }
```

```
  // Класс для шахматной фигуры
  classChessPiece
  {
  // ...

  public ChessSide Side { get; privateset; }
  public ChessPieceType Piece { get; privateset; }

  public ChessPiece(ChessSide side, ChessPieceType type)
      {
          Side = side;
          Piece = type;
      }

  }

  classProgram
  {
  publicstaticvoid Main(string[] args)
      {
  // Создаём объект класс ChessBoard
          ChessBoard board = new ChessBoard(true);

  // Через индексатор по двум индексам обращаемся к определённой ячейке
массива и записываем в неё значение
  board['E', 1] = new ChessPiece(ChessSide.WHITE, ChessPieceType.ROOK);
          board['E', 2] = new ChessPiece(ChessSide.WHITE,
ChessPieceType.PAWN);

  // Получаем значение через индексатор по двум индексам
          ChessPiece myKnight = board['G', 5];
      }
  }
```

Глава 5. НАСЛЕДОВАНИЕ КЛАССОВ

Наследование

Наследование (рис. 5.1) – одна из главных отличительных особенностей ООП.

Наследованием называется механизм получения нового класса на основе уже существующего.

Терминология

Класс, от которого наследуется класс, называется базовым, или родительским.

Новый класс, создаваемый на основе базового класса, называется производным классом.

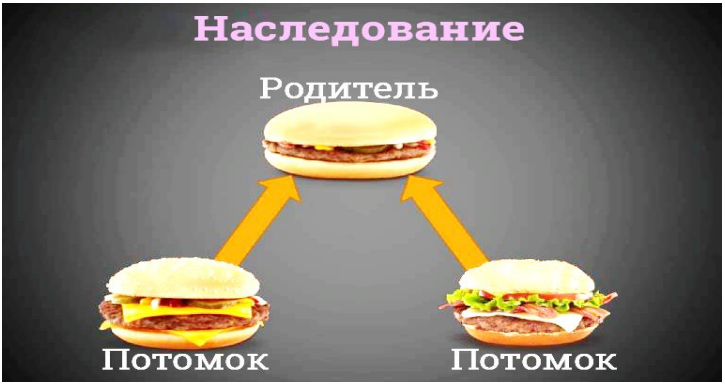

Рисунок 5.1 – Наследование

Суть наследования

Производный класс перенимает всю функциональность (свойства/методы) базового класса, которая по необходимости может быть переопределена.

В производный класс могут быть добавлены любые новые члены, как в любой другой класс. Таким образом, производные классы позволяют расширять функциональность базовых классов, без их изменения.

Предназначение наследования

Главное назначение механизма наследования – повторное использование кода.

А также полиморфизм подтипов... но об этом далее…

Виды наследования

В ООП наследование бывает одиночным и множественным.

– одиночное наследование – когда у производного класса может быть только один базовый класс;

– множественное наследование – когда один класс может быть производным сразу от нескольких других классов.

В C# разрешено только одиночное наследование.

Опасность множественного наследования

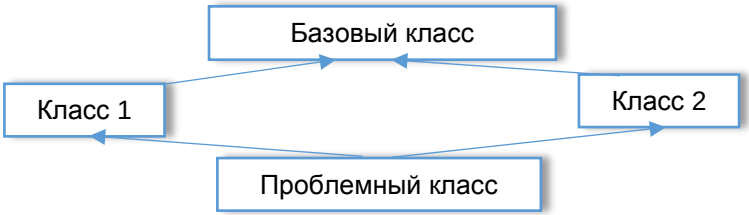

Рисунок 5.2 – Смертельный алмаз смерти

С ним лучше не связываться!

Наследование в C#

Еще раз, в C# класс может наследоваться только от одного базового класса (рис. 5.3).

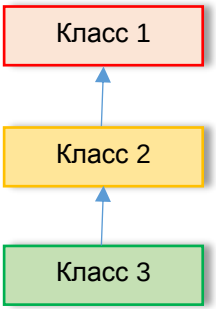

Рисунок 5.3 – Одиночное наследование

Свойство транзитивности

Наследование транзитивно. Это значит, что любые производные классы от производного же класса также являются производными от него. И наоборот, все классы по иерархии наследования выше текущего класса являются для него базовыми.

Т.е. если класс A есть потомок класса B, а B есть потомок класса C, то A также является потомком C.

Принципы наследования

Наследование обычно предполагает уточнение некоторой сущности.

Самый первый класс в иерархии наследования является самым абстрактным и обобщенным, а последующие все более и более уточняют поведение и подходят к частным случаям реализации, см. рис. 5.4 и рис. 5.5.

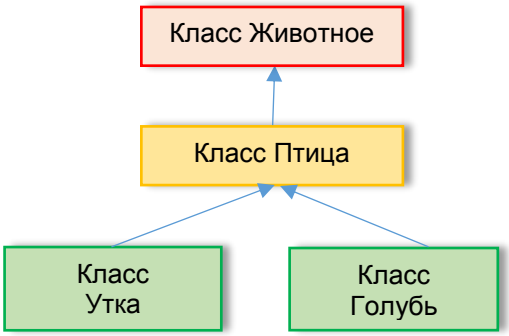

Рисунок 5.4 – Пример: иерархия наследования птиц

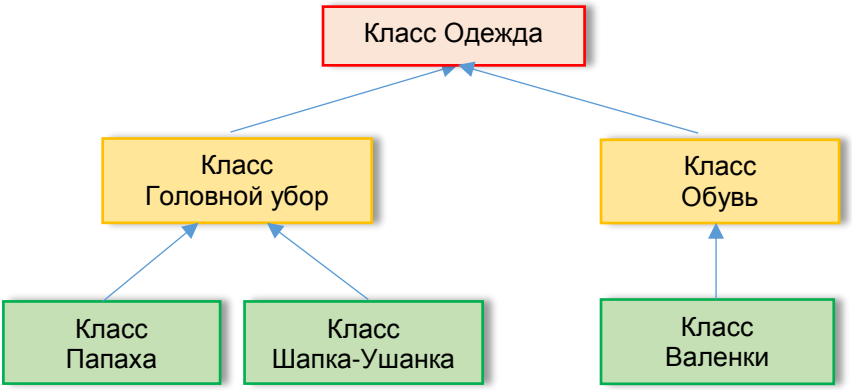

Рисунок 5.5 – Пример: иерархия наследования одежды

Создание производного класса

Для того чтобы объявить, что класс является производным от какого-то другого класса, нужно при его описании после имени класса через двоеточие (:) написать имя родительского класса.

```
// Некоторый класс
class SomeClass
{
// ...
}

// Класс DerivedClass - производный от класса SomeClass
class DerivedClass : SomeClass
{
// ...
}
```

Пример

```
// Класс собаки
classDog
{
// Поля этого класса
publicstring Name { get; set; }
publicstring Breed { get; set; }
publicint Age { get; set; }
publicstring Owner { get; set; }

// Открытый метод
publicvoid Walk()
    {
        Console.WriteLine("The {0} walks!", Name);
        Random randGen = new Random();
```

```
int rndNumber = randGen.Next(4);
if (rndNumber == 3)
        {
this.Pee();
        }
    }

// Закрытый метод
privateint Pee()
    {
        Console.WriteLine("The {0} has urinated..", Name);
        Random randGen = new Random();
int strenth = randGen.Next(1, 11);
return strenth;
    }
}

// Класс собаки-ищейки, производный от класса собаки
class PoliceDog : Dog
{

}
```

Пример

```
classProgram
{
publicstaticvoid Main(string[] args)
    {
// Создаём объект класса Dog
        Dog dog1 = new Dog();
// И используем его свойства и методы
dog1.Name = "Matilda";
        dog1.Walk();

// Создаём объект класса PoliceDog
        PoliceDog dog2 = new PoliceDog();
// И используем свойства и методы, доставшиеся ему по наследству
dog2.Name = "Rufus";
        dog2.Age = 9;
dog2.Walk();
    }
}
```

Особенности наследования

Производный класс наследует всю функциональность базового класса, но не всей ею может пользоваться.

Private члены базового класса не доступны в производных.

Если требуется описать закрытый элемент класса, который должен быть виден и потомкам, используется спецификатор доступа protected (рис. 5.6).

Спецификаторы доступа / Область видимости	public	protected	private
В самом классе	Доступны	Доступны	Доступны
В производных классах	Доступны	Доступны	Не доступны
Из других классов	Доступны	Не доступны	Не доступны

Рисунок 5.6 – Особенности спецификаторов доступа

Пример

```
classGadget
{
// Открытое свойство
publicstring Name { get; set; }

// Закрытое поле
privateint gadgetId;

// Защищённый метод
protectedvoid UpdateNameVersion()
{
// Дописываем к имени окончание
        Name += "ver. 2.0";
    }

}

// Производный класс от гаджета
classSmartphone : Gadget
{
// Конструктор по умолчанию
public Smartphone()
    {
```

```
    // Обращаемся к разным элементам базового класса
    Name = "Vivo Nex 7";
            gadgetId = 2;    // Недоступно
            UpdateNameVersion();

        }

    }

    classProgram
    {
    staticvoid Main(string[] args)
        {
    // Создаём объект класса смартфон
            Smartphone sm = new Smartphone();
    // Обращаемся к разным членам класса
            sm.Name = "New Name";
            sm.gadgetId = 2;            // Не доступно
            sm.UpdateNameVersion();     // Не доступно
        }
    }
```

Добавление новых членов

В производном классе можно описывать любые допустимые для классов конструкции.

Т.е. производный класс состоит из элементов родителя и собственных членов (рис. 5.7).

Рисунок 5.7 – Представление наследования через диаграммы Канта

Потомок класса собаки

```
  // Класс собаки-ищейки, производный от класса собаки
  classPoliceDog : Dog
  {
  // Новые свойства
  publicint Rank { get; set; }
  publicstring Specialization { get; set; }

  // Новые конструкторы, методы и т.д.
  // Всё, что мы пишем в этом классе, дополняет функциональность от
базового класса
  public PoliceDog(string name, int age, string breed, string owner, int
rank, string spec)
      {
          Name = name;
          Age = age;
          Breed = breed;
          Owner = owner;
          Rank = rank;
          Specialization = spec;
      }

  publicint UpRank()
      {
          ++Rank;
  return Rank;
      }

  publicvoid Train()
      {
          Walk();
          Console.WriteLine("The dog {0} goes to the hard training",
Name);
  }
  }
```

Переопределение методов

Иногда необходимо изменить поведение методов, унаследованных от базового класса.

Замена функциональности метода базового класса на другую называется переопределением метода.

Виртуальные методы

Нельзя просто взять и переопределить любой унаследованный метод.

Методы, которые разрешается переопределять, должны быть помечены в базовом классе ключевым словом virtual.

Такие методы называются виртуальными методами.

Если метод виртуальный, это не значит, что он **должен быть** переопределен. Это значит, что он может быть переопределен.

Переопределенные методы

Чтобы переопределить виртуальный метод, нужно написать в производном классе метод такой же сигнатуры (с таким же именем, спецификатором доступа, типом возвращаемого значения и набором параметров) и пометить его ключевым словом override.

Пример иерархии классов с переопределенными методами можно найти в Приложении А.

Обращаемся к методу родителя

Но что, если нам не нужно полностью переписывать метод, а требуется только дополнить его?

Даже если метод переопределен, все равно остается возможность обратиться к первоначальной версии метода.

Ключевое слово base

С помощью ключевого слова base можно обращаться к содержимому базового класса. Получается, что внутри класса есть два специальных ключевых слова this и base.

Слово this используется для доступа к элементам текущего класса (включая унаследованные элементы).

Слово base используется для доступа к оригинальным элементам базового класса.

Пример

Модифицируем метод FindSmth, чтобы он выполнял оригинальный метод, а потом наш дополнительный код.

```
  // ...
  publicoverridestring FindSmth(int distance)
  {
  // Вызываем оригинальный метод FindSmth и сохраняем строку, которую он
вернул
  string found = base.FindSmth(distance);
  // А дальше то же самое, что и было
  var gen = new Random(100);
  int rand = gen.Next();
  // При определённом стечении обстоятельств
  if (distance > 59 && rand > 98)
      {
  // Собака находит взрывчатку
         found = "explosive!";
  }
  return found;
  }
  // ...

  publicstaticvoid Main(string[] args)
  {
  // Создаём объект класса собаки-ищейки
  var dog2 = new PoliceDog("Arnold", 12, "Rottweiler", "Sgt. Bobbey", 2,
"explosive");
```

```
  // И смотрим на разницу
  dog2.FindSmth(100);     // "bone"
      dog2.FindSmth(100);     // "bone"
      dog2.FindSmth(100);     // "tennis ball"
      dog2.FindSmth(100);     // "stick"
      dog2.FindSmth(100);     // "rabbit"
      dog2.FindSmth(100);     // "explosive!"
  }
```

Наследование и конструкторы

Во время создания объекта класса, происходит последовательный вызов конструкторов всех базовых классов от начала иерархии.

В случае с классом собаки:

```
  // Здесь сначала вызывается конструктор по умолчанию класса Object
  // Затем конструктор по умолчанию класса Dog
  // И последним конструктор класса PoliceDog, в который мы передаём
параметры
  var dog2 = new PoliceDog("Arnold", 12, "Rottweiler", "Sgt. Bobbey", 2,
"explosive");
```

Погодите-ка, появился какой-то еще класс Object?

Ремарка. Класс Object

В .NET любой класс, у которого не задан родительский класс, неявно наследуется от класса Object (или типа object).

Класс Object находится в основе иерархии всех классов .NET. Любой класс в .NET является потомком этого класса.

Классы Math, Console, Program, Int32, String, Array, Dog, – все они являются производными от класса Object.

Обычные методы класса Object описаны в таблице 5.1, а статические методы – в таблице 5.2.

Таблица 5.1

Методы класса Object

Название метода	Описание
Equals(Object)	Проверяет, равен ли указанный объект текущему объекту
Finalize()	Финализатор, выполняется перед удалением объекта
GetHashCode()	Используется как хеш-функция по умолчанию
GetType()	Возвращает тип текущего объекта
ToString()	Преобразовывает текущий объект в строку
MemberwiseClone()	Создает поверхностную копию текущего объекта

Таблица 5.2

Статические методы класса Object

Название метода	Описание
Equals(Object, Object)	Проверяет два объекта на равенство, используя обычный метод Object
ReferenceEquals(Object, Object)	Сравнивает ссылки объектов на равенство

Переопределение методов класса Object

Сами по себе эти методы имеют мало практической пользы, но некоторые из них являются виртуальными, а значит – их можно переопределить.

Виртуальные методы Object:

– Equals(Object);

– Finalize();

– GetHashCode();

– ToString().

Метод ToString

Рассмотрим метод ToString, как самый показательный.

Методы ToString используется при неявном приведении объекта класса к типу строки.

Например, когда вы пишите:

```
  Dog myDog = new Dog();

  // Здесь объект класса Dog преобразовывается с помощью метода ToString в
строку для вывода на консоль
  Console.WriteLine(myDog);
  // И здесь
  string str = "Dog: " + myDog;
```

Объект любого класса может быть преобразован к строке. Потому что у любого класса есть этот метод – ToString.

Базовая реализация ToString

Другое дело, что стандартная реализация этого метода преобразует к строке не слишком-то и хорошо.

```
  Dog myDog = new Dog();
  Console.WriteLine(myDog);   // TestConsoleApp1.Dog

  int[] arr = { 1, 2, 3, 4 };
  Console.WriteLine(arr);     // System.Int32[]
```

По умолчанию, базовая реализация метода ToString, которая описана в классе Object, возвращает строку с полным именем типа текущего объекта.

ПереопределениеToString

Переопределив в классе метод ToString, можно написать свою реализацию преобразования объекта класса в строку. Например

```
  classDog
  {
  publicstring Name { get; set; }
  publicstring Breed { get; set; }
  publicint Age { get; set; }
  publicstring Owner { get; set; }

  publicvirtualvoid Walk()
      {
          Console.WriteLine("The {0} walks!", Name);
          Random randGen = new Random();
  int rndNumber = randGen.Next(4);
  if (rndNumber == 3)
          {
  this.Pee();
          }
      }

  privateint Pee()
      {
          Console.WriteLine("The {0} has urinated..", Name);
          Random randGen = new Random();
  int strenth = randGen.Next(1, 11);
  return strenth;
      }

  // Добавляем переопределённый метод ToString базовогокласса Object
  publicoverridestring ToString()
      {
  // Возвращаем собственный вариант строки
  return$"Dog named {Name}, with owner {Owner}, {Breed}, {Age} years";
      }
  }

  classProgram
  {
  staticvoid Main(string[] args)
  {
  // Создаём объект класса Dog
          Dog dog1 = new Dog();
          dog1.Name = "Mikkey";
          dog1.Breed = "American cocker spaniel";
          dog1.Age = 2;
          dog1.Owner = "Mr. Garrison";

  // Создаём объект класса PoliceDog
          PoliceDog dog2 = new PoliceDog("Ralf", 8, "Indian pariah dog",
"Officer Mc'Cry", 3, "Explosive");
```

```
  // Неявное преобразование к строке, вызов метода ToString, который нами
переопределён
  Console.WriteLine(dog1);
  // Вывод: Dog named Mikkey, with owner Mr. Garrison, American cocker
spaniel, 2 years

  // Класс PoliceDog является производным от класса Dog
  // Поэтому он наследует переопределённую версию метода ToString
  Console.WriteLine(dog2);
  // Вывод: Dog named Ralf, with owner Officer Mc'Cry, Indian pariah dog,
8 years

  }
  }
```

Переопределяем переопределенный метод

Но у класса PoliceDog есть два дополнительных свойства, которые тоже необходимо выводить на консоль. Для этого в данном классе можно также переопределить переопределенный метод ToString.

```
  classDog
  {
  publicstring Name { get; set; }
  publicstring Breed { get; set; }
  publicint Age { get; set; }
  publicstring Owner { get; set; }

  publicvirtualvoid Walk()
      {
          Console.WriteLine("The {0} walks!", Name);
          Random randGen = new Random();
  int rndNumber = randGen.Next(4);
  if (rndNumber == 3)
          {
  this.Pee();
          }
      }

  privateint Pee()
      {
          Console.WriteLine("The {0} has urinated..", Name);
          Random randGen = new Random();
  int strenth = randGen.Next(1, 11);
  return strenth;
      }

  // Переопределённый метод в классе Dog
  publicoverridestring ToString()
      {
  // Возвращаем собственный вариант строки
```

```
  return$"Dog named {Name}, with owner {Owner}, {Breed}, {Age} years";
      }
  }

  classPoliceDog : Dog
  {
  publicint Rank { get; set; }
  publicstring Specialization { get; set; }

  public PoliceDog(string name, int age, string breed, string owner, int
rank, string spec)
      {
          Name = name;
          Age = age;
          Breed = breed;
          Owner = owner;
          Rank = rank;
          Specialization = spec;
      }

  publicint UpRank()
      {
          ++Rank;
  return Rank;
      }

  publicvoid Train()
      {
  base.Walk();
          Console.WriteLine("The dog {0} goes to the hard training",
Name);
      }

  publicoverridevoid Walk()
      {
          Console.WriteLine("The police dog {0} walks!", Name);
      }

  // Переопределяем метод Dog-а ToString
  publicoverridestring ToString()
  {
  // Возвращаем строку, сформированную из результата выполнения метода
ToString базового класса
  // И дополнительной информации от текущего класса
  return"Police " + base.ToString() + $". With rank {Rank} and
specializing in {Specialization}";
      }

  }

  classProgram
  {
  staticvoid Main(string[] args)
```

```
      {
  // Создаёмобъекткласса Dog
          Dog dog1 = new Dog();
          dog1.Name = "Mikkey";
          dog1.Breed = "American cocker spaniel";
          dog1.Age = 2;
          dog1.Owner = "Mr. Garrison";

  // Создаёмобъекткласса PoliceDog
          PoliceDog dog2 = new PoliceDog("Ralf", 8, "Indian pariah dog",
"Officer Mc'Cry", 3, "Explosive");

  // Вызов ToString класса Dog
          Console.WriteLine(dog1);
  // Вывод: Dog named Mikkey, with owner Mr. Garrison, American cocker
spaniel, 2 years

  // Вызов ToString класса PoliceDog, который внутри вызывает ToString
класса Dog
  Console.WriteLine(dog2);
  // Вывод: Police Dog named Ralf, with owner Officer Mc'Cry, Indian
pariah dog, 8 years. With rank 3 and specializing in Explosive

      }
  }
```

Наследование и конструкторы

Вернемся к нашим-конструкторам.

При вызове конструктора производного класса перед ним неявно вызываются конструкторы по умолчанию всех базовых классов.

Но часто бывает необходимо выполнять не конструктор по умолчанию в базовых классах, а конструктор с параметрами.

В конструкторе производного класса можно указывать, какой конструктор базового класса вызвать перед ним.

Иерархия классов музыкальных инструментов

Проблема:

```
  // Класс музыкального инструмента
  classMusicalInstrument
  {
  // Свойства
  publicstring Name { get; set; }
  publicstring SoundRange { get; set; }
  publicstring Timbre { get; set; }
  publicint UniqueId { get; privateset; }

  // Закрытыйметод
  privateint GetRandomId()
      {
          Random randGen = new Random();
```

```
  int a = randGen.Next(1000000);
  int b = randGen.Next(1000);
  return a % b + 4321 - b / 2;
      }

  // Конструкторпоумолчанию
  public MusicalInstrument()
      {
          Name = "Unnamed";
          SoundRange = "0";
          Timbre = "void";
          UniqueId = 0;
      }

  // Конструктор с параметрами
  public MusicalInstrument(string name, string soundRange, string timbre)
      {
          Name = name;
          SoundRange = soundRange;
          Timbre = timbre;
  UniqueId = GetRandomId();
      }
  }

  // Класс струнного музыкального инструмента
  classStringInstrument : MusicalInstrument
  {
  // Свойства
  publicstring Type { get; set; }
  publicint StringCount { get; set; }
  publicint ScaleLength { get; set; }

  // Конструктор по умолчанию
  public StringInstrument()
      {
          Name = "Unnamed";
          SoundRange = "0";
          Timbre = "void";
          Type = "None";
          StringCount = 0;
          ScaleLength = 0;

      }

  // Конструктор с параметрами
  public StringInstrument(string name, string soundRange, string timbre,
string type, int stringCount, int scaleLength)
      {
          Name = name;
          SoundRange = soundRange;
          Timbre = timbre;
          Type = type;
          StringCount = stringCount;
```

```
            ScaleLength = scaleLength;
        }

    }

    // Класс фортепиано
    classPiano : StringInstrument
    {
    // Свойства
    publicint KeyCount { get; set; }
    publicint PedalsCount { get; set; }

    }
```

Использование конструкторов базового класса

```
    // Класс музыкального инструмента
    classMusicalInstrument
    {
    // Свойства
    publicstring Name { get; set; }
    publicstring SoundRange { get; set; }
    publicstring Timbre { get; set; }
    publicint UniqueId { get; privateset; }

    // Закрытый метод
    privateint GetRandomId()
        {
            Random randGen = new Random();
    int a = randGen.Next(1000000);
    int b = randGen.Next(1000);
    return a % b + 4321 - b / 2;
        }

    // Конструктор по умолчанию
    public MusicalInstrument()
        {
            Name = "Unnamed";
            SoundRange = "0";
            Timbre = "void";
            UniqueId = 0;
        }

    // Конструктор с параметрами
    public MusicalInstrument(string name, string soundRange, string timbre)
        {
            Name = name;
            SoundRange = soundRange;
            Timbre = timbre;
    UniqueId = GetRandomId();
        }
    }
```

```
  // Класс струнного музыкального инструмента
  classStringInstrument : MusicalInstrument
  {
  // Свойства
  publicstring Type { get; set; }
  publicint StringCount { get; set; }
  publicint ScaleLength { get; set; }

  // Конструктор по умолчанию
  public StringInstrument() : base("Unnamed", "0", "void")
  {
          Type = "None";
          StringCount = 0;
          ScaleLength = 0;

      }

  // Конструктор с параметрами, вызывающий конструктор по умолчанию
базового класса
  public StringInstrument(string type, int stringCount, int scaleLength)
          : base("Unnamed", "0", "void")
      {
          Type = type;
          StringCount = stringCount;
          ScaleLength = scaleLength;
      }

  // Конструктор с параметрами
  public StringInstrument(string name, string soundRange, string timbre,
string type, int stringCount, int scaleLength)
          : base(name, soundRange, timbre)
      {
          Type = type;
          StringCount = stringCount;
          ScaleLength = scaleLength;
      }

  }

  // Класс фортепиано
  classPiano : StringInstrument
  {
  // Свойства
  publicint KeyCount { get; set; }
  publicint PedalsCount { get; set; }

  // Конструктор по умолчанию, вызывающий конструктор по умолчанию
базового класса
  public Piano() : base()
      {
          KeyCount = 0;
          PedalsCount = 0;
  }
```

```
  // Конструктор с параметрами, вызывающий конструктор по умолчанию
базового класса
  public Piano(int keyCount, int pedalCount) : base()
  {
          KeyCount = keyCount;
          PedalsCount = pedalCount;
      }

  // Конструктор с параметрами, вызывающий конструктор с параметрами
базового класса
  public Piano(string name, string soundRange, string timbre, string type,
int stringCount, int scaleLength, int keyCount, int pedalCount)
          : base(name, soundRange, timbre, type, stringCount, scaleLength)
  {
          KeyCount = keyCount;
          PedalsCount = pedalCount;
      }

  }
```

Абстрактные классы

Абстрактный класс – незавершенный класс, который используется только для наследования. Для такого класса невозможно создать объект.

Обычно абстрактные классы используются в качестве начального компонента иерархии классов.

Или просто, в случае, когда класс описывает какую-то абстрактную сущность, от которой всегда должны наследоваться другие классы.

Абстрактный класс помечается ключевым словом abstract.

Абстрактные члены класса

Только в абстрактном классе могут содержаться абстрактные члены класса.

Абстрактными могут быть методы, свойства, индексаторы и события.

Абстрактный член класса не имеет реализации, и должен обязательно быть реализован в производных классах.

Абстрактные члены класса также помечаются ключевым словом abstract.

В производных классах реализация абстрактных членов класса должна быть помечена ключевым словом override.

Абстрактные свойства и методы

Свойства, помеченные как абстрактные, не должны иметь реализации, т.е. у них не должно быть кода аксессоров get и set:

```
  спецификатор_доступа abastract тип Имя
  {
  get;
  // Get безтела
  set;
  // Set без тела
  }
```

Абстрактные методы не должны иметь тела функции, только сигнатуру вызова:

```
спецификатор_доступа abstract возвращаемый_тип Имя(параметры);
// Метод без тела
```

Замечание о static

Ключевые слова abstract и static несовместимы. Собственно, как и слово virtual со словом static.

Статические члены класса в принципе не могут быть ни абстрактными, ни виртуальными, ни любыми другими – только статическими.

Пример

```
// Абстрактный класс животного
abstractclassAnimal
{
// Абстрактные свойства
publicabstractstring Specie { get; protectedset; }
publicabstractstring Name { get; set; }

// Защищённое статическое свойство
protectedstaticstring Voice { get; set; }

// Обычное поле со свойством
privateint age;
publicint Age
    {
get
        {
return age;
}
set
        {
            age = value;
        }
    }

// Абстрактный метод, без реализации
publicabstractbool Breeding();

// Виртуальный метод
publicvirtualvoid Eat()
    {
        Console.WriteLine("{0} eat grass", Name);
    }

// Обыкновенный метод
publicstring MadeSound()
    {
        Console.WriteLine("{0} - {0}!", Voice, Voice);
```

```
  return Voice;
  }
  }

  // Производный класс от класса Животного - класс Черепаха
  classTurtle : Animal
  {

  // Реализуем абстрактное свойство базового класса
  privatestring specie;
  publicoverridestring Specie
      {
  get
          {
  return specie;
          }
  protectedset
          {
  if (string.Compare(value, "Green Turtle", true) == 0 ||
  string.Compare(value, "Flatback", true) == 0 ||
  string.Compare(value, "Loggerhead", true) == 0)
  {
                  specie = value;
              }
          }
      }

  // Реализуем второе абстрактное свойство
  publicoverridestring Name { get; set; }

  // Статический конструктор
  static Turtle()
      {
          Voice = "....";
      }

  // Реализуем абстрактный метод базового класса
  publicoverridebool Breeding()
  {
  var randomGenerator = new Random();
  int randInt = randomGenerator.Next(3);
  if (randInt == 0)
          {
              Console.WriteLine("Turtle {0} laid eggs on the beach!",
Name);
  returnfalse;
          }
  else
          {
              Console.WriteLine("Turtle {0} failed to lay eggs...", Name);
  returntrue;
          }
      }
```

```
  // Новыйметод
  publicvoid TransformToNinja()
      {
  string[] suitableNames = { "Leonardo", "Micheleangelo", "Donatello",
"Raphael" };
  if (Array.IndexOf(suitableNames, Name) != -1)
          {
              Console.WriteLine("And....");
              Console.WriteLine("Transformation!..............");
              Console.WriteLine("Now {0} is Ninja Turtle!", Name);
          }
  else
          {
              Console.WriteLine("And....");
              Console.WriteLine("Something went wrong...........");
              Console.WriteLine("Turtle {0} died.....", Name);
  }
      }
  }
```

Интерфейсы

К абстрактным классам близки Интерфейсы.

Интерфейс (или Протокол) – набор абстрактных членов (обычно методов), которые должны быть обязательно реализованы классами с этим интерфейсом.

Интерфейсы описываются с помощью ключевого слова interface.

Интерфейсы это отдельные элементы в программировании, наподобие классов, структур, перечислений.

```
  Interface IName
  {
  // ...
  }
```

Имена интерфейсов должны начинаться на букву I.

Интерфейсы и классы

Как мы уже знаем, любой класс может наследоваться от другого класса.

Помимо этого, любой класс может реализовывать любое количество интерфейсов.

Класс, реализующий интерфейс

Для обозначения реализации интерфейса используется такая же нотация, как и при наследовании.

Единственное, что при этом имя родительского класса, если он есть, должно всегда быть первым в списке.

```
classSomeClass1 : IInterface1
{
// ...
}

classSomeClass2 : ParentClass, IInterface1
{
// ...
}

classSomeClass3 : ParentClass, IInterface1, IInterface2, IInterface3
{
// ...
}
```

Различия интерфейсов и абстрактных классов

Интерфейсы состоят из абстрактных методов (или других членов класса).

В отличие от абстрактных классов, в интерфейсе не может присутствовать ничего другого, т.е. никакой реализации (до версии C#8.0).

Абстрактные члены в абстрактном классе могут иметь различные спецификаторы доступа, а в интерфейсе элементы всегда public.

Класс может наследовать лишь один абстрактный класс, а интерфейсов иметь неограниченное количество.

Но главное различие – идейное.

Предназначение интерфейсов

Абстрактные классы применяются при необходимости описать некоторую неопределенную сущность, понятие, раскрываемое в дальнейшем через производные классы.

А интерфейсы описывают способность делать что-то, наличие какой-либо функциональности.

Например, интерфейс ICallable – описывает способность совершать и отвечать на звонки. Будет реализовываться классами Telephone, Smartphone, Tablet и WalkieTalkie.

Поэтому имена интерфейсов принято заканчивать суффиксом -able.

Еще немного об интерфейсах

Методы в интерфейсах описываются без всяких дополнительных ключевых слов, и даже без спецификаторов доступа.

А при реализации классом все методы из интерфейса уже должны быть со спецификатором доступа public.

Интерфейсы играют особую роль в полиморфизме, но об этом попозже.

```
Interface ICallable
{
bool MakeCall(string number);
void TakeCall();
}
```

Примеры

Модифицируем пример с собакой, добавив к нему интерфейс для способности лаять:

```
interfaceIBarkable
{
void Bark(int power);
void Whine();
}

classDog : IBarkable
{
publicstring Name { get; set; }
publicstring Breed { get; set; }
publicint Age { get; set; }
publicstring Owner { get; set; }

publicvirtualvoid Walk()
    {
        Console.WriteLine("The {0} walks!", Name);
        Random randGen = new Random();
int rndNumber = randGen.Next(4);
if (rndNumber == 3)
        {
this.Pee();
        }
    }

privateint Pee()
    {
        Console.WriteLine("The {0} has urinated..", Name);
        Random randGen = new Random();
int strenth = randGen.Next(1, 11);
return strenth;
    }

publicoverridestring ToString()
    {
return$"Dog named {Name}, with owner {Owner}, {Breed}, {Age} years";
    }

publicvoid Bark(int power)
{
switch (power)
        {
case 0:
                Whine();
break;
case 1:
                Console.WriteLine("Woof...");
break;
case 2:
                Console.WriteLine("Woof!");
```

```
break;
case 3:
                Console.WriteLine("Woof! Woof! Woof!");
break;
case 4:
                Console.WriteLine("WOOOOF!!! WOOOFFFF!!!");
break;
default:
                Console.WriteLine("Quack");
break;
        }
    }

publicvoid Whine()
    {
        Console.WriteLine("Whiiee..");
        Console.WriteLine("Whie...");
        Console.WriteLine(".....");
}
}
```

Наследование интерфейсов

Интерфейсы, также, как и классы, могут наследовать другие интерфейсы.

Но тут нет ничего особенного. Если у класса в интерфейсах встретятся методы с одинаковой сигнатурой, одна реализация подойдет для обоих интерфейсов.

Стерильные классы

Стерильные или запечатанные классы – это классы, от которых запрещено наследоваться, от такого класса невозможно создать производный.

Запечатанными обычно назначаются классы в распространяемых библиотеках, которые имеют логически завершенную структуру и функциональность, и расширение которых больше не предполагается.

Для того, чтобы отметить класс как запечатанный, используется ключевое слово sealed.

Пример

```
// Класс Змеи, производный от класса Животного
classSnake : Animal
{
// ...
}

// Запечатанный класс Зелёного питона, производный от класса Змеи
// От него нельзя больше делать производные классы
sealedclassGreenTreePython : Snake
{
// ...
}
```

Запечатанные методы

Методы тоже можно запечатывать. Но не все, а только переопределенные виртуальные.

Таким образом, они защищаются от последующего переопределения.

Для этого метод помечается тем же ключевым словом sealed.

Пример

```
// Допустим, метод Breeding у класса Snake:
// sealed может быть только у override метода
publicoverridesealedbool Breeding()
{
    Random r = new Random();
int numberOfEggs = r.Next(4, 20);
    Console.WriteLine("The snake laid {0} eggs", numberOfEggs);
}

// Теперь класс GreenTreePython не сможет переопределить этот метод
```

Сокрытие методов

И, напоследок, кое-что еще.

Помимо переопределения методов, есть еще и сокрытие методов.

Сокрытие – это создание в производном классе метода, одинакового по сигнатуре с методом в базовом классе. Метод в производном классе перекрывает метод в базовом классе.

Т.е. это написание метода с таким же именем и параметрами, как в базовом классе.

И за счет совпадения имен будет виден только один из них, а именно скрывающий метод в производном классе.

Для сокрытия скрывающий метод должен быть помечен ключевым словом new.

Применение

Главным образом сокрытие нужно тогда, когда в классе обязательно требуется метод с некоторым названием, а метод с таким названием уже есть в базовом классе, и он не виртуальный.

И из-за того, что он не виртуальный, нельзя выполнить переопределение этого метода.

Тогда единственным способом описать метод с таким же именем остается механизм сокрытия.

Пример

```
classDog
{
publicstring Name { get; set; }
publicstring Breed { get; set; }
```

```
  publicint Age { get; set; }
  publicstring Owner { get; set; }

  publicvirtualvoid Walk()
      {
          Console.WriteLine("The {0} walks!", Name);
          Random randGen = new Random();
  int rndNumber = randGen.Next(4);
  if (rndNumber == 3)
          {
  this.Pee();
          }
      }

  privateint Pee()
      {
          Console.WriteLine("The {0} has urinated..", Name);
          Random randGen = new Random();
  int strenth = randGen.Next(1, 11);
  return strenth;
      }

  // Переопределение
  publicoverridestring ToString()
      {
  return$"Dog named {Name}, with owner {Owner}, {Breed}, {Age} years";
      }

  // Новый метод
  publicint GetStamina()
      {
  int stamina = 20 * (8 - Math.Abs(Age - 8)) / 8;
          Console.WriteLine("The dog stamina: {0}/100", stamina);
  return stamina;
      }
  }

  classPoliceDog : Dog
  {
  publicint Rank { get; set; }
  publicstring Specialization { get; set; }

  public PoliceDog(string name, int age, string breed, string owner, int
rank, string spec)
      {
          Name = name;
          Age = age;
          Breed = breed;
          Owner = owner;
          Rank = rank;
          Specialization = spec;
      }
```

```
  publicint UpRank()
      {
          ++Rank;
  return Rank;
      }

  publicvoid Train()
      {
  base.Walk();
          Console.WriteLine("The dog {0} goes to the hard training",
Name);
      }

  publicoverridevoid Walk()
      {
          Console.WriteLine("The police dog {0} walks!", Name);
      }

  // Переопределённый метод
  publicoverridestring ToString()
      {
  return"Police " + base.ToString() + $". With rank {Rank} and
specializing in {Specialization}";
      }

  // Скрывающий метод
  publicnewint GetStamina()
  {
  // Изменённая формула - вместо 20 здесь значение 100
  int stamina = 100 * (8 - Math.Abs(Age - 8)) / 8;
          Console.WriteLine("The police dog stamina: {0}/100", stamina);
  return stamina;
      }

  }

  classProgram
  {
  staticvoid Main(string[] args)
      {
  // Создаём объект класса Dog
          Dog dog1 = new Dog();
          dog1.Name = "Mikkey";
          dog1.Breed = "American cocker spaniel";
          dog1.Age = 2;
          dog1.Owner = "Mr. Garrison";

  // Создаём объект класса PoliceDog
          PoliceDog dog2 = new PoliceDog("Ralf", 8, "Indian pariah dog",
"Officer Mc'Cry", 3, "Explosive");

  // Вызов ToString класса Dog
          dog1.GetStamina();
```

```
  // Вывод: The dog stamina: 5/100

  // Вызов ToString класса PoliceDog, который внутри вызывает ToString
класса Dog
          dog2.GetStamina();
  // Вывод: The police dog stamina: 100/100
  }
  }
```

Сокрытие, переопределение...

При сокрытии в классе остается два варианта метода: скрытый, унаследованный от базового класса, доступ к которому нельзя получить; и новый видимый, который перекрывает предыдущий.

Зачем это, и в чем отличие от переопределения?..

Пора выйти из тени Полиморфизму!

Глава 6. ПОЛИМОРФИЗМ

Определение полиморфизма

Полиморфизмом (рис. 6.1) называется способность одних типов выступать в форме других типов. Или же способность функции обрабатывать данные разных типов.

Рисунок 6.1 – Полиморфизм

База

Как один тип может выступать в форме другого типа?

Попробуем взглянуть на наследование с такой стороны:

Наследование – это отношение «является». Любой производный класс «является» сущностью, которую представляет его базовый класс.

Пусть у нас есть класс Fruit. И класс Banana, производный от него. А еще есть класс Apple, тоже потомок Fruit.

В таком случае, Banana «является» Fruit, и Apple «является» Fruit.

Полиморфизм в коде

Это свойство классов проявляется в коде в виде возможности представить объект какого-то класса в виде объекта базового для него класса.

```
// Базовый класс Фрукт
classFruit
{
// Свойство Название
publicstring Name { get; set; }
}

// Производный от Фрукта класс Банан
classBanana : Fruit
{
// Свойство Длина
publicint Length { get; set; }
}

// Производный от Фрукта класс Яблоко
classApple : Fruit
{
// Свойство Цвет
publicstring Color { get; set; }
}

classProgram
{
staticvoid Main(string[] args)
    {
// Создаём экземпляр класса Banana
        Banana ban = new Banana();
// Присваиваем объект типа Banana ссылке на тип Fruit
        Fruit fruit = ban;

// Создаём экземпляр класса Apple
        Apple ap = new Apple();
// Присваиваем объект типа Aplle ссылке на тип Fruit
        Fruit fr2 = ap;

// Создаём ещё один объект типа Apple и помещаем его в тип Fruit
        Fruit fr3 = new Apple();
    }
}
```

Виды полиморфизма

Полиморфизм бывает 2 видов:

– параметрический (истинный) полиморфизм;

– adhoc полиморфизм.

Adhoc полиморфизм представляет собой не совсем настоящий полиморфизм. Он наблюдается тогда, когда мы вручную прописываем различные варианты поведения для различных типов.

Adhoc полиморфизм

Простейшим случаем adhoc полиморфизма можно считать перегрузку функций: когда для различных типов параметров прописываются различные варианты методов.

```
// Класс Шеф-повара
classChef
{
// Метод Приготовить что-нибудь из банана
publicstring CookSmth(Banana b)
{
return"Banana cake";
    }

// Метод Приготовить что-нибудь из яблока
publicstring CookSmth(Apple a)
{
return"Charlotte";
    }

// Метод Приготовить что-нибудь из киви
publicstring CookSmth(Kiwi k)
{
return"Smoothie";
    }

}
```

Ещё один вариант adhoc полиморфизма – когда для всех типов используется один общий метод, но все равно проверяется исходный тип и для каждого типа выполняются отдельные действия.

```
classChef
{
// Метод Приготовить что-нибудь из фрукта
publicstring CookSmth(Fruit fruit)
    {
string meal = "nothing";
// Если тип фрукта  - банан
if (fruit is Banana)
```

```
        {
            meal = "Banana cake";
        }
// Если тип фрукта  - яблоко
elseif (fruit is Apple)
        {
            meal = "Charlotte";
        }
// Если тип фрукта  - киви
elseif (fruit is Kiwi)
        {
            meal = "Smoothie";
        }
return meal;
    }

}

classProgram
{
staticvoid Main(string[] args)
    {
// Создаём шефа
        Chef mrChef = new Chef();

// По одному экземпляру каждого класса
        Fruit fruit1 = new Fruit();
Banana fruit2 = new Banana();
        Apple fruit3 = new Apple();
Kiwi fruit4 = new Kiwi();

// И смотрим варианты вызова метода CookSmth,
// передавая в него объекты разных типов
string meal = mrChef.CookSmth(fruit1);
        Console.WriteLine("First meal: " + meal);

        meal = mrChef.CookSmth(fruit2);
        Console.WriteLine("Next meal: " + meal);

        meal = mrChef.CookSmth(fruit3);
        Console.WriteLine("Next meal: " + meal);

        meal = mrChef.CookSmth(fruit4);
        Console.WriteLine("And final meal: " + meal);
}
}
```

Коллекции и полиморфизм

Благодаря полиморфизму можно не только создавать методы, которые будут работать с разными типами, как с одним, но и работать с коллекциями из объектов разных классов.

Зная, что объекты классов Banana, Apple и Kiwi могут быть представлены как объекты класса Fruit, можно создать массив фруктов, и поместить в него все наши объекты.

Упростим код из предыдущего примера с помощью этой техники.

Фрукты

```
  staticvoid Main(string[] args)
  {
  // Шефа не трогаем
      Chef mrChef = new Chef();

  // Объявляем массив фруктов и инициализируем его списком объектов разных
классов
  Fruit[] fruits = { new Fruit(), new Banana(), new Apple(), new Kiwi() };

  // Циклом перебираем фрукты в массиве
  foreach (Fruit fruit in fruits)
      {
  string meal = mrChef.CookSmth(fruit);
          Console.WriteLine("Next meal: " + meal);
  }
  }
```

Оператор is

Как вы уже поняли, оператор is используется для проверки «является» ли объект указанным типом, т.е. находится ли этот тип в иерархии наследования проверяемого объекта.

Главная особенность именно в том, что он проверяет не только тот тип, которым создавался объект, а все типы в его иерархии наследования (а также реализуемые интерфейсы).

Это значит, что, например, для объекта класса Apple положительными были бы проверки, что он является Apple, Fruit и Object.

Синтаксис вы уже видели:

```
  объект is тип
```

Класс Type

Существует и другой способ узнать тип объекта - через класс Type.

Type – специальный класс, представляющий собой информацию о некотором типе. Для любого типа (класса) можно получить соответствующий ему объект класса Type.

И именно для этого в классе Object есть метод GetType(). А раз он есть в классе Object, он есть любом классе .NET.

Этот метод возвращает объект типа для своего объекта.

Но с чем его сравнивать?

Оператор typeof

Оператор typeof возвращает объект класса Type для заданного типа.

Синтаксис применения следующий:

```
typeof(тип)
```

Итак, GetType позволяет получить объект Type для объекта, typeof позволяет получить объект Type для типа.

Скомбинировав их, можем записать проверку на однозначное соответствие объекта какому-либо типу:

```
объект.GetType() == typeof(тип)
```

Пример

```
Banana banan = new Banana();

Console.WriteLine("Operator is:");
Console.WriteLine("  banan is Banana: {0}", banan is Banana);  // true
Console.WriteLine("  banan is Kiwi: {0}", banan is Kiwi);
      // false
Console.WriteLine("  banan is Fruit: {0}", banan is Fruit);
      // true
Console.WriteLine("  banan is Object: {0}", banan is Object);  // true


Console.WriteLine("Class Type:");
Console.WriteLine("  banan.GetType() == typeof(Banana): {0}",
banan.GetType() == typeof(Banana));     // true
Console.WriteLine("  banan.GetType() == typeof(Kiwi): {0}",
banan.GetType() == typeof(Kiwi));               // false
Console.WriteLine("  banan.GetType() == typeof(Fruit): {0}",
banan.GetType() == typeof(Fruit));              // false
Console.WriteLine("      banan.GetType()   ==   typeof(Object):   {0}",
banan.GetType() == typeof(Object));     // false
```

Обратное преобразование

Иногда необходимо восстановить исходный тип объекта.

Например, передав объект класса Banana в метод в виде Fruit, требуется восстановить из этого Fruit исходный тип Banana.

Если попробовать выполнить преобразование через обычный синтаксис приведения типов, то в случае, если попадётся несовместимый тип, программа закончится аварийно – выбросит исключение.

```
publicvoid SomeMethod(Fruit fruit)
{
// В случае, если сюда передать любой фрукт, кроме банана
// Произойдет исключение - преобразование типов невозможно
    Banana ban = (Banana)fruit;
Console.WriteLine("Banana length is: " + ban.Length);
}
```

Оператор as

Конечно, можно сначала написать условие, проверяющее тип, и только в случае выполнения этого условия проводить преобразование... Но есть и другой способ.

Оператор as выполняет преобразование объекта в тип, находящийся в его иерархии наследования, а если указанный тип не связан с объектом – возвращает нулевую ссылку null. Аварийных исключений здесь не происходит.

Синтаксис у него такой:

```
ожидаемый_тип новое_имя = объект as ожидаемый_тип;
```

Оператор as подходит только для преобразования типов, связанных наследственными узами.

Использование

```
// Теперь нужно только проверить, что преобразование удалось выполнить
Banana ban = fruit as Banana;
if (ban != null)
{
      Console.WriteLine("Banana length is: " + ban.Length);
}
```

Вообще, начиная с 7 версии C# оператор as стал немного бесполезным, потому что его функциональность добавили к новой форме оператор ais:

```
// Теперь преобразовывать можно прямо в операторе is
if (fruit is Banana ban)
{

      // Если проверка is пройдёт, то здесь будет доступен
преобразованный объект ban
      Console.WriteLine("Banana length is: " + ban.Length);
}
```

Вспомнили про Object..

Если Object является прародителем всего и вся, значит объект любого класса может быть представлен типом object?

В общем-то, конечно.

Можно написать метод, принимающий параметр типа object, и тогда в этот метод можно будет передать объект абсолютно любого класса.

Правда, это редко бывает нужно.

Боксинг и анбоксинг

Но у такого представления в виде типа object есть и свои преимущества, например, если говорить о хранении в виде object структур (стандартных типов данных: int, float, bool и т.д.).

Структуры являются типами значений, а это значит, что в методы передаются копии их значений.

А Object является классом, т.е. ссылочным типом, и объекты типа object передаются в методы по ссылке.

У этого даже есть специальные названия:

Представление типа значения в виде object – боксинг.

Возвращение типа значения из типа object в объект исходного типа – это анбоксинг.

Виды полиморфизма

Еще одна классификация видов полиморфизма.

По типу применяемых элементов полиморфизм делится на:

– полиморфизм подтипов;

– полиморфизм интерфейсов.

Мы пока рассматриваем полиморфизм подтипов (т.е. на основе наследования).

И до этого он у нас был ненастоящим (adhoc полиморфизм).

Истинный (параметрический) полиморфизм

Что же такое истинный полиморфизм?

Полиморфизм считается истинным, когда в методе описан только один сценарий поведения для любых входных типов. Но, тем не менее, поведение меняется в зависимости от типа входящего объекта.

И как такого можно достичь?

Перегрузка методов

И это ответ – с помощью перегрузки методов.

Описав в базовом классе виртуальный метод, функция, работающая с этим базовым классом, может вызывать этот виртуальный метод. А он, в свою очередь, у каждого производного класса будет переопределен и реализован по-своему.

Т.е. вызываться будет один метод, но он будет делать разные вещи, в зависимости от того, как он переопределен в этом классе.

Пример настоящего полиморфизма

```
classFruit
{
publicstring Name { get; set; }

// Виртуальный метод "Получитьблюдо"
publicvirtualstring GetMealFromThis()
    {
return"Nothing";
    }
}

classBanana : Fruit
```

```
  {
  publicint Length { get; set; }

  // Этот метод переопределён
  // И для банана возвращает такую строку
  publicoverridestring GetMealFromThis()
      {
  return"Banana Cake";
      }
  }

  classApple : Fruit
  {
  publicstring Color { get; set; }

  // Для яблока - вторую
  publicoverridestring GetMealFromThis()
      {
  return"Charlotte";
      }
  }

  classKiwi : Fruit
  {
  publicstring Size { get; set; }

  // А для киви - третью
  publicoverridestring GetMealFromThis()
      {
  return"Smoothie";
      }
  }

  classChef
  {
  // Метод Приготовления
  publicstring CookSmth(Fruit fruit)
  {
  // Просто возвращает результат выполнения метода GetMealFromThis
  return fruit.GetMealFromThis();
      }

  }

  classProgram
  {
  staticvoid Main(string[] args)
      {
          Chef mrChef = new Chef();

          Fruit[] fruits = { new Fruit(), new Banana(), new Apple(), new
Kiwi() };
```

```
foreach (Fruit fruit in fruits)
        {
string meal = mrChef.CookSmth(fruit);
            Console.WriteLine("Next meal: " + meal);

}
    }
}
```

Переопределение и сокрытие

Теперь можно рассказать и про различия переопределения от сокрытия.

При переопределении, как вы уже видели, в случае представления объекта в форме базового класса, при вызове виртуального метода вызывается его переопределенная версия из дочернего класса.

При сокрытии же такого не происходит, и полиморфизм не работает. Для объекта базового класса всегда будет выполняться вариант метода, описанный в базовом классе.

Демонстрация различий

```
classFruit
{
publicstring Name { get; set; }

publicvirtualstring GetMealFromThis()
    {
return"Nothing";
    }

// Добавили новый виртуальный метод
publicvirtualstring GetSauce()
    {
return"out sauce";
    }
}

classBanana : Fruit
{
publicint Length { get; set; }

publicoverridestring GetMealFromThis()
    {
return"Banana Cake";
}

// Скрываем оригинал новой версией
publicnewstring GetSauce()
{
return" banana ketchup";
    }
}
```

```
classApple : Fruit
{
publicstring Color { get; set; }

publicoverridestring GetMealFromThis()
    {
return"Charlotte";
}

// Скрываем оригинал новой версией
publicnewstring GetSauce()
{
return" apple juice";
    }
}

classKiwi : Fruit
{
publicstring Size { get; set; }

publicoverridestring GetMealFromThis()
    {
return"Smoothie";
}

// Скрываем оригинал новой версией
publicnewstring GetSauce()
{
return" kiwi compote";
    }
}

classChef
{
publicstring CookSmth(Fruit fruit)
{
// Формирует результат из вызовов двух методов
return fruit.GetMealFromThis() + " with" + fruit.GetSauce();
}

}
```

Большой пример с абстрактным классом и переопределениями методов можно посмотреть в Приложении Б.

А в Приложении В можно найти пример еще больше – полноценную программу с использованием иерархий наследования.

Проблема

Имеется класс Улитки и есть класс Черепахи. А черепахи едят улиток. Это значит, что улитка также должна быть Feed. Но она не может одновременно наследоваться и от класса Animal и от класса Feed.

И здесь на помощь приходят Интерфейсы.

Полиморфизм интерфейсов

Также, как, и с базовыми классами, объекты, реализующие какой-либо интерфейс, могут быть представлены в виде объекта типа этого интерфейса. Такой тип полиморфизма называется Полиморфизм интерфейсов.

Пример

```
// Интерфейс "Лающее"
interfaceIBarkable
{
// Метод Лаять
void Bark(int power);
// МетодСкулить
void Whine();
}

// Класс Собаки, реализующего интерфейс "Лающее"
classDog : IBarkable
{
publicstring Name { get; set; }
publicstring Breed { get; set; }
publicint Age { get; set; }
publicstring Owner { get; set; }

publicvirtualvoid Walk()
    {
        Console.WriteLine("The {0} walks!", Name);
        Random randGen = new Random();
int rndNumber = randGen.Next(4);
if (rndNumber == 3)
        {
this.Pee();
        }
    }

privateint Pee()
    {
        Console.WriteLine("The {0} has urinated..", Name);
        Random randGen = new Random();
int strenth = randGen.Next(1, 11);
return strenth;
    }

publicoverridestring ToString()
    {
return$"Dog named {Name}, with owner {Owner}, {Breed}, {Age} years";
    }

// РеализацияметодаЛаять
publicvoid Bark(int power)
    {
switch (power)
```

```
        {
case 0:
                Whine();
break;
case 1:
                Console.WriteLine(" - Woof...");
break;
case 2:
                Console.WriteLine(" - Woof!");
break;
case 3:
                Console.WriteLine(" - Woof! Woof! Woof!");
break;
case 4:
                Console.WriteLine(" - WOOOOF!!! WOOOFFFF!!!");
break;
default:
                Console.WriteLine(" - Quack.");
break;
        }
    }

// Реализация метода Скулить
publicvoid Whine()
{
        Console.WriteLine(" - Whiiee..");
        Console.WriteLine(" - Whie...");
Console.WriteLine("   .....");
    }
}

// Класс Полицейская Собака - потомок класса Собаки
classPoliceDog : Dog
{
// ...
}

// КлассКота
classCat
{
private Random generator;

// Конструктор
public Cat()
{
// Для инициализации генератора случайных чисел
        generator = new Random();
}

// МетодМяукнуть
protectedstring Meow()
    {
string word = " - Meeeoooowwww!";
```

```
        Console.WriteLine(word);
return word;
    }

// Метод Убежать от кого-то лающего
// В качестве параметра - объект интерфейса
publicvoid RunAwayFrom(IBarkable barkable)
    {
// Гавкаем и мяукаем друг на друга
int rnd, iter = 1;
do
        {
            Meow();
            rnd = generator.Next(2);
            barkable.Bark(iter);
            ++iter;
        }
while (rnd != 0);
// Иубегаем
        Console.WriteLine("The cat ran away and climbed a tree.\n");
    }
}

classProgram
{
staticvoid Main(string[] args)
    {
// Создаём экземпляр кота
        Cat cat = new Cat();

// Экземпляр собаки
        Dog dog1 = new Dog();

// Вызываем метод кота RunAwayFrom, передавая туда объект собаки
// Потому что собака - лающее
        cat.RunAwayFrom(dog1);

// Создаём экземпляр полицейской собаки
        PoliceDog dog2 = new PoliceDog();

// Вызываем метод, передаём полицейского пса
// Полицейская собака тоже лающее
        cat.RunAwayFrom(dog2);

// Создаём экземпляр собаки и присваеваем объекту интерфейса
IBarkable barkable = new Dog();

// Тоже подходит
cat.RunAwayFrom(barkable);
    }
}
```

Дополняем

```
  classDrunkard : IBarkable
  {

  publicvoid Bark(int power)
      {
  switch (power)
          {
  case 0:
                  Whine();
  break;
  case 1:
              Console.WriteLine(" - Wof-f-f..");
  break;
  case 2:
              Console.WriteLine(" - Wooof! Hrraaarf!");
  break;
  case 3:
              Console.WriteLine(" - Hurrrrgff!.. Gh-rrr-aaaaah! Wof!");
  break;
  case 4:
              Console.WriteLine(" - WOF! R-R-RAFF! A new ide suda....");
  break;
  default:
              Console.WriteLine(" - ROF! FOPH! WOF! RGRA-A-A-a-a-a....
Arghf-awww-shaaafff!!");
  break;
          }
      }

  publicvoid Whine()
      {
          Console.WriteLine(" - Ukh!");
      }
  }

  // ...

  Drunkard man = new Drunkard();

  cat.RunAwayFrom(man);

  IBarkable barkable2 = new Drunkard();

  cat.RunAwayFrom(barkable2);
  // ...
```

Явная реализация интерфейсов

Помимо обычной реализации членов интерфейса существуем механизм явной реализации.

Явно реализованный член интерфейса будет доступен в объекте только тогда, когда объект будет представлен в форме этого интерфейса.

С помощью этого можно «прятать» методы внутри классов, а также управлять поведением членов класса в зависимости от формы их представления.

Синтаксис

Чтобы явно реализовать элемент интерфейса в классе, перед его именем нужно через точку обозначить название интерфейса, которому этот элемент принадлежит:

```
// Интерфейс
interfaceITestable
{
// С одним методом
void Test();
}

// Класс, реализующий его
classTestClass : ITestable
{
// Обычная реализация
publicvoid Test()
{
        Console.WriteLine("Implicit Interface implementation");
    }

// Явная реализация для ITestable
publicvoid ITestable.Test()
    {
        Console.WriteLine("Explicit Interface implementation");
    }
}

classProgram
{
staticvoid Main(string[] args)
    {
        TestClass c = new TestClass();
// Выведетодно
        c.Test();

        ITestable i = c;
// Выведет другое
        i.Test();
    }
}
```

Прячем методы

В классе можно описать только явную реализацию метода интерфейса. Тогда такой метод не будет виден в этом классе, даже изнутри, пока его не привести к типу интерфейса.

```
interfaceITestable
{
void Test();
}

classTestClass : ITestable
{
// Только явная реализация для ITestable
publicvoid ITestable.Test()
    {
        Console.WriteLine("Explicit Interface implementation");
    }

// Method in wich we call Test method
publicvoid TestMethodTest()
    {
        Console.WriteLine("Can I access Test method inside a class?");
        Test();     // Right answer - no
    }
}

classProgram
{
staticvoid Main(string[] args)
{
        TestClass c = new TestClass();

// Ошибка - нет такого метода
c.Test();

// Аналогично
        c.TestMethodTest();

        ITestable i = c;
// Сработает
        i.Test();

}
}
```

Совпадение имён

В случае совпадения названий методов в Интерфейсах/Классе, для каждого можно написать свою реализацию:

```
// Базовый класс Спортсмен
classSportsman
{
// Метод Run
publicvoid Run()
    {
        Console.WriteLine("I am a Sportsman");
    }
}
```

```
// ИнтерфейсБегун
interfaceIJogger
{
// Метод Run
void Run();
}

// ИнтерфейсЛыжник
interfaceISkier
{
// И здесь метод Run
void Run();
}

// Производный класс Атлет, реализующий интерфейсы ISkier, IJogger
classAthlete : Sportsman, ISkier, IJogger
{
// Перекрывающий метод базового класса метод Run
publicnewvoid Run()
    {
        Console.WriteLine("I am an Athlete");
}

// Явная реализация Run для интерфейса ISkier
void ISkier.Run()
    {
        Console.WriteLine("I am a Skier");
}

// Явная реализация Run для интерфейса IJogger
void IJogger.Run()
    {
        Console.WriteLine("I am a Jogger");
    }
}

classProgram
{
staticvoid Main(string[] args)
    {
// Создаём атлета
var sp = new Athlete();
// Вызываем метод Run из-под атлета
        sp.Run();
// Вызываем метод Run из-под спортсмена
        (sp as Sportsman).Run();
// Вызываем метод Run из-под лыжника
        (sp as ISkier).Run();
// Вызываем метод Run из-под бегуна
        (sp as IJogger).Run();
    }
}
```

Массив в .NET

Массив в .NET представлен классом Array, и этот класс реализует интерфейс списка IList.

Т.е. в нем есть все методы для работы со списком: Add, Remove, Clear, IndexOf, Count (но большинство из них, конечно же, не рабочие). Увидеть и применить их можно толькопредставив массив в виде интерфейса IList.

```
int[] mas = newint[10];
IList<int> vs = mas;
```

Гибкость

Интерфейсы дают неимоверную гибкость.

Для полиморфизма, как правило, используют именно интерфейсы. Наследование имеет значительное ограничение из-за невозможности иметь нескольких родителей, поэтому на практике его применяют в основном только для переиспользования кода.

Сила полиморфизма интерфейсов

Полиморфизм интерфейсов позволяет создавать программу на основе интерфейсов без фиксированной реализации.

Такая способность открывает возможность легкой поддержки и модификации любых частей кода, расширяемость и адаптируемость, простоту тестирования и дополнительных интеграций.

Мыслить интерфейсами

Почти во всех случаях грамотная архитектура ООП приложения должна опираться именно на Интерфейсы. Любые зависимости следует представлять в виде интерфейсов, все взаимодействие программных компонентов должно вестись через интерфейсы.

Для построения правильной архитектуры ПО нужно мыслить интерфейсами, а не классами.

Обширную демонстрацию применения интерфейсов можно увидеть в тестовой программе из Приложения Г и в программе из Приложения Д.

Пример

```
// Интерфейс Логгера
interfaceILogger
{
// С одним методом - для записи строки
void Log(string text);
}

// Перечисление видов логгеров
enumLoggerType { Console, File }

// Статический класс для создания логгера
```

```
staticclassLoggerFabric
{
// Вложенные типы: классы, видимые только внутри класса:

// Приватный класс Консольного логгера, реализует интерфейс ILogger
privateclassConsoleLogger : ILogger
    {
// Реализация метода Log из интерфейса ILogger
publicvoid Log(string text)
{
// Выводим строку на консоль
            Console.WriteLine(text);
        }
    }

// Приватный класс Файлового логгера, тоже реализует интерфейс логгера
privateclassFileLogger : ILogger
    {
// Приватное поле только для чтения для имени файла
privatereadonlystring filename;

// Конструктор по умолчанию
public FileLogger()
        {
// Задаём имя файла
            filename = "log.log";
}

// Реализация метода Log из интерфейса ILogger
publicvoid Log(string text)
        {
// Записываемстрокувфайл
            File.AppendAllText(filename, text + "\n");
}
    }

// Обычные члены класса LoggerFabric:
// Приватный статический словарь
// Ключом выступаем элемент перечисления LoggerType
// Значением - объект, реализующий интерфейс логгера
privatestatic Dictionary<LoggerType, ILogger> loggers;

// Статический конструктор
static LoggerFabric()
    {
// Создание списка
loggers = new Dictionary<LoggerType, ILogger>();
}

// Публичный статический метод Получения логгера
// Выдаёт запрашиваемый логгер, по типу логгера
publicstatic ILogger GetLogger(LoggerType type)
{
```

```
        ILogger logger;
  // Пытаемся получить логгер из словаря по переданному типу логгера в
качестве ключа
  if (loggers.TryGetValue(type, out logger))
  {
  // Если он есть, возвращаем его
  return logger;
        }
  else
        {
  // Если элемента с таким ключом в словаре нет
  // То, в зависимости от переданного типа логгера
  switch (type)
            {
  case LoggerType.Console:
  // Если это Console - создаём объект класса консольного логгера
  logger = new ConsoleLogger();
  break;
  case LoggerType.File:
  // Если File - создаём объект класса файлового логгера
                    logger = new FileLogger();
  break;
            }
  // Добавляем созданный объект логгера в словарь по ключу его типа
            loggers.Add(type, logger);
  // И возвращаем его
  return logger;
        }
    }
  }

  classProgram
  {
  staticvoid Main(string[] args)
    {
        ILogger console = LoggerFabric.GetLogger(LoggerType.Console);
        console.Log("First String to Console Logger");
        console.Log("Second string to Console Logger");
        console.Log("Third string to Console Logger");

        ILogger file = LoggerFabric.GetLogger(LoggerType.File);
        file.Log("First String to File Logger");
        file.Log("Second string to File Logger");
        file.Log("Third string to File Logger");

        ILogger file2 = LoggerFabric.GetLogger(LoggerType.File);
        console.Log((file == file2).ToString());
  }
  }
```

Стандартные интерфейсы .NET

В .NET есть много готовых интерфейсов, которые можно использовать в своих классах, чтобы назначить им требуемую функциональность.

– IClonable;
– IComparable;
– IEnumerable, IEnumerator;
– IList;
– IConvertible;
– IFormattable.

Пример IFormattable

```
  // Класс Температуры, реализующий стандартный интерфейс IFormattable
  publicclassTemperature : IFormattable
  {
  // Закрытое поле для хранения температуры (в градусах по Цельсию)
  privatedecimal temp;

  // Конструктор
  public Temperature(decimal temperature)
  {
  // Проверяем адекватность значения
  if (temperature < -273.15m)
  thrownew ArgumentOutOfRangeException(string.Format("{0} is less than
absolute zero.",
  temperature));
  this.temp = temperature;
      }

  // Свойство для получения температуры в градусах по Цельсию
  publicdecimal Celsius
      {
  get { return temp; }
  }

  // Свойство для получения температуры в градусах по Фаренгейту
  publicdecimal Fahrenheit
      {
  get { return temp * 9 / 5 + 32; }
  }

  // Свойство для получения температуры в градусах по Кельвину
  publicdecimal Kelvin
      {
  get { return temp + 273.15m; }
  }

  // Переопределение метода ToString от Object-а
  publicoverridestring ToString()
```

```
      {
  // Вызываем ToString с форматированием по культуре
  returnthis.ToString("G", CultureInfo.CurrentCulture);
  }

  // Перегрузка метода ToString с одним параметром - форматом
  publicstring ToString(string format)
  {
  // Вызываем ToString с форматированием по культуре
  returnthis.ToString(format, CultureInfo.CurrentCulture);
  }

  // Перегрузка ToString с параметрами формата и культуры, в соответствии
с которой нужно провести форматирование
  publicstring ToString(string format, IFormatProvider provider)
      {
  // Если формат не задан
  if (string.IsNullOrEmpty(format))
  {
  // Задаём как глобальный
              format = "G";
          }
  // Если культура не задана
  if (provider == null)
          {
  // Задаём как текущую культуру в системе
              provider = CultureInfo.CurrentCulture;
          }
  // В зависимости от переданного формата
  switch (format.ToUpperInvariant())
          {
  case"G":
  case"C":
  // Если G или C возвращаем в градусах Цельсия
  return temp.ToString("F2", provider) + " °C";
  case"F":
  // Если F - в градусах Фаренгейта
  return Fahrenheit.ToString("F2", provider) + " °F";
  case"K":
  // Если K — в градусах Кельвина
  return Kelvin.ToString("F2", provider) + " K";
  default:
  // Если какой-то другой формат - исключение
  thrownew FormatException(string.Format("The {0} format string is not
supported.", format));
          }
      }
  }

  classProgram
  {
  publicstaticvoid Main()
      {
```

```
  // Создаём объект температуры
          Temperature temp1 = new Temperature(0);
  // Используем наше форматирование во WriteLine
          Console.WriteLine("{0:C} (Celsius) = {0:K} (Kelvin) = {0:F}
(Fahrenheit)\n", temp1);

  // Создаём новый объект температуры
          temp1 = new Temperature(-40);
  // Используем метод форматирования с разными вариантами CultureInfo
  // Для текущей культуры установленной в системе
  Console.WriteLine(string.Format(CultureInfo.CurrentCulture, "{0:C}
(Celsius) = {0:K} (Kelvin) = {0:F} (Fahrenheit)", temp1));
  // Для французской культуры
          Console.WriteLine(string.Format(new CultureInfo("fr-FR"), "{0:C}
(Celsius) = {0:K} (Kelvin) = {0:F} (Fahrenheit)\n", temp1));

  // Опять создаём новый объект температуры
          temp1 = new Temperature(32);
  // Выводим консоль с использованием версии ToString с форматом
          Console.WriteLine("{0} (Celsius) = {1} (Kelvin) = {2}
(Fahrenheit)\n",
  temp1.ToString("C"), temp1.ToString("K"), temp1.ToString("F"));
  }
  }
```

Иерархия классов с переопределением виртуальных методов

```
classDog
{
publicstring Name { get; set; }
publicstring Breed { get; set; }
publicint Age { get; set; }
publicstring Owner { get; set; }

publicvoid Walk()
    {
        Console.WriteLine("The {0} walks!", Name);
        Random randGen = new Random();
int rndNumber = randGen.Next(4);
if (rndNumber == 3)
        {
this.Pee();
        }
    }

privateint Pee()
    {
        Console.WriteLine("The {0} has urinated..", Name);
        Random randGen = new Random();
int strenth = randGen.Next(1, 11);
return strenth;
    }

// Новый виртуальный метод "Найти что-нибудь"
// в пределах заданной дистанции
publicvirtualstring FindSmth(int distance)
{
// Генерируем случайное число
        Random gen = new Random();
int rand = gen.Next(100);
string found = "";
// В зависимости от дистанции поиска
if (distance < 10)
{
            found = rand > 80 ? "stick" : "nothing";
        }
else
        {
// В зависимости от случайного числа выбираем, что собака найдёт
if (rand < 33)
            {
                found = "stick";
            }
elseif (rand > 33 && rand < 66)
```

```
            {
                found = "bone";
            }
  elseif (rand > 66 && rand < 95)
            {
                found = "tennis ball";
            }
  else
            {
  string[] vars = { "bottle", "treasure", "doll", "rabbit", "money",
"phone", "broom", "book" };
                rand = gen.Next(vars.Length);
  found = vars[rand];
            }
         }
  // Возвращаем найденную вещь
  return found;
      }
  }

  // Класс собаки-ищейки, производный от класса собаки
  classPoliceDog : Dog
  {
  publicint Rank { get; set; }
  publicstring Specialization { get; set; }

  public PoliceDog(string name, int age, string breed, string owner, int
rank, string spec)
      {
          Name = name;
          Age = age;
          Breed = breed;
          Owner = owner;
          Rank = rank;
          Specialization = spec;
      }

  publicint UpRank()
      {
          ++Rank;
  return Rank;
      }

  publicvoid Train()
      {
          Walk();
          Console.WriteLine("And then goes to the hard training", Name);
  }

  // Новый метод, переопределяющий метод FindSmth базового класса.
  publicoverridestring FindSmth(int distance)
      {
```

```
  var gen = new Random(100);
  int rand = gen.Next();
  string found = "";
  if (distance < 60)
          {
              found = "nothing";
          }
  else
          {
              found = rand < 99 ? "nothing" : "explosive!";
          }
  return found;
      }

  }

  classProgram
  {
  publicstaticvoid Main(string[] args)
  {
  // Создаём объект класса собаки
  var dog1 = new Dog();

  dog1.FindSmth(100);     // "bone"
          dog1.FindSmth(100);     // "bone"
          dog1.FindSmth(100);     // "tennis ball"
          dog1.FindSmth(100);     // "stick"
          dog1.FindSmth(100);     // "rabbit"
          dog1.FindSmth(100);     // "tennis ball"

  // Создаём объект класса собаки-ищейки
  var dog2 = new PoliceDog("Arnold", 12, "Rottweiler", "Sgt. Bobbey", 2,
"explosive");

          dog2.FindSmth(100);     // "nothing"
          dog2.FindSmth(100);     // "nothing"
          dog2.FindSmth(100);     // "nothing"
          dog2.FindSmth(100);     // "nothing"
          dog2.FindSmth(100);     // "nothing"
  dog2.FindSmth(100);     // "explosive!"
      }
  }
```

Тестовая программа «Оркестр» через наследование классов

```
// Базовый абстрактный класс Музыкального Инструмента
abstractclassMusicalInstrument
{
// Свойствотип
publicstring Type { get; set; }

// Конструктор по умолчанию
public MusicalInstrument()
    {
        Type = "Unknown";
    }

// Конструктор с параметром
public MusicalInstrument(string type)
    {
        Type = type;
    }

// Абстрактный метод
publicabstractvoid Play();
}

// Производный класс Гитары
classGuitar : MusicalInstrument
{
// Свойство Количество струн
publicint StringCount { get; set; }

// Конструктор по умолчанию
public Guitar()
    {
        Type = "Spanish guitar";
        StringCount = 6;
    }

// Конструктор с параматерами
public Guitar(string type, int strings) : base(type)
    {
        StringCount = strings;
    }

// Переопределённый метод Play
// Играет музыку гитары
publicoverridevoid Play()
    {
        Console.WriteLine("Jug-jug-strum-neowwwwwhh");
        Console.WriteLine("Wwwaahhhhhwahwahwahwahhhh");
```

```
        Console.WriteLine("Neowwwwwwh... strum");
        Console.WriteLine("Neowh-newh-newh-newh-newh-newwwwhh..");
    }

}

// Производный класс Трубы
classTrumpet : MusicalInstrument
{
// Конструктор по умолчанию
public Trumpet()
    {
        Type = "Piccolo trumpet";
    }

// Конструктор с параметром
public Trumpet(string type) : base(type) { }

// Переопределённый метод Play
// Играет звуки трубы
publicoverridevoid Play()
    {
        Console.WriteLine("Thuuuuuu! Thu! Thu!");
        Console.WriteLine("Pah-pa-pah!");
        Console.WriteLine("Thuuuuuu-rhu! Thu! Thu! Rhuuh!");
    }

}

// Производный класс Барабанов
classDrumKit : MusicalInstrument
{
// Свойство Количество барабанов
publicint Drums { get; set; }

// Конструктор по умолчанию
public DrumKit()
{
        Type = "Four-piece kit";
        Drums = 4;
    }

// Конструктор с параметрами
public DrumKit(string type, int drums) : base(type)
    {
        Drums = drums;
    }

// Переопределённый метод Play
// Издаёт звуки барабанов
publicoverridevoid Play()
    {
```

```
            Console.WriteLine("Rub-a-dub dabum tish!");
            Console.WriteLine("Bum! Brrum! Brrrumble!!!!");
            Console.WriteLine("Pump-a-rum-parum Pump-a-rum! Tish! ");
        }
    }

    // Класс Музыкант
    classMusician
    {
    // Свойста Имя, Опыт и Музыкальный инструмент
    publicstring Name { get; privateset; }
    publicint Experience { get; set; }
    public MusicalInstrument Instrument { get; set; }

    // Конструктор с параметрами
    public Musician(string name, int exp, MusicalInstrument instrum)
        {
    this.Name = name;
    this.Experience = exp;
    this.Instrument = instrum;
        }

    // Метод "Исполнять музыку"
    publicvoid PerformMusic()
        {
    // Вызывает метод Play для его текущего инструмента
    this.Instrument.Play();
        }

    }

    classProgram
    {
    staticvoid Main(string[] args)
        {
    // Создаёминструменты
            Guitar guitar = new Guitar();
            Trumpet trumpet = new Trumpet("Baroque trumpet");
            DrumKit drumKit = new DrumKit("Electronic drums", 8);

    // Создаёммузыканта
            Musician musician = new Musician("Robert", 12, guitar);
            ConsoleKeyInfo key;
    do
            {
                Console.WriteLine("Choose an instrument or start playing
music!");
                Console.WriteLine("  1 - Select Guitar");
                Console.WriteLine("  2 - Select Trumpet");
                Console.WriteLine("  3 - Select DrumKit");
                Console.WriteLine("  Other - Start Playing");
                Console.WriteLine("  ESC - Exit");
                key = Console.ReadKey(true);
```

```
  // Выбор инструмента
  switch (key.Key)
              {
  // В зависимости от кейса даём музыканту разные инструменты
  case ConsoleKey.D1:
                      musician.Instrument = guitar;
  break;
  case ConsoleKey.D2:
                      musician.Instrument = trumpet;
  break;
  case ConsoleKey.D3:
                      musician.Instrument = drumKit;
  break;
  case ConsoleKey.Escape:
  break;
  default:
                      Console.WriteLine("To stop playing music press any
key.");
  do
                      {
  // Простовызываемметод PerformMusic
                          musician.PerformMusic();
                          System.Threading.Thread.Sleep(1000);
                      }
  while (!Console.KeyAvailable);
  // Пока не будет нажата любая клавиша
  Console.ReadKey(true);
  break;
              }
              Console.Clear();
          }
  while (key.Key != ConsoleKey.Escape);
  // Заканчиваем программу по нажатию ESC
      }
  }
```

Программа «Зверинец» через наследование классов

```
// Класс Пища
abstractclassFeed
{
      // Свойства
      publicstring Kind { get; set; }
      publicstring Composition { get; set; }
      publicint KKal { get; set; }
}

// Класс Трава
classGrass : Feed
{
      publicstring Freshness { get; set; }
      publicstring Dryness { get; set; }
}

// Класс Орех
classNut : Feed
{
      publicint Strength { get; set; }
}

// Класс Листья
classLeaves : Feed
{
      publicint Number { get; set; }
}

abstractclass Animal
{
      publicabstractstring Specie { get; protectedset; }
      publicabstractstring Name { get; set; }

      protected Random gen;

      protectedstaticstring Voice { get; set; }

      publicint Age { get; set; }

      public Animal()
      {
            gen = new Random();
      }

      // Абстрактный метод Breeding
      publicabstractbool Breeding();

      // Абстрактный метод GetCub возвращает объект класса Animal
      publicabstract Animal GetCub();
```

```
        // Виртуальный метод Eat принимает объект класса Feed
        publicvirtualvoid Eat(Feed feed)
        {
                Console.WriteLine($"{Specie}{Name} eat {feed.Kind}");
        }

        publicstring MadeSound()
        {
                Console.WriteLine($"{Voice} - {Voice}!");
                return Voice;
        }


  }

  classTurtle : Animal
  {
        // Закрытое поле - булевый флаг, используется в методах для контроля
соответствия размножения-детёнышей
        privatebool canGetChild;

        privatestring specie;
        publicoverridestring Specie
        {
                get
                {
                        return specie;
                }
                protectedset
                {
                        if (string.Compare(value, "Green Turtle", true) == 0 ||
                                string.Compare(value, "Flatback", true) == 0 ||
                                string.Compare(value, "Loggerhead", true) == 0)
                        {
                                specie = value;
                        }
                }
        }

        publicoverridestring Name { get; set; }

        static Turtle()
        {
                Voice = "....";
        }

        // Конструктор по умолчанию
        public Turtle()
        {
                canGetChild = false;
                Specie = "turtle";
        }

        // Переопределённый метод Breeding
```

```
        publicoverridebool Breeding()
        {
                int randInt = gen.Next(3);
                if (randInt == 0)
                {
                        // Сшансом 1:3 черепаха откладывает яйцо
                        Console.WriteLine("Turtle {0} laid egg on the
beach!", Name);
                        System.Threading.Thread.Sleep(500);
                        Console.WriteLine("...................");
                        System.Threading.Thread.Sleep(2000);
                        // И переключаем флагна true
                        canGetChild = true;
                        Console.WriteLine("Now you can get turtles child!");
                        returntrue;
                }
                else
                {
                        // 2:3 что яйца отложить не выйдет
                        Console.WriteLine("Turtle {0} failed to lay
eggs...", Name);
                        returntrue;
                }
        }

        // Переопределённый метод GetCub
        publicoverride Animal GetCub()
        {
                // Если было отложено яйцо
                if (canGetChild)
                {
                        canGetChild = false;
                        int rnd = gen.Next(4);
                        if (rnd != 3)
                        {
                                // Выдаём черепашку с шансом 3:4
                                Console.WriteLine("New turtle hatches from
egg!");
                                returnnew Turtle();
                        }
                        else
                        {
                                // 1:4 что черепашки не будет
                                Console.WriteLine("Oh no! Sand snake catch
and kill little turtle!");
                                returnnull;
                        }
                }
                else
                {
                        // Иначе выдаём пустую ссылку null
                        Console.WriteLine("To have a cub, turtle must first
breed!");
                        returnnull;
```

```
                }
        }

        // Переопределённый метод Eat
        publicoverridevoid Eat(Feed feed)
        {
                // Проверям, что тип пищи - трава
                if (feed is Grass)
                {
                        // И тогда вызываем метод Eat базового класса
                        base.Eat(feed);
                }
                else
                {
                        // Если не трава, то черепаха не может съесть это
                        Console.WriteLine("Turtle can't eat this!");
                }
        }

  }

  // КлассУлитки
  classSnail : Animal
  {
        // Булевый флаг для индикации, отложила ли улитка яйца
        privateint breedAttemptLeft;
        // Количество отложенных яиц
        privateint eggsCount;

        publicoverridestring Specie { get; protectedset; }

        publicoverridestring Name { get; set; }

        static Snail()
        {
                Voice = ".................What do you expect to hear from the
snail?";
        }

        // Конструктор по умолчанию
        public Snail()
        {
                // У улитки есть 5 попыток отложить яйца
                breedAttemptLeft = 5;
                // Изначально у неё нет яиц
                eggsCount = 0;
                Specie = "snail";
        }

        // Переопределённый метод Breeding
        publicoverridebool Breeding()
        {
                if (breedAttemptLeft != 0)
                {
```

```
                        // Отнимаем попытку
                        breedAttemptLeft--;
                        var randomGenerator = new Random();
                        int randInt = randomGenerator.Next(6);
                        if (randInt == 0)
                        {
                                // С вероятностью 1:6 она откладывает 0-50 яиц
                                eggsCount = randomGenerator.Next(100);
                                Console.WriteLine("Snail {0} buried {1} eggs
in the ground!", Name, eggsCount);
                                System.Threading.Thread.Sleep(500);
                                Console.WriteLine("...................");
                                System.Threading.Thread.Sleep(2000);
                                Console.WriteLine("Now eggs can hatch!");
                                returntrue;
                        }
                        else
                        {
                                Console.WriteLine("Snail {0} failed to lay
eggs...", Name);
                                returnfalse;
                        }
                }
                else
                {
                        Console.WriteLine("Snail {0} can no longer breed!",
Name);
                        returnfalse;
                }
        }

        // Переопределённый метод GetCub
        publicoverride Animal GetCub()
        {
                // Если есть яйца
                if (eggsCount != 0)
                {
                        // Можно получать маленьких улиточек
                        eggsCount--;
                        Console.WriteLine("{0} eggs left...", eggsCount);
                        Random gen = new Random();
                        int randInt = gen.Next(5);
                        // Если повезёт, т.к. не из каждого яйца они
вылупляются
                        if (randInt == 0)
                        {
                                Console.WriteLine("Ctchhh! There is new
Snail!");
                                returnnew Snail();
                        }
                        else
                        {
                                Console.WriteLine("Ctchhh! But.. Egg is
empty...");
```

```
                                returnnull;
                        }
                }
                else
                {
                        Console.WriteLine("No eggs - no cubs. Sorry.");
                        returnnull;
                }
        }

        // Переопределённый метод Eat
        publicoverridevoid Eat(Feed feed)
        {
                // Если пища это Листья
                if (feed is Leaves)
                {
                        // То можно есть
                        base.Eat(feed);
                }
                else
                {
                        // Иначе не стоит
                        Console.WriteLine("You're kidding! Snail can't eat
this!");
                }
        }
  }
  // Класс Зверинца
  classMenagerie
  {
        // Список животных
        public List<Animal> Animals { get; set; }

        // Индексатор
        public Animal this[int index]
        {
                get { return Animals[index]; }
                set { Animals[index] = value; }
        }

        // Конструктор для инициализации списка
        public Menagerie()
        {
                Animals = new List<Animal>();
        }

        // Метод для добавления нового животного в зверинец
        publicvoid AddNewAnimal(Animal a)
        {
                Animals.Add(a);
        }

        // Метод для кормления животного
        publicvoid FeedAnimal(Animal a, Feed f, int quantity = 1)
```

```
        {
                // Ищем это животное в нашем зверинце
                int index = Animals.IndexOf(a);
                // Если оно присутствует
                if (index != -1)
                {
                        // Кормим quantity раз
                        for (int i = 0; i < quantity; ++i)
                        {
                                Animals[index].Eat(f);
                        }
                }
        }

        // Метод для размножения случайного животного в зверинце
        public Animal RandomBreed()
        {
                Random random = new Random();
                int i = random.Next(Animals.Count);
                // Выбираем случайное животное
                Animal animal = Animals[i];
                // Вызываем метод размножения
                animal.Breeding();
                // Возвращаем животное, которое размножалось
                return animal;
        }
}

classProgram
{
        staticvoid Main(string[] args)
        {
                // Черепаха Боб
                Turtle t = new Turtle();
                t.Name = "Bob";

                // Улитка Мэгги
                Snail s = new Snail();
                s.Name = "Maggy";

                // Улитка Маркус
                Snail ss = new Snail();
                ss.Name = "Markus";

                // Создаём зверинец
                Menagerie mg = new Menagerie();
                // Забрасываем в него нашу живность
                mg.AddNewAnimal(t);
                mg.AddNewAnimal(s);
                mg.AddNewAnimal(ss);

                // Создаём листья
                Leaves leaves = new Leaves();
                leaves.Kind = "Palm leaves";
```

```
                // Кормим Маркуса 5 листьями
                mg.FeedAnimal(ss, leaves, 5);
                // Кормим Боба листиком
                mg.FeedAnimal(t, leaves);

                string[] names = { "Austin", "Brock", "Caleb", "Dominic",
"Emmett", "Finn", "Guy", "Hogan", "Irving",
                "Jace", "Kingston", "Lambert", "Mickey", "Nolan", "Oliver",
"Parker", "Quinn", "Ramsey", "Samson",
                "Thomas", "Upton", "Vinny", "Wade", "Xander", "York", "Zane" };

                Random gen = new Random();
                ConsoleKeyInfo key;
                // По циклу до нажатия ESC
                do
                {
                        Console.WriteLine();
                        Console.WriteLine("There are {0} animals in the
menagerie", mg.Animals.Count);
                        Console.WriteLine();
                        key = Console.ReadKey(true);
                        // Вызываем метод случайного размножения
                        Animal someAnimal = mg.RandomBreed();
                        // Пытаемся получить детёныша от размножающегося
животного
                        Animal cub = someAnimal.GetCub();
                        // Если детёныш есть
                        if (cub != null)
                        {
                                int randomNameIndex =
gen.Next(names.Length);
                                cub.Name = names[randomNameIndex];
                                Console.WriteLine();
                                Console.WriteLine("Ta-dam! We have
replenishment of the family!");
                                Console.WriteLine("It is a {0} {1}",
cub.Specie, cub.Name);
                                mg.AddNewAnimal(cub);
                        }
                }
                while (key.Key != ConsoleKey.Escape);
        }
  }
```

Дополненная программа «Зверинец» через интерфейсы

```
// Вместо базового класса Пища - интерфейс Съестное
interfaceIEatable
{
string Composition { get; set; }
int KKal { get; set; }
}

// Трава реализует интерфейс съестное
classGrass : IEatable
{
publicstring Freshness { get; set; }
publicstring Dryness { get; set; }
publicstring Composition { get; set; }
publicint KKal { get; set; }

publicoverridestring ToString()
    {
return"Grass";
    }
}

// Орех реализует интерфейс съестное
classNut : IEatable
{
publicstring Composition { get; set; }
publicint KKal { get; set; }
publicint Strength { get; set; }

publicoverridestring ToString()
    {
return"Nut";
    }
}

// Листья реализуют интерфейс съестное
classLeaves : IEatable
{
publicint Number { get; set; }
publicstring Composition { get; set; }
publicint KKal { get; set; }

publicoverridestring ToString()
    {
return"Leaves";
}
}

// Теперь у нас есть интерфейс, описывающий как быть животным
interfaceIAnimal
```

```
  {
  string Specie { get; set; }
  string Name { get; set; }
  int Age { get; set; }
      Random Gen { get; }

  bool Breeding();
      IAnimal GetCub();
  bool Eat(IEatable feed);
  string MadeSound();
  string Print();
  }

  // Черепаха вместо наследования класса реализует интерфейс
  classTurtle : IAnimal
  {
  // Закрытое поле - булевый флаг, используется в методах для контроля
соответствия размножения-детёнышей
  privatebool canGetChild;

  privatestring specie;
  publicstring Specie
      {
  get
          {
  return specie;
          }
  set
          {
  if (string.Compare(value, "Green Turtle", true) == 0 ||
  string.Compare(value, "Flatback", true) == 0 ||
  string.Compare(value, "Loggerhead", true) == 0)
              {
                  specie = value;
              }
          }
      }

  publicstring Name { get; set; }
  publicint Age { get; set; }

  // Интересный способ инициализации
  private Random gen;
  public Random Gen
      {
  get
          {
  if (gen == null)
              {
                  gen = new Random();
              }
  return gen;
          }
      }
```

```
public Turtle()
    {
        canGetChild = false;
        Specie = "Green Turtle";
}

// Реализация метода Breeding из интерфейса IAnimal
publicbool Breeding()
    {
int randInt = Gen.Next(3);
if (randInt == 0)
        {
// С шансом 1:3 черепаха откладывает яйцо
Console.WriteLine("Turtle {0} laid egg on the beach!", Name);
            System.Threading.Thread.Sleep(500);
            Console.WriteLine("...................");
            System.Threading.Thread.Sleep(2000);
// И переключаем флаг на true
            canGetChild = true;
            Console.WriteLine("Now you can get turtles child!");
returntrue;
        }
else
        {
// 2:3 что яйца отложить не выйдет
Console.WriteLine("Turtle {0} failed to lay eggs...", Name);
returntrue;
        }
    }

// Реализация метода GetCub из интерфейса IAnimal
public IAnimal GetCub()
    {
// Если было отложено яйцо
if (canGetChild)
        {
            canGetChild = false;
int rnd = Gen.Next(2);
if (rnd != 0)
            {
// Выдаём черепашку с шансом 1:2
Console.WriteLine("New turtle hatches from egg!");
returnnew Turtle();
            }
else
            {
// 1:2 что черепашки не будет
Console.WriteLine("Oh no! Sand snake catch and kill little turtle!");
returnnull;
            }
        }
else
        {
```

```
// Иначе выдаём пустую ссылку null
            Console.WriteLine("To have a cub, turtle must first breed!");
returnnull;
        }
    }

// Реализация метода Eat изинтерфейса IAnimal
// Принимает корм в виде интерфейса IEatable
publicbool Eat(IEatable feed)
    {
// Проверям, что тип пищи - трава
if (feed is Grass || feed is Snail)
        {
            Console.WriteLine("{0} {1} eats {2}", Specie, Name, feed);
returntrue;
        }
else
        {
// Если не трава, то черепаха не может съесть это
Console.WriteLine("Turtle can't eat {0}!", feed);
returnfalse;
        }
    }

// Реализация метода MadeSound из интерфейса IAnimal
publicstring MadeSound()
    {
var turtleVoice = ".......";
        Console.WriteLine(turtleVoice);
return turtleVoice;
}

// Реализация метода Pring из интерфейса IAnimal
publicstring Print()
    {
string ret = Specie + " " + Name;
// Будет выводить информацию зелёненьким
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine(ret);
        Console.ResetColor();
return ret;
    }
}

// Класс улитки реализует интерфейса IAnimal - значит она животное
// Также улитка может быть съеденой - она реализует интерфейс IEatable
classSnail : IAnimal, IEatable
{
// Булевый флаг для индикации, отложила ли улитка яйца
privateint breedAttemptLeft;
// Количество отложенных яиц
privateint eggsCount;

publicstring Specie { get; set; }
```

```
  publicstring Name { get; set; }
  publicint Age { get; set; }

  private Random gen;
  public Random Gen
      {
  get
          {
  if (gen == null)
              {
                  gen = new Random();
              }
  return gen;
          }
      }
  // Конструкторпоумолчанию
  public Snail()
  {
  // У улитки есть 5 попыток отложить яйца
          breedAttemptLeft = 5;
  // Изначально у неё нет яиц
          eggsCount = 0;
          Specie = "Snail";
      }

  // Реализация метода Breeding из интерфейса IAnimal
  publicbool Breeding()
      {
  if (breedAttemptLeft != 0)
          {
  // Отнимаем попытку
              breedAttemptLeft--;
  int randInt = Gen.Next(5);
  if (randInt == 0 || randInt == 1)
              {
  // С вероятностью 2:5 она откладывает 0-50 яиц
  int eggs = Gen.Next(50);
                  Console.WriteLine("Snail {0} buried {1} eggs in the
ground!", Name, eggs);
                  eggsCount += eggs;
                  System.Threading.Thread.Sleep(500);
                  Console.WriteLine("...................");
                  System.Threading.Thread.Sleep(2000);
                  Console.WriteLine("Now eggs can hatch!");
  returntrue;
              }
  else
              {
                  Console.WriteLine("Snail {0} failed to lay eggs...", Name);
  returnfalse;
              }
          }
  else
          {
```

```
if (eggsCount == 0)
            {
                Console.WriteLine("Snail {0} can no longer breed!", Name);
}
returnfalse;
        }
    }

// Реализация метода GetCub из интерфейса IAnimal
public IAnimal GetCub()
    {
// Если есть яйца
if (eggsCount != 0)
        {
// Можно получать маленьких улиточек
eggsCount--;
            Console.WriteLine("{0} eggs left...", eggsCount);
int randInt = Gen.Next(5);
// Если повезёт, т.к. не из каждого яйца они вылупляются
if (randInt == 0)
            {
                Console.WriteLine("Ctchhh! There is new Snail!");
returnnew Snail();
            }
else
            {
                Console.WriteLine("Ctchhh! But.. Egg is empty...");
returnnull;
            }
        }
else
        {
            Console.WriteLine("No eggs - no cubs. Sorry.");
returnnull;
        }
    }

// Реализация метода Eat из интерфейса IAnimal
publicbool Eat(IEatable feed)
    {
// Если пища это Листья
if (feed is Leaves)
        {
// То можно есть
            Console.WriteLine("{0} {1} eats {2}", Specie, Name, feed);
returntrue;
        }
else
        {
// Иначе не стоит
            Console.WriteLine("You're kidding! Snail can't eat {0}!", feed);
returnfalse;
}
    }
```

```
// Реализация метода MadeSound из интерфейса IAnimal
publicstring MadeSound()
    {
var snailSong = "Two little eyes, one and two.\n" +
"One pretty shell." +
"Nice to meet you." +
"He has no legs, just a tail." +
"It's a snail, it's a snail, it's a snail.";
        Console.WriteLine(snailSong);
return snailSong;
}

// Реализация метода Pring из интерфейса IAnimal
publicstring Print()
    {
string ret = Specie + " " + Name;
// Будет выводить информацию красненьким
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(ret);
        Console.ResetColor();
return ret;
    }

// Реализация интерфейса IEatable в классе Улитки
publicstring Composition { get; set; }
publicint KKal { get; set; }

publicoverridestring ToString()
    {
return Specie;
}
}

// Класс Зверинца
classMenagerie
{
// Список животных, теперь через интерфейс
public List<IAnimal> Animals { get; set; }

// Индексатор
public IAnimal this[int index]
    {
get { return Animals[index]; }
set { Animals[index] = value; }
}

// Конструктор для инициализации списка
public Menagerie()
    {
        Animals = new List<IAnimal>();
    }

// Метод для добавления нового животного в зверинец
```

```
  publicvoid AddNewAnimal(IAnimal a)
  {
          Animals.Add(a);
      }

  // Метод для кормления кого-то, кто реализует интерфейс IAnimal,
  // чем-то, что реализует интерфейс IEatable
  publicvoid FeedAnimal(IAnimal a, IEatable f, int quantity = 1)
  {
  // Ищем это животное в нашем зверинце
  int index = Animals.IndexOf(a);
  // Если оно присутствует
  if (index != -1)
          {
  // Если корм - это другое животное
  if (f is IAnimal liveFeed)
  {
  // Пытаемся скормить
  bool itIsEated = Animals[index].Eat(f);
  // Если животное успешно съели, и оно было в нашем зверинце (было, потому
что здесь же оно и удаляется)
  if (itIsEated && Animals.Remove(liveFeed))
                  {
  // Ругаем проектировщиков такой системы
                      Console.WriteLine("\nWhat are you do! It was " +
liveFeed.Name + "!!\n");
  }

              }
  // Иначе можно не волноваться
  else
              {
  // Кормим quantity раз
  for (int i = 0; i < quantity; ++i)
  {
                      Animals[index].Eat(f);
                  }
              }
          }
      }

  // Метод для размножения случайного животного в зверинце
  public IAnimal RandomBreed()
      {
          Random random = new Random();
  int i = random.Next(Animals.Count);
  // Выбираем случайное животное
          IAnimal animal = Animals[i];
  // Вызываем метод размножения
          animal.Breeding();
  // Возвращаем животное, которое размножалось
  return animal;
      }
```

```
  // Метод для вывода на консоль пронумерованного списка животных в зверинце
  publicvoid ShowAnimals()
      {
          Console.WriteLine();
  for (int i = 1; i < Animals.Count + 1; ++i)
          {
              Console.Write(" {0,2}. ", i);
  // С применением метода IAnimal.Print
              Animals[i - 1].Print();
          }
          Console.WriteLine();
      }
  }

  classProgram
  {
  staticvoid Main(string[] args)
      {
  // Черепаха Боб
          Turtle t = new Turtle();
          t.Name = "Bob";

  // Улитка Мэгги
          Snail s = new Snail();
          s.Name = "Maggy";

  // Улитка Маркус
          Snail ss = new Snail();
          ss.Name = "Markus";

  // Создаём зверинец
          Menagerie mg = new Menagerie();
  // Забрасываем в него нашу живность
          mg.AddNewAnimal(t);
  mg.AddNewAnimal(s);
          mg.AddNewAnimal(ss);

  // Создаём листья
          Leaves leaves = new Leaves();
  // Кормим Маркуса 5 листьями
          mg.FeedAnimal(ss, leaves, 5);
  // Кормим Боба листиком
          mg.FeedAnimal(t, leaves);

  // Пытаемся скормить Маркуса Мэгги
          mg.FeedAnimal(s, ss);

  // Отправляем Маркуса на закуску Бобу
          mg.FeedAnimal(t, ss);


  string[] names = { "Austin", "Brock", "Caleb", "Dominic", "Emmett",
"Finn", "Guy", "Hogan", "Irving",
```

```
  "Jace", "Kingston", "Lambert", "Mickey", "Nolan", "Oliver", "Parker",
"Quinn",
  "Ramsey", "Samson", "Thomas", "Upton", "Vinny", "Wade", "Xander", "York",
"Zane" };

          Random gen = new Random();
          ConsoleKeyInfo key;
          Console.WriteLine("Press ESC to Exit, SPACE to Show animals in the
menagerie, ANY KEY to breeding attempt:");
  // По циклу до нажатия ESC
  do
  {
              Console.WriteLine();
              Console.WriteLine("There are {0} animals in the menagerie",
mg.Animals.Count);
              Console.WriteLine();
              key = Console.ReadKey(true);
  // На пробел
  switch (key.Key)
              {
  case ConsoleKey.Spacebar:
  // Вызываемметодзверинца ShowAnimals
  mg.ShowAnimals();
  break;
  case ConsoleKey.Escape:
  break;
  default:
  // Вызываем метод случайного размножения
                      IAnimal someAnimal = mg.RandomBreed();
  // Пытаемся получить детёныша от размножающегося животного
  IAnimal cub = someAnimal.GetCub();
  // Если детёныше сть
  if (cub != null)
                      {
  // Выбираем ему случайное имя из массива имён names
  int randomNameIndex = gen.Next(names.Length);
                          cub.Name = names[randomNameIndex];
                          Console.WriteLine();
                          Console.WriteLine("Ta-dah! We have replenishment
of the family!");
  // Добавляем его в зверинец
                          mg.AddNewAnimal(cub);
  Console.Write("It is a ");
                          cub.Print();
                      }
  break;
              }

          }
  while (key.Key != ConsoleKey.Escape);
      }
  }
```

Тестовая программа «Оркестр» через интерфейсы

```
// Интерфейс Создатель музыки
interfaceIMusicMaker
{
string Type { get; set; }
void Play();
}

// Класс Гитара, реализует интерфейс Создатель музыки
classGuitar : IMusicMaker
{
// Свойство Количество струн
publicint StringCount { get; set; }

// Свойство тип – из интерфейса IMusicMaker
publicstring Type { get; set; }

// Конструктор по умолчанию
public Guitar()
{
        Type = "Spanish guitar";
        StringCount = 6;
    }

// Конструктор с параметрами
public Guitar(string type, int strings)
    {
        Type = type;
        StringCount = strings;
    }

// Реализация метода Play из интерфейса
// Играет музыку гитары
publicvoid Play()
    {
        Console.WriteLine("Jug-jug-strum-neowwwwwhh");
Console.WriteLine("Wwwaahhhhhwahwahwahhhh");
        Console.WriteLine("Neowwwwwwh... strum");
        Console.WriteLine("Neowh-newh-newh-newh-newh-newwwwhh..");
}

}

// Класс Трубы, реализует интерфейс Создателя музыки
classTrumpet : IMusicMaker
{
// Свойство Тип из интерфейса IMusicMaker
publicstring Type { get; set; }
```

```
// Конструктор по умолчанию
public Trumpet()
    {
        Type = "Piccolo trumpet";
    }

// Конструктор с параметром
public Trumpet(string type)
    {
        Type = type;
}

// Реализация метода Play из интерфейса
// Играет звуки трубы
publicvoid Play()
    {
        Console.WriteLine("Thuuuuuu! Thu! Thu!");
        Console.WriteLine("Pah-pa-pah!");
        Console.WriteLine("Thuuuuuu-rhu! Thu! Thu! Rhuuh!");
    }

}

// Класс Барабанов, реализует интерфейс Создателя музыки
classDrumKit : IMusicMaker
{
// Свойство Количество барабанов
publicint Drums { get; set; }
// Свойство Тип из интерфейса IMusicMaker
publicstring Type { get; set; }

// Конструктор по умолчанию
public DrumKit()
{
        Type = "Four-piece kit";
        Drums = 4;
    }

// Конструктор с параметрами
public DrumKit(string type, int drums)
    {
        Type = type;
        Drums = drums;
    }

// Реализация метода Play из интерфейса
// Издаёт звуки барабанов
publicvoid Play()
    {
        Console.WriteLine("Rub-a-dub dabum tish!");
        Console.WriteLine("Bum! Brrum! Brrrumble!!!!");
        Console.WriteLine("Pump-a-rum-parum Pump-a-rum! Tish! ");
```

```
    }
}

// Интерфей сМузыканта
interfaceIMusicPlayer
{
    IMusicMaker Instrument { get; set; }
void PerformMusic();
}
// Класс Музыкант
classMusician : IMusicPlayer
{
// Свойства Имя, Опыт
publicstring Name { get; privateset; }
publicint Experience { get; set; }

// Свойство Инструмент теперь от интерфейса IMusicPlayer
public IMusicMaker Instrument { get; set; }

// Конструктор с параметрами
public Musician(string name, int exp, IMusicMaker instrum)
    {
this.Name = name;
this.Experience = exp;
this.Instrument = instrum;
    }

// Метод "Исполнять музыку" от интерфейса IMusicPlayer
publicvoid PerformMusic()
    {
// Вызывает метод Play для его текущего инструмента
this.Instrument.Play();
    }

}

// Класс Воробей, который реализует интерфейс Исполнителя музыки
classSparrow : IMusicPlayer
{
// Свойство Гнездо
publicstring Nest { get; set; }
// Свойство Вид
publicstring Kind { get; set; }

// Реализация интерфейса Исполнителя музыки
public IMusicMaker Instrument { get; set; }

publicvoid PerformMusic()
    {
if (Instrument is Voice voice)
        {
for (int i = 0; i < 3; ++i)
            {
```

```
                voice.SetVoice();
                voice.Play();
            }
        }
    }
}

// Класс Voice, реализующий интерфейс IMusicMaker
classVoice : IMusicMaker
{
// Свойство Type из интерфейса IMusicMaker
publicstring Type { get; set; }

// Свойство - список издаваемых звуков
private List<string> voices;

// Конструктор
public Voice()
    {
        voices = new List<string>();
    }

// Метод для задания звуков
publicvoid SetVoice(paramsstring[] voices)
    {
this.voices = new List<string>(voices);
    }

// Метод Play из интерфейса IMusicMaker
publicvoid Play()
{
        Random randGen = new Random();
// Выбираем случайным образом количество повторений
int voiceIndex, num = randGen.Next(1, voices.Count + 1);
for (int i = 0; i < num; ++i)
        {
// Выбираем случайный звук из списка
            voiceIndex = randGen.Next(voices.Count);
            Console.WriteLine(voices[voiceIndex]);
        }
        Console.WriteLine();
    }
}

classProgram
{
staticvoid Main(string[] args)
    {
// Создаём инструменты
        Guitar guitar = new Guitar();
        Trumpet trumpet = new Trumpet("Baroque trumpet");
        DrumKit drumKit = new DrumKit("Electronic drums", 8);
```

```
// Создаём музыканта
        Musician musician = new Musician("Robert", 12, guitar);
        ConsoleKeyInfo key;
do
        {
            Console.WriteLine("Choose an instrument or start playing music!");
            Console.WriteLine("  1 - Select Guitar");
            Console.WriteLine("  2 - Select Trumpet");
            Console.WriteLine("  3 - Select DrumKit");
            Console.WriteLine("  Other - Start Playing");
            Console.WriteLine("  ESC - Exit");
            key = Console.ReadKey(true);
// Выбор инструмента
switch (key.Key)
            {
// В зависимости от кейса даём музыканту разные инструменты
case ConsoleKey.D1:
                    musician.Instrument = guitar;
break;
case ConsoleKey.D2:
                    musician.Instrument = trumpet;
break;
case ConsoleKey.D3:
                    musician.Instrument = drumKit;
break;
case ConsoleKey.Escape:
break;
default:
                    Console.WriteLine("To stop playing music press any key.");
do
                    {
// Просто вызываем метод PerformMusic
                        musician.PerformMusic();
                        System.Threading.Thread.Sleep(1000);
                    }
while (!Console.KeyAvailable);
// Пока не будет нажата любая клавиша
Console.ReadKey(true);
break;
            }
            Console.Clear();
        }
while (key.Key != ConsoleKey.Escape);
// Заканчиваем программу по нажатию ESC
    }
}
```

ИСПОЛЬЗУЕМАЯ ЛИТЕРАТУРА

1. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5 / Пер. с англ. – 6-е изд. – М. : Вильямс, 2013. – 1312 с.

2. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# / Пер. с англ. – 4-е изд., исправ. – М. : Издетельство «Русская редакция»; СПб. : Питер, 2019. – 898 с.

3. OOP Principles, https://www.slideshare.net/ssuser8f1bf3/oop-principles-37419002 – [Электронный ресурс], Дата доступа: 17.10.2018.

4. Язык программирования C# и .NET, https://metanit.com/sharp/general.php – [Электронный ресурс], Дата доступа: 03.02.2019.

5. C# 5.0 и платформа .NET 4.5, https://professorweb.ru/my/csharp/charp_theory/level1/infocsharp.php, [Электронный ресурс], Дата доступа: 24.02.2019.

Учебное издание

Карпович Павел Сергеевич

C# в примерах

Практикум

В двух частях.
Часть 1

Редактор *Д .С. Ковалевский*
Редактор технический *Ю. Э. Недбальская*
Компьютерная верстка *Ю. Э. Недбальская*
Корректор *Д. С. Ковалевский*

Подписано в печать16.12.2019. Формат 60×84 1/16. Бумага офсетная
Ризография. Усл. печ. л. 8,37. Уч.-изд. л. 4,39.
Тираж 50 экз. Заказ 667.

Выпущено по заказу ГУО «Республиканский институт
повышения квалификации и переподготовки работников
Министерства труда и социальной защиты Республики Беларусь».

Издатель и полиграфическое исполнение:
учреждение образования
«Минский государственный ПТК полиграфии».

Свидетельство о государственной регистрации издателя,
изготовителя и распространителя печатных изданий
№ 1/129 от 27.12.2013.

Ул. В. Хоружей, 7, 220005, г. Минск.