## **Структура программы**

### Основное:

1. **Пространства имен (namespace):**
   * Используются для организации кода и избежания конфликтов имен.
   * Пример: namespace MyApp { ... }
2. **Классы:**
   * Основная структурная единица программ на C#.
   * Пример: class Program { ... }
3. **Метод** Main**:**
   * Точка входа в программу.
   * Обязательно должен быть в одном из классов.
   * Пример:

static void Main(string[] args)

{

Console.WriteLine("Hello, World!");

}

1. **Вывод на консоль:**
   * Используется метод Console.WriteLine.
   * Пример: Console.WriteLine("Привет, мир!");
2. **Ключевые слова:**
   * using – для подключения пространств имен.
   * static – делает метод доступным без создания экземпляра класса.
   * void – указывает, что метод ничего не возвращает.

### Минимальный пример программы:

using System;

namespace MyApp

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Hello, World!");

}

}

}

Статья объясняет, как эти элементы взаимодействуют друг с другом, чтобы создать простую программу.

### **Переменные и константы в C#**

#### **Переменные**

1. **Определение:**
   * Переменная — это именованная область памяти для хранения данных определённого типа.
   * Имеет три характеристики: **тип**, **имя** и **значение**.
2. **Синтаксис:**

тип имя\_переменной;

1. **Требования к имени переменной:**
   * Имя начинается с буквы или символа \_.
   * Может содержать буквы, цифры, символ \_, но без пробелов и знаков пунктуации.
   * Не должно совпадать с ключевыми словами языка.
2. **Инициализация:**
   * Можно присвоить значение сразу при определении:

string name = "Tom";

* + Или после определения:

string name;

name = "Tom";

1. **Особенности:**
   * C# — регистрозависимый язык: name и Name — разные переменные.
   * Значение переменной можно менять в ходе выполнения программы:

string name = "Tom";

name = "Bob";

#### **Константы**

1. **Определение:**
   * Константы — это неизменяемые значения, задаваемые при их создании.
2. **Синтаксис:**
   * Используется ключевое слово const перед типом:

const string NAME = "Tom";

1. **Особенности:**
   * Обязательно задавать значение при объявлении.
   * Константу нельзя изменить после определения.

const string NAME = "Tom";

NAME = "Bob"; // Ошибка

1. **Рекомендация:**
   * Для имён констант часто используется верхний регистр (например, NAME).

#### **Основное различие:**

* Если значение данных может изменяться — используем переменные.
* Если значение должно быть неизменным — используем константы.

### **Литералы в C#**

Литералы — неизменяемые значения, используемые в коде. Они бывают различных типов: логические, целочисленные, вещественные, символьные, строковые и null.

**Логические литералы:**  
Представляют истину или ложь. Имеют два значения: true и false.  
Пример:

Console.WriteLine(true); // true

Console.WriteLine(false); // false

**Целочисленные литералы:**  
Представляют целые числа (положительные и отрицательные).  
Форматы:

* **Десятичная система** (обычные числа):

Console.WriteLine(5); // 5

* **Двоичная система** (начинается с 0b):

Console.WriteLine(0b1011); // 11

* **Шестнадцатеричная система** (начинается с 0x):

Console.WriteLine(0xFF); // 255

**Вещественные литералы:**  
Представляют дробные числа.  
Форматы:

* С фиксированной запятой:

Console.WriteLine(3.14); // 3.14

* В экспоненциальной форме MЕp (M \* 10^p):

Console.WriteLine(3.2e3); // 3200

Console.WriteLine(1.2E-1); // 0.12

**Символьные литералы:**  
Представляют одиночные символы, заключённые в одинарные кавычки:

Console.WriteLine('A'); // A

Специальные управляющие последовательности:

* \n — перевод строки
* \t — табуляция
* \\ — слеш  
  Пример:

Console.WriteLine("Привет\nмир");

Символы можно задавать через:

* Шестнадцатеричные ASCII-коды: '\x78' (например, x).
* Unicode-коды: '\u0411' (например, Б).

**Строковые литералы:**  
Строки заключаются в двойные кавычки:

Console.WriteLine("Hello, World!");

Внутренние кавычки экранируются:

Console.WriteLine("Компания \"Рога и копыта\"");

Поддерживают управляющие последовательности (например, \n).

**Литерал** null**:**  
Обозначает отсутствие значения или объекта:

string name = null;

Литералы помогают задавать фиксированные значения в программе и активно используются при разработке.

### **Типы данных в C#**

В C# типы данных определяют, какие значения может хранить переменная и какие операции можно над ней выполнять. Базовые типы данных:

**Логический тип**

* bool: хранит true или false.  
  Пример:

bool alive = true;

bool isDead = false;

**Целочисленные типы**

* byte: от 0 до 255 (1 байт).
* sbyte: от -128 до 127 (1 байт).
* short: от -32 768 до 32 767 (2 байта).
* ushort: от 0 до 65 535 (2 байта).
* int: от -2 147 483 648 до 2 147 483 647 (4 байта).
* uint: от 0 до 4 294 967 295 (4 байта).
* long: от –9×10¹⁸ до 9×10¹⁸ (8 байт).
* ulong: от 0 до 18×10¹⁸ (8 байт).

Пример:

int a = 10;

long b = 123456789L;

**Типы с плавающей точкой**

* float: от -3.4×10³⁸ до 3.4×10³⁸ (4 байта), используется суффикс F.
* double: от ±5.0×10⁻³²⁴ до ±1.7×10³⁰⁸ (8 байт).
* decimal: для высокоточной арифметики, занимает 16 байт, используется суффикс M.

Пример:

float x = 3.14F;

decimal y = 100.5M;

**Символьный тип**

* char: хранит один символ Unicode (2 байта).  
  Пример:

char letter = 'A';

char code = '\u0041'; // 'A'

**Строковый тип**

* string: хранит набор символов Unicode.  
  Пример:

string hello = "Hello, World!";

**Универсальный тип**

* object: базовый тип для всех данных, может хранить значение любого типа.

Пример:

object a = 42;

object b = "Hello";

**Интерполяция строк**  
Позволяет вставлять значения переменных в строки:

string name = "Tom";

int age = 33;

Console.WriteLine($"Имя: {name}, Возраст: {age}");

**Неявная типизация**

* Используется ключевое слово var. Тип переменной выводится компилятором на основе присвоенного значения.  
  Пример:

var number = 10; // int

var text = "Hello"; // string

* Нельзя использовать без инициализации или присваивать null.

**Суффиксы для литералов**

* F/f — для float: float num = 3.14F;
* M/m — для decimal: decimal num = 100.5M;
* U/u, L/l, UL/ul — для uint, long, ulong.

**Системные типы**  
Каждый встроенный тип имеет эквивалент в .NET:

* int = System.Int32
* string = System.String  
  Пример:

int x = 10;

System.Int32 y = 10; // эквивалентно

### **Консольный ввод-вывод в C#**

#### **Вывод на консоль**

1. Console.WriteLine  
   Вывод строки с переходом на новую строку:

Console.WriteLine("Добро пожаловать в C#!");

1. **Интерполяция строк**  
   Использование переменных в строках:

string name = "Tom";

int age = 34;

Console.WriteLine($"Имя: {name} Возраст: {age}");

1. **Форматирование строк с плейсхолдерами**  
   Замена {0}, {1}, {2} значениями:

Console.WriteLine("Имя: {0} Возраст: {1}", name, age);

1. Console.Write  
   Вывод без перехода на новую строку:

Console.Write("Введите имя: ");

#### **Ввод с консоли**

1. Console.ReadLine  
   Считывает строку с консоли:

Console.Write("Введите имя: ");

string? name = Console.ReadLine();

Console.WriteLine($"Привет, {name}!");

1. **Считывание данных других типов**  
   Используются методы преобразования:
   * Convert.ToInt32() — к int
   * Convert.ToDouble() — к double
   * Convert.ToDecimal() — к decimal

Пример:

Console.Write("Введите возраст: ");

int age = Convert.ToInt32(Console.ReadLine());

Console.Write("Введите рост: ");

double height = Convert.ToDouble(Console.ReadLine());

#### **Особенности**

* Метод Console.ReadLine возвращает null, если нечего считать. Переменные для ввода объявляются как string?.
* **Дробные числа** вводятся с учетом региональных настроек:
  + Разделитель — точка (США) или запятая (Россия).

Пример программы:

Console.Write("Введите имя: ");

string? name = Console.ReadLine();

Console.Write("Введите возраст: ");

int age = Convert.ToInt32(Console.ReadLine());

Console.Write("Введите рост: ");

double height = Convert.ToDouble(Console.ReadLine());

Console.WriteLine($"Имя: {name} Возраст: {age} Рост: {height}м");

### **Арифметические операции в C#**

#### **Бинарные операции**

1. **Сложение** +

int x = 10;

int z = x + 12; // 22

1. **Вычитание** -

int x = 10;

int z = x - 6; // 4

1. **Умножение** \*

int x = 10;

int z = x \* 5; // 50

1. **Деление** /
   * Целые числа: результат округляется до целого.

int x = 10;

int z = x / 4; // 2

* + Дробные числа: результат сохраняет дробную часть.

double z = 10.0 / 4.0; // 2.5

1. **Остаток от деления** %

int x = 10;

int z = x % 4; // 2

#### **Унарные операции**

1. **Инкремент** ++
   * Префиксный: ++x — увеличивает значение до возврата.
   * Постфиксный: x++ — увеличивает значение после возврата.

int x = 5;

int z = ++x; // z = 6, x = 6

int y = x++; // y = 6, x = 7

1. **Декремент** --
   * Префиксный: --x — уменьшает значение до возврата.
   * Постфиксный: x-- — уменьшает значение после возврата.

int x = 5;

int z = --x; // z = 4, x = 4

int y = x--; // y = 4, x = 3

#### **Приоритет операций**

* **От высокого к низкому:**
  1. Инкремент (++), декремент (--)
  2. Умножение (\*), деление (/), остаток от деления (%)
  3. Сложение (+), вычитание (-)
* Для изменения порядка используются скобки:

int d = (10 + 5) \* 2; // 30

#### **Ассоциативность операторов**

* **Левоассоциативные операторы** (по умолчанию): выполняются слева направо.  
  Пример:

int x = 10 / 5 \* 2; // (10 / 5) \* 2 = 4

#### **Примеры**

1. Простой декремент и умножение:

int c = 40;

int b = 5;

int d = c-- - b \* 3; // d = 25, c = 39

1. Изменение порядка операций:

int c = 40;

int b = 5;

int d = (c - (--b)) \* 2; // d = 70, b = 4

## **Поразрядные операции**

Поразрядные операции выполняются над отдельными разрядами чисел, представленными в двоичной системе. Вот основные операции и их примеры:

1. **Логические операции**:
   * **& (логическое И)**: возвращает 1, если оба бита в разряде равны 1, иначе 0.

2 & 5 // 010 & 101 = 000 (0)

* + **| (логическое ИЛИ)**: возвращает 1, если хотя бы один из битов равен 1.

2 | 5 // 010 | 101 = 111 (7)

* + **^ (исключающее ИЛИ - XOR)**: возвращает 1, если биты разные, иначе 0.

45 ^ 102 // 101101 ^ 1100110 = 1001011 (75)

* + **~ (логическое отрицание)**: инвертирует все биты числа.

~12 // 00001100 -> 11110011 (-13)

1. **Представление отрицательных чисел**:
   * В C# используется дополнительный код (two's complement), где старший разряд указывает на знак числа.

int x = 12;

int y = ~x + 1; // -12

1. **Операции сдвига**:
   * **<< (сдвиг влево)**: сдвигает биты числа влево.

4 << 1 // 100 << 1 = 1000 (8)

* + **>> (сдвиг вправо)**: сдвигает биты числа вправо.

16 >> 1 // 10000 >> 1 = 1000 (8)

* + Используются для умножения или деления на 2.

1. **Пример практического применения**:
   * Поразрядные операции позволяют упаковывать несколько чисел в одно для экономии памяти.

int result = 0;

result = result | value1;

result = result << 2;

result = result | value2;

result = result << 2;

result = result | value3;

Console.WriteLine(result); // 57

1. **Восстановление значений**:
   * Для извлечения чисел из упакованного результата используется поразрядная операция и сдвиг:

int newValue3 = result & 0b0000011;

result = result >> 2;

int newValue2 = result & 0b0000011;

result = result >> 2;

int newValue1 = result & 0b0000011;

Таким образом, поразрядные операции обеспечивают эффективную работу с данными на уровне битов, экономя память и ускоряя выполнение операций.

## **Операции присваивания**

Операции присваивания в C# позволяют установить значение переменной. Основная операция — это =, но также существуют и комбинированные операции, которые выполняют операцию и присваивание одновременно. Вот краткое описание:

1. **Базовая операция присваивания**: int x = 23; — присваивает значение 23 переменной x.
2. **Множественное присваивание**: a = b = c = 34; — присваивает значение 34 всем переменным.
3. **Комбинированные операции присваивания**:
   * +=: добавляет и присваивает: A += B эквивалентно A = A + B.
   * -=: вычитает и присваивает: A -= B эквивалентно A = A - B.
   * \*=: умножает и присваивает: A \*= B эквивалентно A = A \* B.
   * /=: делит и присваивает: A /= B эквивалентно A = A / B.
   * %=: находит остаток и присваивает: A %= B эквивалентно A = A % B.
   * &=, |=, ^=: поразрядные операции и присваивание.
   * <<=, >>=: сдвиг разрядов и присваивание.
4. **Приоритет выполнения**: операции присваивания имеют низкий приоритет и выполняются справа налево.

Примеры:

int a = 10;

a += 10; // 20

a -= 4; // 16

a \*= 2; // 32

a /= 8; // 4

a <<= 4; // 64

a >>= 2; // 16

Операции присваивания всегда выполняются справа налево.

## **Преобразования базовых типов данных**

В C# преобразования типов данных бывают двух типов: **сужающие** и **расширяющие**.

1. **Расширяющие преобразования** (widening): тип с меньшей разрядностью преобразуется в тип с большей разрядностью без потери данных, например, byte в ushort.
2. **Сужающие преобразования** (narrowing): тип с большей разрядностью преобразуется в тип с меньшей разрядностью, что может привести к потере данных, например, ushort в byte.

### Явные и неявные преобразования:

* **Неявные** (implicit) происходят автоматически, например, при преобразовании данных с меньшей разрядностью в тип с большей разрядностью (например, byte в int).
* **Явные** (explicit) требуют указания типа вручную через операцию преобразования (тип\_данных), например, (byte)(a + 70).

### Примеры:

* **Неявное преобразование**:

byte a = 4;

ushort b = a; // Компилятор автоматически расширяет тип

* **Явное преобразование**:

int a = 33;

int b = 600;

byte c = (byte)(a + b); // Потеря данных: результат 121, т.к. 633 выходит за пределы byte

### Потеря точности:

При сужающих преобразованиях, например, если сумма чисел выходит за пределы диапазона целевого типа, старшие биты могут быть усечены, что приведет к потере точности.

### Преобразования между типами:

Для безопасных преобразований существует таблица, описывающая, в какие типы можно безопасно преобразовывать данные. Например, int можно безопасно преобразовать в long, float, double, decimal.

### Примечание:

Когда преобразовываются дробные типы, например, double в decimal, требуется явное преобразование, чтобы избежать потери точности.

## **Условные выражения**

В C# операции, возвращающие логическое значение типа bool, включают операции сравнения и логические операции.

### Операции сравнения:

1. ==: Проверяет равенство. Возвращает true, если операнды равны, иначе — false.
2. !=: Проверяет неравенство. Возвращает true, если операнды не равны, иначе — false.
3. <: Проверяет, меньше ли первый операнд второго.
4. >: Проверяет, больше ли первый операнд второго.
5. <=: Проверяет, меньше или равно ли значение первого операнда второму.
6. >=: Проверяет, больше или равно ли значение первого операнда второму.

### Логические операции:

1. |: Логическое ИЛИ. Возвращает true, если хотя бы один операнд равен true.
2. &: Логическое И. Возвращает true, если оба операнда равны true.
3. ||: Логическое ИЛИ. Работает как |, но второй операнд вычисляется только, если первый равен false.
4. &&: Логическое И. Работает как &, но второй операнд вычисляется только, если первый равен true.
5. !: Логическое отрицание. Инвертирует значение операнда.
6. ^: Исключающее ИЛИ. Возвращает true, если один из операндов равен true, но не оба.

### Разница между |/|| и &/&&:

* | и & всегда вычисляют оба операнда.
* || и && используют "короткое замыкание": второй операнд вычисляется только если это необходимо для определения результата.

Таким образом, операции || и && более эффективны с точки зрения производительности, так как позволяют избежать излишних вычислений.

## **Конструкция if..else и тернарная операция**

### Условные конструкции в C#:

1. if: Проверяет условие и выполняет блок кода, если условие истинно (true).

if (условие) { выполняемый код }

1. else: Выполняется, если условие в if ложно (false).

if (условие) { код }

else { другой код }

1. else if: Используется для проверки дополнительных условий, если первое условие ложно.

if (условие1) { код1 }

else if (условие2) { код2 }

else { код по умолчанию }

1. **Тернарная операция**: Позволяет выполнить проверку условия в одной строке.

условие ? выражение1 : выражение2

Пример:

int result = x < y ? (x + y) : (x - y);

1. **Сокращение кода**: Если блок if или else содержит одну инструкцию, можно опустить фигурные скобки:

if (условие) инструкция;

else инструкция;

### Пример:

int num1 = 8, num2 = 6;

if (num1 > num2)

Console.WriteLine($"Число {num1} больше числа {num2}");

else if (num1 < num2)

Console.WriteLine($"Число {num1} меньше числа {num2}");

else

Console.WriteLine("Число num1 равно числу num2");

### **Циклы в C#**

1. **Цикл** for:
   * Состоит из трех частей: инициализация переменной, условие выполнения цикла, действия после каждой итерации.

for (инициализация; условие; действие) {

// код

}

Пример:

for (int i = 1; i < 4; i++) {

Console.WriteLine(i);

}

1. **Цикл** while:
   * Проверяет условие перед выполнением кода.

while (условие) {

// код

}

Пример:

int i = 6;

while (i > 0) {

Console.WriteLine(i);

i--;

}

1. **Цикл** do...while:
   * Выполняет код хотя бы один раз, затем проверяет условие.

do {

// код

} while (условие);

Пример:

int i = -1;

do {

Console.WriteLine(i);

i--;

} while (i > 0);

1. **Цикл** foreach:
   * Перебирает элементы коллекции.

foreach (тип переменной in коллекция) {

// код

}

Пример:

foreach (char c in "Tom") {

Console.WriteLine(c);

}

1. **Операторы** break **и** continue:
   * break: Завершает цикл.
   * continue: Пропускает текущую итерацию и переходит к следующей. Пример break:

for (int i = 0; i < 9; i++) {

if (i == 5) break;

Console.WriteLine(i);

}

Пример continue:

for (int i = 0; i < 9; i++) {

if (i == 5) continue;

Console.WriteLine(i);

}

1. **Вложенные циклы**:
   * Циклы могут быть вложены друг в друга. Пример:

for (int i = 1; i < 10; i++) {

for (int j = 1; j < 10; j++) {

Console.Write($"{i \* j} \t");

}

Console.WriteLine();

}

**Массивы**

### 1. **Массивы в C#**

Массив — это коллекция однотипных данных. Для объявления массива в C# используется синтаксис:

тип\_переменной[] название\_массива;

### 2. **Объявление и инициализация массива**

Пример объявления массива целых чисел:

int[] nums = new int[4]; // Массив из 4 элементов

Также можно сразу указать значения:

int[] nums = {1, 2, 3, 5};

### 3. **Доступ к элементам массива**

Каждый элемент массива можно получить с помощью индекса (индексация начинается с 0):

Console.WriteLine(nums[3]); // Получим 5

nums[1] = 505; // Изменим второй элемент массива

Console.WriteLine(nums[1]); // 505

### 4. **Свойство Length**

Массивы имеют свойство Length, которое показывает количество элементов:

Console.WriteLine(nums.Length); // 4

### 5. **Получение элементов с конца**

С C# 8 можно использовать оператор ^ для получения элементов с конца:

Console.WriteLine(nums[^1]); // 5 — последний элемент

### 6. **Перебор элементов массива**

Для перебора массива можно использовать циклы foreach или for:

foreach (int i in nums)

{

Console.WriteLine(i);

}

Или с помощью цикла for:

for (int i = 0; i < nums.Length; i++)

{

Console.WriteLine(nums[i]);

}

### 7. **Многомерные массивы**

Массивы могут быть многомерными. Например, двухмерный массив:

int[,] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };

### 8. **Перебор многомерных массивов**

Для перебора многомерных массивов можно использовать вложенные циклы:

int[,] numbers = { {1, 2, 3}, {4, 5, 6} };

for (int i = 0; i < numbers.GetLength(0); i++) // Проходим по строкам

{

for (int j = 0; j < numbers.GetLength(1); j++) // Проходим по столбцам

{

Console.Write(numbers[i, j] + "\t");

}

Console.WriteLine();

}

### 9. **Массив массивов (Зубчатые массивы)**

Массив массивов — это массив, элементы которого являются другими массивами, например:

int[][] nums = new int[3][];

nums[0] = new int[] {1, 2};

nums[1] = new int[] {1, 2, 3};

nums[2] = new int[] {1, 2, 3, 4, 5};

Перебор зубчатого массива:

foreach (int[] row in nums)

{

foreach (int number in row)

{

Console.Write($"{number} ");

}

Console.WriteLine();

}

### Основные понятия:

* **Ранг (rank)**: количество измерений массива (например, для двухмерного массива ранг = 2).
* **Длина измерения (dimension length)**: количество элементов в одном измерении.
* **Длина массива (array length)**: общее количество элементов во всем массиве.

Пример для двухмерного массива:

int[,] numbers = new int[3, 4]; // Ранг = 2, длина первого измерения = 3, второго = 4

Console.WriteLine(numbers.Length); // 12 — общее количество элементов

## **Методы**

Методы в C# — это именованные блоки кода, которые выполняют определенные действия. Метод может быть определен с различными модификаторами, типом возвращаемого значения и параметрами. Простой пример метода:

void SayHello() {

Console.WriteLine("Hello");

}

**Вызов метода:** Чтобы вызвать метод, нужно использовать его имя с параметрами (если они есть):

SayHello();

**Пример с параметрами:**

void Greet(string name) {

Console.WriteLine("Hello, " + name);

}

Greet("John");

**Сокращенная запись:** Если метод состоит из одной строки, его можно сократить с помощью =>:

void SayHello() => Console.WriteLine("Hello");

**Пример с несколькими методами:**

void SayHelloRu() { Console.WriteLine("Привет"); }

void SayHelloEn() { Console.WriteLine("Hello"); }

string language = "en";

switch (language) {

case "en": SayHelloEn(); break;

case "ru": SayHelloRu(); break;

}

Методы можно вызывать многократно в разных частях программы, что повышает повторное использование кода.

**Параметры методов**

**Параметры методов в C# позволяют передавать данные в метод и использовать их для выполнения действий.**

**Определение метода с параметрами:**

void PrintMessage(string message) {

Console.WriteLine(message);

}

**Передача аргументов в параметры:** Параметры передаются в метод при его вызове:

PrintMessage("Hello world"); // Hello world

**Пример с несколькими параметрами:**

void Sum(int x, int y) {

Console.WriteLine($"{x} + {y} = {x + y}");

}

Sum(10, 15); // 10 + 15 = 25

**Сокращенная запись метода с параметрами:**

void Sum(int x, int y) => Console.WriteLine($"{x} + {y} = {x + y}");

**Соответствие типов параметров и аргументов:** Типы параметров и передаваемых значений должны совпадать. Однако, C# поддерживает автоматические преобразования типов:

PrintPerson("Tom", 37); // Name: Tom Age: 37

**Необязательные параметры:** Можно задавать значения по умолчанию для параметров:

void PrintPerson(string name, int age = 1, string company = "Undefined") {

Console.WriteLine($"Name: {name} Age: {age} Company: {company}");

}

**Пример вызова метода с необязательными параметрами:**

PrintPerson("Tom", 37); // Name: Tom Age: 37 Company: Undefined

PrintPerson("Tom"); // Name: Tom Age: 1 Company: Undefined

**Именованные параметры:** Можно указать параметры по имени, нарушив порядок их определения:

PrintPerson(name: "Tom", company: "Microsoft", age: 37);

**Консольный вывод:**

Name: Tom Age: 37 Company: Microsoft

Name: Tom Age: 37 Company: Undefined

Name: Tom Age: 1 Company: Undefined

## **Возвращение значения и оператор return**

**Возвращение значения и оператор return в C#:**

1. **Методы с возвращаемым значением**: Методы могут возвращать значения, для этого используется оператор return.

Пример:

string GetMessage() {

return "Hello";

}

1. **Правила использования** return:
   * Метод с типом, отличным от void, должен использовать return для возврата значения.
   * Тип возвращаемого значения должен совпадать с типом, указанным в определении метода.

Пример ошибки:

string GetMessage() {

return 3; // Ошибка! Метод должен возвращать строку

}

1. **Использование результатов методов**: Результаты методов можно присваивать переменным или передавать в другие методы.

Пример:

string message = GetMessage(); // Присваивание результата переменной

PrintMessage(GetMessage()); // Использование результата в другом методе

1. **Сложные выражения с** return: Можно возвращать сложные выражения или результаты вызова других методов.

Пример:

int Sum(int x, int y) {

return x + y;

}

Console.WriteLine(Sum(10, 15)); // 25

1. **Сокращенные методы с результатом**: Методы с возвращаемым значением могут быть сокращены с помощью оператора =>.

Пример:

string GetMessage() => "Hello";

int Sum(int x, int y) => x + y;

1. **Выход из метода с** return: Оператор return не только возвращает значение, но и завершает выполнение метода.

Пример:

string GetHello() {

return "Hello";

Console.WriteLine("After return"); // Не выполнится

}

1. **Использование** return **в методах с типом** void: В методах с типом void return используется для выхода из метода без возвращения значения.

Пример:

void PrintPerson(string name, int age) {

if(age > 120 || age < 1) {

Console.WriteLine("Недопустимый возраст");

return;

}

Console.WriteLine($"Имя: {name} Возраст: {age}");

}

**Консольный вывод**:

Имя: Tom Возраст: 37

Недопустимый возраст

## **Передача параметров по ссылке и значению. Выходные параметры**

1. **Передача по значению**:
   * Аргумент передается копией, изменения внутри метода не влияют на оригинальную переменную.
   * Пример:

void Increment(int n) {

n++;

}

* + Консольный вывод:

Число до метода Increment: 5

Число в методе Increment: 6

Число после метода Increment: 5

1. **Передача по ссылке (ref)**:
   * Используется модификатор ref, параметр передается по ссылке, изменения в методе влияют на оригинальную переменную.
   * Пример:

void Increment(ref int n) {

n++;

}

* + Консольный вывод:

Число до метода Increment: 5

Число в методе Increment: 6

Число после метода Increment: 6

1. **Выходные параметры (out)**:
   * Модификатор out используется для передачи параметров, которые метод должен инициализировать.
   * Пример:

void Sum(int x, int y, out int result) {

result = x + y;

}

* + Консольный вывод:

25

1. **Использование нескольких выходных параметров**:
   * Можно вернуть несколько значений через out параметры.
   * Пример:

void GetRectangleData(int width, int height, out int area, out int perimeter) {

area = width \* height;

perimeter = 2 \* (width + height);

}

* + Консольный вывод:

Площадь прямоугольника: 200

Периметр прямоугольника: 60

1. **Входные параметры (in)**:
   * Модификатор in используется для параметров, которые передаются по ссылке, но не могут быть изменены внутри метода.
   * Пример:

void GetRectangleData(in int width, in int height, out int area, out int perimeter) {

area = width \* height;

perimeter = 2 \* (width + height);

}

1. **Readonly** ref **параметры**:
   * В C# 12 можно использовать параметры ref readonly, которые доступны только для чтения.
   * Пример:

void Increment(ref readonly int n) {

// n++; // Ошибка

}

**Основное различие между** ref**,** out**, и** in**:**

* ref: передача по ссылке с возможностью изменения параметра.
* out: передача по ссылке, метод обязан инициализировать параметр.
* in: передача по ссылке, но параметр доступен только для чтения.

**Массив параметров и ключевое слово params**

**Массив параметров и ключевое слово params в C#:**

1. **Использование** params:
   * Ключевое слово params позволяет передавать неопределенное количество параметров в метод.
   * Пример:

void Sum(params int[] numbers) {

int result = 0;

foreach (var n in numbers) {

result += n;

}

Console.WriteLine(result);

}

* + Вызовы метода:

Sum(1, 2, 3, 4); // 10

Sum(); // 0

* + Можно передавать как массив, так и отдельные значения, или вообще не передавать параметры.

1. **Параметры с** params **и обычные параметры**:
   * Параметры с params должны идти последними в методе, а другие обязательные параметры — до них.
   * Пример:

void Sum(int initialValue, params int[] numbers) {

int result = initialValue;

foreach (var n in numbers) {

result += n;

}

Console.WriteLine(result);

}

* + Вызовы метода:

Sum(10, 1, 2, 3); // 16

Sum(5); // 5

1. **Невозможность указания других параметров после** params:
   * После параметра с params нельзя указывать другие параметры.
   * Пример с ошибкой:

void Sum(params int[] numbers, int initialValue) {} // Неправильно

1. **Массив как параметр без** params:
   * В отличие от params, если метод принимает массив как параметр, нужно передавать именно массив.
   * Пример:

void Sum(int[] numbers, int initialValue) {

int result = initialValue;

foreach (var n in numbers) {

result += n;

}

Console.WriteLine(result);

}

* + Вызов:

int[] nums = { 1, 2, 3, 4 };

Sum(nums, 10); // 20

* + Невозможно вызвать метод так:

Sum(1, 2, 3, 4); // Ошибка

**Основное различие**:

* params: позволяет передавать неопределенное количество значений.
* Обычный массив: требует явной передачи массива.

## **Рекурсивные функции**

1. **Рекурсивная функция факториала**:
   * Функция вызывает саму себя до достижения базового случая.
   * Пример:

int Factorial(int n)

{

if (n == 1) return 1;

return n \* Factorial(n - 1);

}

* + Процесс:
    - Функция вызывает себя с уменьшением значения на 1, пока не достигнет базового случая (n == 1).
    - Пример с Factorial(4):
      * Factorial(4) → 4 \* Factorial(3)
      * Factorial(3) → 3 \* Factorial(2)
      * Factorial(2) → 2 \* Factorial(1)
      * Factorial(1) возвращает 1, и вычисление продолжает раскручиваться обратно.

1. **Рекурсивная функция Фибоначчи**:
   * Формула: f(n) = f(n-1) + f(n-2), с базовыми случаями f(0) = 0 и f(1) = 1.
   * Пример:

int Fibonachi(int n)

{

if (n == 0 || n == 1) return n;

return Fibonachi(n - 1) + Fibonachi(n - 2);

}

* + Процесс:
    - Функция вызывает себя дважды, уменьшая n на 1 и 2, до тех пор, пока не достигнет 0 или 1.

1. **Рекурсия vs Циклы**:
   * Рекурсивные функции, хотя и элегантные, могут быть менее эффективными по сравнению с циклами.
   * Пример вычисления чисел Фибоначчи с использованием цикла:

static int Fibonachi2(int n)

{

int result = 0, b = 1, tmp;

for (int i = 0; i < n; i++)

{

tmp = result;

result = b;

b += tmp;

}

return result;

}

* + Циклические решения обычно более эффективны, чем рекурсия, особенно для чисел Фибоначчи.

1. **Когда использовать рекурсию**:
   * Рекурсия может быть полезной для задач, связанных с древовидными структурами, например, при обходе файловых систем или каталогов, где рекурсивные вызовы обеспечивают более элегантное решение.

**Основные моменты**:

* Рекурсивные функции полезны для задач с повторяющимися подзадачами, но могут быть менее эффективны для простых задач (например, вычисление чисел Фибоначчи).
* Важно наличие базового случая, чтобы предотвратить бесконечную рекурсию.

## **Локальные функции**

1. **Определение локальной функции**:
   * Локальная функция — это функция, определенная внутри другого метода.
   * Она обычно выполняет действия, которые применяются только в рамках этого метода.
   * Пример:

void Compare(int[] numbers1, int[] numbers2)

{

int numbers1Sum = Sum(numbers1);

int numbers2Sum = Sum(numbers2);

// Дополнительный код

}

int Sum(int[] numbers) // Локальная функция

{

int result = 0;

foreach (int number in numbers)

result += number;

return result;

}

1. **Преимущества локальных функций**:
   * Локальные функции удобны для задач, которые не требуют их повторного использования за пределами метода.
   * Они помогают избежать избыточного кода, например, при повторяющихся вычислениях.
2. **Статические локальные функции**:
   * Локальные функции могут быть **статическими** с помощью модификатора static.
   * Статическая функция не может обращаться к переменным окружающего метода.
   * Пример с ошибкой при попытке обратиться к переменной:

static bool IsPassed(int number, int lim)

{

return number > lim; // Правильно, переменная limit передается как параметр

}

**Основные моменты**:

* Локальные функции помогают инкапсулировать логику внутри метода.
* Статические локальные функции не могут обращаться к переменным внешнего метода, только через параметры.

## **Конструкция switch**

1. **Общее описание**:
   * switch оценивает выражение и сравнивает его с набором значений в case.
   * Каждый case соответствует значению, и если совпадение найдено, выполняется соответствующий блок кода.
2. **Основная структура**:

switch (выражение)

{

case значение1:

// код

break;

case значение2:

// код

break;

default:

// код, если нет совпадений

break;

}

1. **Пример использования**:

string name = "Tom";

switch (name)

{

case "Bob":

Console.WriteLine("Ваше имя - Bob");

break;

case "Tom":

Console.WriteLine("Ваше имя - Tom");

break;

default:

Console.WriteLine("Неизвестное имя");

break;

}

* + Если значение переменной name не совпадает с ни одним из case, выполняется блок default.

1. **Переход между** case:
   * Можно использовать goto case для перехода к другому блоку case.

switch (number)

{

case 1:

Console.WriteLine("case 1");

goto case 5; // переход к case 5

case 5:

Console.WriteLine("case 5");

break;

}

1. **Возврат значения из** switch:
   * В switch можно использовать return для возврата значения из метода.

int DoOperation(int op, int a, int b)

{

switch (op)

{

case 1: return a + b;

case 2: return a - b;

default: return 0;

}

}

1. **Упрощенная запись через выражение** switch:
   * В C# 8.0 и выше можно использовать выражение switch для получения результата:

int result = op switch

{

1 => a + b,

2 => a - b,

\_ => 0

};

1. **Важные моменты**:
   * Если метод использует switch для возврата значения, необходимо учесть все возможные варианты, включая блок \_ (default).
   * Без блока default, метод может не скомпилироваться, если входное значение не соответствует ни одному из case.
2. **Краткая форма метода**:
   * Метод можно записать в виде однострочника:

int DoOperation(int op, int a, int b) => op switch

{

1 => a + b,

2 => a - b,

3 => a \* b,

\_ => 0

};

## **Перечисления enum**

1. **Общее описание**:
   * Перечисление (enum) — это тип данных, представляющий набор логически связанных констант.
2. **Объявление перечисления**:

enum НазваниеПеречисления

{

значение1,

значение2,

// ...

}

1. **Пример простого перечисления**:

enum DayTime

{

Morning,

Afternoon,

Evening,

Night

}

1. **Использование перечислений**:
   * Переменные типа перечисления могут быть использованы как обычные переменные:

DayTime dayTime = DayTime.Morning;

1. **Пример использования в условном операторе**:

if (dayTime == DayTime.Morning)

Console.WriteLine("Доброе утро");

1. **Использование перечислений в методах**:

void PrintMessage(DayTime dayTime)

{

switch (dayTime)

{

case DayTime.Morning: Console.WriteLine("Доброе утро"); break;

case DayTime.Afternoon: Console.WriteLine("Добрый день"); break;

case DayTime.Evening: Console.WriteLine("Добрый вечер"); break;

case DayTime.Night: Console.WriteLine("Доброй ночи"); break;

}

}

1. **Перечисления в качестве параметров**:
   * Пример с арифметическими операциями:

enum Operation

{

Add,

Subtract,

Multiply,

Divide

}

void DoOperation(double x, double y, Operation op)

{

double result = op switch

{

Operation.Add => x + y,

Operation.Subtract => x - y,

Operation.Multiply => x \* y,

Operation.Divide => x / y

};

Console.WriteLine(result);

}

1. **Типы перечислений**:
   * Перечисления могут иметь явный тип (например, byte, int):

enum DayTime : byte

{

Morning,

Afternoon,

Evening,

Night

}

1. **Значения перечислений**:
   * По умолчанию, каждому элементу перечисления присваивается целочисленное значение, начиная с 0:

Console.WriteLine((int) DayTime.Morning); // 0

1. **Явное указание значений**:
   * Можно указать значения элементов перечисления:

enum DayTime

{

Morning = 3,

Afternoon, // 4

Evening, // 5

Night // 6

}

1. **Перечисления с одинаковыми значениями**:
   * Константы перечисления могут иметь одинаковые значения:

enum DayTime

{

Morning = 1,

Afternoon = Morning,

Evening = 2,

Night = 2

}

1. **Ограничения**:
   * Нельзя присвоить числовое значение переменной перечисления напрямую:

DayTime now = 2; // Ошибка

Перечисления полезны для создания набора логически связанных значений и могут использоваться для более читаемого и управляемого кода.

# **Классы, структуры и пространства имен**

## **Классы и объекты**

**Общее описание классов:**

* C# — объектно-ориентированный язык, и программы можно представлять как взаимосвязанные объекты.
* Класс — это описание объекта, а объект — экземпляр этого класса.
* **Объявление класса**:
* Классы определяются с помощью ключевого слова class:

## **Класс Program и метод Main. Программы верхнего уровня**

**Точка входа и программы верхнего уровня в C#**

1. **Метод Main**:
   * Точка входа в программу на C# — это метод Main.
   * Он обязательно присутствует в консольных приложениях, но с версии C# 9.0 (и в Visual Studio 2022) появилась возможность использовать **программы верхнего уровня** (top-level programs), где метод Main генерируется автоматически компилятором.
2. **Программа верхнего уровня**:
   * В Visual Studio 2022 можно писать код без явного определения класса Program и метода Main. Компилятор сам создаст метод Main за кулисами.
   * Пример программы верхнего уровня:

Console.WriteLine("Hello, World!");

Этот код эквивалентен следующему:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Hello, World!");

}

}

1. **Структура программы**:
   * Для программы верхнего уровня можно определять переменные, константы и методы без необходимости использовать класс Program и метод Main.
   * Пример с функцией:

string hello = "Hello METANIT.COM";

Print(hello);

void Print(string message)

{

Console.WriteLine(message);

}

Этот код эквивалентен следующему с явным определением метода Main:

class Program

{

static void Main(string[] args)

{

string hello = "Hello METANIT.COM";

Print(hello);

void Print(string message)

{

Console.WriteLine(message);

}

}

}

1. **Определение классов в программах верхнего уровня**:
   * Классы и другие типы могут быть определены после инструкций верхнего уровня, например:

Person tom = new();

tom.SayHello();

class Person

{

public void SayHello() => Console.WriteLine("Hello");

}

Эквивалент с явным методом Main:

class Program

{

static void Main(string[] args)

{

Person tom = new();

tom.SayHello();

}

}

class Person

{

public void SayHello() => Console.WriteLine("Hello");

}

Таким образом, с C# 9.0 и Visual Studio 2022 появилась возможность писать программы без явного метода Main, что упрощает код и делает его более удобным для быстрого прототипирования.

## **Структуры**

1. **Определение структуры**:
   * Для создания структуры используется ключевое слово struct:

struct Person

{

public string name;

public int age;

}

* + В C# 12 можно опустить фигурные скобки, если структура пуста:

struct Person;

1. **Элементы структуры**:
   * Структуры могут содержать поля, методы и другие члены. Пример:

struct Person

{

public string name;

public int age;

public void Print()

{

Console.WriteLine($"Имя: {name} Возраст: {age}");

}

}

1. **Создание и инициализация объектов структуры**:
   * Инициализация структуры происходит через конструктор, который генерируется автоматически, если не определен явный.
   * Пример инициализации:

Person tom = new Person();

tom.name = "Tom";

tom.Print(); // Имя: Tom Возраст: 0

1. **Непосредственная инициализация полей**:
   * Структуру можно инициализировать вручную, если все поля публичные:

Person tom;

tom.name = "Sam";

tom.age = 37;

tom.Print(); // Имя: Sam Возраст: 37

1. **Инициализация полей по умолчанию** (C# 10 и выше):
   * Можно задать значения по умолчанию при определении полей:

struct Person

{

public string name = "Tom";

public int age = 1;

}

1. **Конструкторы структуры**:
   * Структуры могут иметь явные конструктора:

struct Person

{

public string name;

public int age;

public Person(string name, int age)

{

this.name = name;

this.age = age;

}

}

* + Также можно использовать цепочку конструкторов:

public Person() : this("Tom") { }

public Person(string name) : this(name, 1) { }

1. **Первичные конструкторы** (C# 12):
   * Структуры могут иметь первичный конструктор с параметрами:

public struct Person(string name, int age)

{

public void Print() => Console.WriteLine($"name: {name}, age: {age}");

}

1. **Инициализация с помощью инициализатора**:
   * Можно использовать инициализатор для создания объекта:

Person tom = new Person { name = "Tom", age = 22 };

1. **Копирование структуры с изменениями** (C# 9 и выше):
   * Для копирования объекта с изменениями используется оператор with:

Person tom = new Person { name = "Tom", age = 22 };

Person bob = tom with { name = "Bob" };

bob.Print(); // Имя: Bob Возраст: 22

Таким образом, структуры в C# предоставляют способ создания типов данных с фиксированным состоянием и поведением, используя простоту и низкую нагрузку на память.

## **Типы значений и ссылочные типы**

1. **Типы значений**:
   * Примеры: int, byte, float, bool, char, enum, struct.
   * Хранятся в **стеке**.
   * При передаче копируются: каждая переменная имеет собственное значение.
2. **Ссылочные типы**:
   * Примеры: object, string, class, interface, delegate.
   * Хранятся в **куче** (heap), а ссылки на них — в стеке.
   * При передаче копируется **ссылка**, но обе переменные указывают на один объект в куче.

**Память**:

* **Стек (stack)**: хранит значения типов значений, параметры и локальные переменные методов. Быстрая и ограниченная память.
* **Куча (heap)**: хранит объекты ссылочных типов. Более медленная, управляется сборщиком мусора.

**Копирование значений**:

* Типы значений копируются полностью.
* Ссылочные типы копируют только ссылки, изменяя объект, вы меняете все ссылки на него.

**Составные типы**

* **Структуры**: хранятся в стеке, но если содержат поля ссылочного типа, те размещаются в куче.
* **Классы**: полностью размещаются в куче, а ссылки на них — в стеке.

**Передача параметров**:

* **По значению (по умолчанию)**:
  + Копируется ссылка или значение. Поля можно изменить, но нельзя заменить объект.
* **По ссылке (ref)**:
  + Передается ссылка, можно изменять объект или переназначать его.

Пример:

void ChangePerson(Person person) { person.name = "Alice"; } // Изменит поле

void ChangePerson(ref Person person) { person = new Person(); } // Заменит объект

## **Область видимости (контекст) переменных и констант**

1. **Контексты переменных**:
   * **Контекст класса**: переменные (поля) доступны в любом методе класса. Называются **глобальными переменными**.
   * **Контекст метода**: переменные, определенные внутри метода, называются **локальными** и доступны только в этом методе.
   * **Контекст блока кода**: переменные доступны только в пределах блока кода {}.
2. **Примеры и области действия**:

class Person // Контекст класса

{

string type = "Person"; // Глобальная переменная класса

public void PrintName() // Контекст метода

{

string name = "Tom"; // Локальная переменная метода

{ // Контекст блока кода

string shortName = "Tomas"; // Локальная переменная блока

Console.WriteLine(type); // Доступна переменная класса

Console.WriteLine(name); // Доступна переменная метода

Console.WriteLine(shortName); // Доступна переменная блока

}

// shortName недоступна за пределами блока кода

}

}

1. **Скрытие переменных**:
   * Локальная переменная метода или блока кода **скрывает** переменную класса с тем же именем:

class Person

{

string name = "Tom"; // Переменная класса

public void PrintName()

{

string name = "Tomas"; // Локальная переменная метода скрывает переменную класса

Console.WriteLine(name); // Вывод: Tomas

}

}

1. **Ограничения**:
   * В одном контексте нельзя объявить несколько переменных с одинаковыми именами:

void Example()

{

int x = 10;

// int x = 20; // Ошибка: переменная с таким именем уже существует в этом контексте

}

1. **Жизненный цикл переменных**:
   * Переменные уничтожаются, когда выполнение выходит за пределы их контекста:
     + Локальные переменные метода уничтожаются после завершения метода.
     + Переменные блока уничтожаются после выхода из блока.

## **Пространства имен**

1. **Определение и назначение**:
   * Пространства имен позволяют организовать код в логические блоки, отделяя функциональность для удобства и предотвращения конфликтов имен.
   * Для определения используется ключевое слово namespace, за которым следует имя пространства:

namespace MyNamespace

{

// содержимое пространства

}

1. **Пример простого пространства имен**:

namespace Base

{

class Person

{

string name;

public Person(string name) => this.name = name;

public void Print() => Console.WriteLine($"Имя: {name}");

}

}

1. **Использование классов из пространства имен**:
   * Обращение через полное имя:

Base.Person tom = new("Tom");

tom.Print();

* + Подключение пространства с помощью using:

using Base;

Person tom = new("Tom");

tom.Print();

1. **Вложенные пространства имен**:
   * Пространства имен могут быть вложенными:

namespace Base

{

namespace PersonTypes

{

class Person { /\* ... \*/ }

}

namespace OrganisationTypes

{

class Company { /\* ... \*/ }

}

}

* + Обращение к классам:

using Base.PersonTypes;

Base.OrganisationTypes.Company company = new("Microsoft");

Person person = new("Tom", company);

1. **Пространства имен уровня файла (C# 10 и .NET 6)**:
   * Пространства имен можно объявлять на уровне файла:

namespace Base;

class Person

{

string name;

public Person(string name) => this.name = name;

public void Print() => Console.WriteLine($"Имя: {name}");

}

* + Подключение аналогично:

using Base;

Person tom = new("Tom");

tom.Print();

1. **Ключевые моменты**:
   * Пространства имен уменьшают вероятность конфликтов имен в больших проектах.
   * Подключение через using упрощает работу с множеством классов.
   * Вложенные пространства имен помогают структурировать сложные проекты.

## **Глобальные пространства имен**

1. **Что такое глобальные пространства имен?**
   * Глобальные пространства имен позволяют подключать пространство имен один раз для всего проекта, что избавляет от необходимости прописывать using в каждом файле.
   * Вводятся с помощью директивы global using.
2. **Пример использования глобальных пространств имен**:
   * В обычном подходе, чтобы использовать пространство имен Base, необходимо подключать его в каждом файле:

// Пример в файле Program.cs

using Base;

Company microsoft = new("Microsoft");

Person tom = new("Tom", microsoft);

tom.Print();

* + С использованием глобальных пространств имен:

global using Base;

Company microsoft = new("Microsoft");

Person tom = new("Tom", microsoft);

tom.Print();

1. **Определение глобальных пространств имен**:
   * Глобальные пространства имен можно объявить в любом файле проекта. Например:

global using Base;

* + После этого пространство Base будет доступно во всех файлах проекта без дополнительного using.

1. **Определение глобальных пространств имен в отдельном файле**:
   * Для удобства глобальные using можно вынести в отдельный файл, например, GlobalUsings.cs:

global using System.Text;

global using System.Reflection;

global using Base;

* + Этот файл служит центральным местом для управления глобальными подключениями.

1. **Преимущества**:
   * Упрощение кода за счет уменьшения дублирования.
   * Легкость управления подключениями для всего проекта через один файл (GlobalUsings.cs).
2. **Поддержка**:
   * Глобальные пространства имен поддерживаются начиная с .NET 6 и C# 10.

## **Свойства**

### Свойства в C#

Свойства — это специальные методы доступа, которые обеспечивают удобный и контролируемый доступ к полям классов и структур. Они могут включать дополнительную логику, например, проверку значений при установке.

### **Стандартное определение свойства**

[модификаторы] тип\_свойства название\_свойства

{

get { действия при получении значения свойства }

set { действия при установке значения свойства }

}

* get: возвращает значение свойства.
* set: устанавливает новое значение через параметр value.

### **Пример**

class Person

{

private string name = "Undefined";

public string Name

{

get { return name; }

set { name = value; }

}

}

### **Типы свойств**

1. **Свойства только для чтения:**

public string Name

{

get { return name; }

}

* + Можно только получить значение.

1. **Свойства только для записи:**

public int Age

{

set { age = value; }

}

* + Можно только установить значение.

1. **Вычисляемые свойства:**

public string FullName

{

get { return $"{firstName} {lastName}"; }

}

* + Возвращают вычисленное значение, не связанное напрямую с полем.

### **Модификаторы доступа**

* Могут применяться к get и set отдельно:

public string Name

{

get { return name; }

private set { name = value; }

}

### **Автоматические свойства**

Упрощают объявление:

public string Name { get; set; }

public int Age { get; set; } = 30; // Значение по умолчанию

### init **свойства (C# 9.0)**

* Можно установить только при инициализации:

public string Name { get; init; }

Пример использования:

var person = new Person { Name = "Tom" };

### required **свойства (C# 11)**

* Должны быть инициализированы:

public required string Name { get; set; }

Пример:

var person = new Person { Name = "Tom", Age = 30 };

### **Сокращенная запись**

1. Сокращение get/set:

public string Name

{

get => name;

set => name = value;

}

1. Полное сокращение:

public string Name => name;

Свойства позволяют управлять доступом к данным и включать дополнительную логику, оставаясь удобным инструментом для работы с полями классов и структур.

## **Перегрузка методов**

**Перегрузка методов в C# позволяет создавать методы с одинаковым именем, но разными сигнатурами. Сигнатура метода включает:**

1. **Имя метода.**
2. **Количество параметров.**
3. **Типы параметров.**
4. **Порядок параметров.**
5. **Модификаторы параметров (например,** ref**,** out**).**

**Не входят в сигнатуру:**

* Названия параметров.
* Тип возвращаемого значения.

### Примеры перегрузки

1. **Отличие по количеству параметров:**

void Add(int a, int b);

void Add(int a, int b, int c);

1. **Отличие по типу параметров:**

void Add(int a, int b);

void Add(double a, double b);

1. **Отличие по модификаторам параметров:**

void Increment(ref int val);

void Increment(int val);

### Некорректная перегрузка

* Методы с одинаковой сигнатурой, но разными названиями параметров или возвращаемыми типами, не считаются перегруженными:

int Sum(int x, int y);

void Sum(int x, int y); // Ошибка: сигнатуры совпадают.

### Использование

Пример вызова перегруженных методов:

Calculator calc = new Calculator();

calc.Add(1, 2); // Вывод: 3

calc.Add(1, 2, 3); // Вывод: 6

calc.Add(1.4, 2.5); // Вывод: 3.9

**Вывод:** Перегрузка позволяет использовать методы с одним именем, повышая читаемость и гибкость кода.

1. **Имя метода.**
2. **Количество параметров.**
3. **Типы параметров.**
4. **Порядок параметров.**
5. **Модификаторы параметров (например,** ref**,** out**).**

**Не входят в сигнатуру:**

* Названия параметров.
* Тип возвращаемого значения.

### Примеры перегрузки

1. **Отличие по количеству параметров:**

void Add(int a, int b);

void Add(int a, int b, int c);

1. **Отличие по типу параметров:**

void Add(int a, int b);

void Add(double a, double b);

1. **Отличие по модификаторам параметров:**

void Increment(ref int val);

void Increment(int val);

### Некорректная перегрузка

* Методы с одинаковой сигнатурой, но разными названиями параметров или возвращаемыми типами, не считаются перегруженными:

int Sum(int x, int y);

void Sum(int x, int y); // Ошибка: сигнатуры совпадают.

### Использование

Пример вызова перегруженных методов:

Calculator calc = new Calculator();

calc.Add(1, 2); // Вывод: 3

calc.Add(1, 2, 3); // Вывод: 6

calc.Add(1.4, 2.5); // Вывод: 3.9

**Вывод:** Перегрузка позволяет использовать методы с одним именем, повышая читаемость и гибкость кода.

## **Статические члены и модификатор static**

**Статические члены в C#**

Статические поля, методы, свойства и конструкторы относятся ко всему классу/структуре, а не к отдельным объектам. Для обращения к ним не требуется создавать экземпляр класса.

### **Статические поля**

* Хранят состояние, общее для всех объектов класса.
* Определяются с ключевым словом static.
* Доступ к ним осуществляется через имя класса:

Console.WriteLine(Person.retirementAge);

Person.retirementAge = 67;

* Память для статических полей выделяется даже без создания объектов класса.

### **Статические свойства**

* Обеспечивают доступ к статическим данным с помощью геттеров и сеттеров:

public static int RetirementAge

{

get { return retirementAge; }

set { if (value > 1 && value < 100) retirementAge = value; }

}

### **Счетчики с использованием статических полей**

* Для отслеживания общего количества созданных объектов:

static int counter = 0;

public static int Counter => counter;

public Person() => counter++;

### **Статические методы**

* Определяют поведение, общее для всех объектов.
* Обращение осуществляется через имя класса:

Person.CheckRetirementStatus(bob);

* Могут работать только со статическими членами класса.

### **Статический конструктор**

* Используется для инициализации статических данных.
* Особенности:
  + Не имеет параметров.
  + Выполняется автоматически при первом обращении к классу.
  + Пример:

static Person()

{

retirementAge = (DateTime.Now.Year == 2022) ? 65 : 67;

}

### **Статические классы**

* Содержат только статические члены.
* Объявляются с модификатором static.
* Используются для функций, не привязанных к состоянию объекта:

static class Operations

{

public static int Add(int x, int y) => x + y;

public static int Subtract(int x, int y) => x - y;

}

### **Пример использования**

Console.WriteLine(Operations.Add(5, 4)); // 9

Console.WriteLine(Person.RetirementAge); // 65

Person bob = new(68);

bob.СheckAge();

## **Константы, поля и структуры для чтения**

### **Константы и поля для чтения в C#**

### **Константы (**const**)**

* Значение устанавливается при объявлении и не может быть изменено.
* Константа относится ко всему классу и доступна через имя класса:

public const string type = "Person";

Console.WriteLine(Person.type); // Person

* **Особенности:**
  + Значение известно во время компиляции.
  + Неявно являются static.
  + Используются для хранения неизменяемых данных, общих для всего класса.

### **Поля для чтения (**readonly**)**

* Значение может быть установлено:
  + При объявлении.
  + В конструкторе.
* После инициализации значение изменить нельзя.

public readonly string name = "Undefined";

public Person(string name) => this.name = name;

* **Особенности:**
  + Значение может задаваться во время выполнения.
  + Поле может быть статическим или нестатическим.

### **Сравнение** const **и** readonly

| ****Параметр**** | ****const**** | ****readonly**** |
| --- | --- | --- |
| Установка значения | Только при объявлении | При объявлении или в конструкторе |
| Время определения значения | Компиляция | Выполнение программы |
| Модификатор static | Неявно статические | Может быть статическим |
| Использование | Для неизменяемых констант | Для значений, инициализируемых в рантайме |

### **Структуры для чтения (**readonly struct**)**

* Поля и свойства таких структур должны быть только для чтения:

readonly struct Person

{

public readonly string Name { get; }

public int Age { get; }

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

* Используются для создания неизменяемых типов данных.

# **Интерфейсы**

## **Определение интерфейсов**

Интерфейс представляет ссылочный тип, который может определять некоторый функционал - набор методов и свойств без реализации. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

### **Определение интерфейса**

Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в C# начинаются с заглавной буквы **I**, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования.

Что может определять интерфейс? В целом интерфейсы могут определять следующие сущности:

* Методы
* Свойства
* Индексаторы
* События
* Статические поля и константы (начиная с версии C# 8.0)

Однако интерфейсы не могут определять нестатические переменные. Например, простейший интерфейс, который определяет все эти компоненты:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | interface IMovable  {  // константа  const int minSpeed = 0; // минимальная скорость  // статическая переменная  static int maxSpeed = 60; // максимальная скорость  // метод  void Move(); // движение  // свойство  string Name { get; set; } // название    delegate void MoveHandler(string message); // определение делегата для события  // событие  event MoveHandler MoveEvent; // событие движения  } |

В данном случае определен интерфейс IMovable, который представляет некоторый движущийся объект. Данный интерфейс содержит различные компоненты, которые описывают возможности движущегося объекта. То есть интерфейс описывает некоторый функционал, который должен быть у движущегося объекта.

Методы и свойства интерфейса могут не иметь реализации, в этом они сближаются с абстрактными методами и свойствами абстрактных классов. В данном случае интерфейс определяет метод Move, который будет представлять некоторое передвижение. Он не имеет реализации, не принимает никаких параметров и ничего не возвращает.

То же самое в данном случае касается свойства Name. На первый взгляд оно похоже на автоматическое свойство. Но в реальности это определение свойства в интерфейсе, которое не имеет реализации, а не автосвойство.

### **Модификаторы доступа**

Еще один момент в объявлении интерфейса: если его члены - методы и свойства не имеют модификаторов доступа, то фактически по умолчанию доступ **public**, так как цель интерфейса - определение функционала для реализации его классом. Это касается также и констант и статических переменных, которые в классах и структурах по умолчанию имееют модификатор private. В интерфейсах же они имеют по умолчанию модификатор public. И например, мы могли бы обратиться к константе minSpeed и переменной maxSpeed интерфейса IMovable:

|  |  |
| --- | --- |
| 1  2 | Console.WriteLine(IMovable.maxSpeed); // 60  Console.WriteLine(IMovable.minSpeed); // 0 |

Но также, начиная с версии C# 8.0, мы можем явно указывать модификаторы доступа у компонентов интерфейса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | interface IMovable  {  public const int minSpeed = 0; // минимальная скорость  private static int maxSpeed = 60; // максимальная скорость  public void Move();  protected internal string Name { get; set; } // название  public delegate void MoveHandler(string message); // определение делегата для события  public event MoveHandler MoveEvent; // событие движения  } |

Как и классы, интерфейсы по умолчанию имеют уровень доступа **internal**, то есть такой интерфейс доступен только в рамках текущего проекта. Но с помощью модификатора public мы можем сделать интерфейс общедоступным:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface IMovable  {  void Move();  } |

### **Реализация по умолчанию**

Также начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Это значит, что мы можем определить в интерфейсах полноценные методы и свойства, которые имеют реализацию как в обычных классах и структурах. Например, определим реализацию метода Move по умолчанию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | interface IMovable  {  // реализация метода по умолчанию  void Move()  {  Console.WriteLine("Walking");  }  } |

С реализацией свойств по умолчанию в интерфейсах дело обстоит несколько сложнее, поскольку мы не можем определять в интерфейсах нестатические переменные, соответственно в свойствах интерфейса мы не можем манипулировать состоянием объекта. Тем не менее реализацию по умолчанию для свойств мы тоже можем определять:

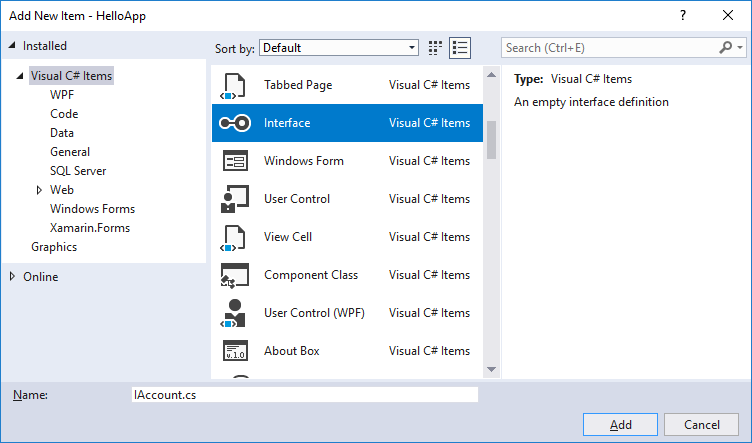
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | interface IMovable  {  // реализация метода по умолчанию  void Move() => Console.WriteLine("Walking");  // реализация свойства по умолчанию  // свойство только для чтения  int MaxSpeed { get { return 0; } }  } |

Стоит отметить, что если интерфейс имеет приватные методы и свойства (то есть с модификатором private), то они должны иметь реализацию по умолчанию. То же самое относится к статическим методам (не обязательно приватным):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | Console.WriteLine(IMovable.MaxSpeed); // 60  IMovable.MaxSpeed = 65;  Console.WriteLine(IMovable.MaxSpeed); // 65  double time = IMovable.GetTime(500, 10);  Console.WriteLine(time); // 50    interface IMovable  {  public const int minSpeed = 0; // минимальная скорость  private static int maxSpeed = 60; // максимальная скорость  // находим время, за которое надо пройти расстояние distance со скоростью speed  static double GetTime(double distance, double speed) => distance / speed;  static int MaxSpeed  {  get => maxSpeed;  set  {  if (value > 0) maxSpeed = value;  }  }  } |

### **Добавление интерфейса**

Стоит отметить, что в Visual Studio есть специальный компонент для добавления нового интерфейса в отдельном файле. Для добавления интерфейса в проект можно нажать правой кнопкой мыши на проект и в появившемся контекстном меню выбрать **Add**-> **New Item...** и в диалоговом окне добавления нового компонента выбрать пункт **Interface**:



## **Применение интерфейсов**

Интерфейс представляет некое описание типа, набор компонентов, который должен иметь тип данных. И, собственно, мы не можем создавать объекты интерфейса напрямую с помощью конструктора, как например, в классах:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | IMovable m = new IMovable(); // ! Ошибка, так сделать нельзя    interface IMovable  {  void Move();  } |

В конечном счете интерфейс предназначен для реализации в классах и структурах. Например, реализуем выше определенный интерфейс IMovable:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // применение интерфейса в классе  class Person : IMovable  {  public void Move()  {  Console.WriteLine("Человек идет");  }  }  // применение интерфейса в структуре  struct Car : IMovable  {  public void Move()  {  Console.WriteLine("Машина едет");  }  } |

При применении интерфейса, как и при наследовании после имени класса или структуры указывается двоеточие и затем идут названия применяемых интерфейсов. При этом класс должен реализовать все методы и свойства применяемых интерфейсов, если эти методы и свойства не имеют реализации по умолчанию.

Если методы и свойства интерфейса не имеют модификатора доступа, то по умолчанию они являются публичными, при реализации этих методов и свойств в классе и структуре к ним можно применять только модификатор public.

Применение интерфейса в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | Person person = new Person();  Car car = new Car();  DoAction(person);  DoAction(car);    void DoAction(IMovable movable) => movable.Move();    interface IMovable  {  void Move();  }  class Person : IMovable  {  public void Move() => Console.WriteLine("Человек идет");  }  struct Car : IMovable  {  public void Move() => Console.WriteLine("Машина едет");  } |

В данной программе определен метод DoAction(), который в качестве параметра принимает объект интерфейса IMovable. На момент написания кода мы можем не знать, что это будет за объект - какой-то класс или структура. Единственное, в чем мы можем быть уверены, что этот объект обязательно реализует метод Move и мы можем вызвать этот метод.

Иными словами, интерфейс - это контракт, что какой-то определенный тип обязательно реализует некоторый функционал.

Консольный вывод данной программы:

Человек идет

Машина едет

### **Реализация интерфейсов по умолчанию**

Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Зачем это нужно? Допустим, у нас есть куча классов, которые реализуют некоторый интерфейс. Если мы добавим в этот интерфейс новый метод, то мы будем обязаны реализовать этот метод во всех классах, применяющих данный интерфейс. Иначе подобные классы просто не будут компилироваться. Теперь вместо реализации метода во всех классах нам достаточно определить его реализацию по умолчанию в интерфейсе. Если класс не реализует метод, будет применяться реализация по умолчанию.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | IMovable tom = new Person();  Car tesla = new Car();  tom.Move(); // Walking  tesla.Move(); // Driving  interface IMovable  {  void Move() => Console.WriteLine("Walking");  }  class Person : IMovable { }  class Car : IMovable  {  public void Move() => Console.WriteLine("Driving");  } |

В данном случае интерфейс IMovable определяет реализацию по умолчанию для метода Move. Класс Person не реализует этот метод, поэтому он применяет реализацию по умолчанию в отличие от класса Car, который определяет свою реализацию для метода Move.

Стоит отметить, что хотя для объекта класса Person мы можем вызвать метод Move - ведь класс Person применяет интерфейс IMovable, тем не менее мы не можем написать так:

|  |  |
| --- | --- |
| 1  2 | Person tom = new Person();  tom.Move(); // Ошибка - метод Move не определен в классе Person |

### **Множественная реализация интерфейсов**

Интерфейсы имеют еще одну важную функцию: в C# не поддерживается множественное наследование, то есть мы можем унаследовать класс только от одного класса, в отличие, скажем, от языка С++, где множественное наследование можно использовать. Интерфейсы позволяют частично обойти это ограничение, поскольку в C# классы и структуры могут реализовать сразу несколько интерфейсов. Все реализуемые интерфейсы указываются через запятую:

|  |  |
| --- | --- |
| 1  2  3  4 | class myClass: myInterface1, myInterface2, myInterface3, ...  {    } |

Рассмотрим на примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Message hello = new Message("Hello World");  hello.Print(); // Hello World    interface IMessage  {  string Text { get; set; }  }  interface IPrintable  {  void Print();  }  class Message : IMessage, IPrintable  {  public string Text { get; set; }  public Message(string text) => Text = text;  public void Print()=> Console.WriteLine(Text);  } |

В данном случае определены два интерфейса. Интерфейс IMessage определяет свойство Text, которое представляет текст сообщения. А интерфейс IPrintable определяет метод Print.

Класс Message реализует оба интерфейса и затем применяется в программе.

### **Интерфейсы в преобразованиях типов**

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Поскольку класс Message реализует интерфейс IMessage, то переменная типа IMessage может хранить ссылку на объект типа Message:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // Все объекты Message являются объектами IMessage  IMessage hello = new Message("Hello METANIT.COM");  Console.WriteLine(hello.Text); // Hello METANIT.COM    // Не все объекты IMessage являются объектами Message, необходимо явное приведение  // Message someMessage = hello; // ! Ошибка    // Интерфейс IMessage не имеет свойства Print, необходимо явное приведение  // hello.Print(); // ! Ошибка    // если hello представляет класс Message, выполняем преобразование  if (hello is Message someMessage) someMessage.Print(); |

Преобразование от класса к его интерфейсу, как и преобразование от производного типа к базовому, выполняется автоматически. Так как любой объект Message реализует интерфейс IMessage.

Обратное преобразование - от интерфейса к реализующему его классу будет аналогично преобразованию от базового класса к производному. Так как не каждый объект IMessage является объектом Message (ведь интерфейс IMessage могут реализовать и другие классы), то для подобного преобразования необходима операция приведения типов. И если мы хотим обратиться к методам класса Message, которые не определены в интерфейсе IMessage, но являются частью класса Message, то нам надо явным образом выполнить преобразование типов:

|  |  |
| --- | --- |
| 1 | if (hello is Message someMessage) someMessage.Print(); |

## **Явная реализация интерфейсов**

Кроме неявного применения интерфейсов, которое было рассмотрено в прошлой статье, существует также явная реализация интерфейса. При явной реализации указывается название метода или свойства вместе с названием интерфейса. При этом при реализации мы не можем использовать какие-либо модификаторы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | interface IAction  {  void Move();  }  class BaseAction : IAction  {  void IAction.Move() => Console.WriteLine("Move in Base Class");  } |

Следует учитывать, что при явной реализации интерфейса его методы и свойства **не являются** частью интерфейса класса. Поэтому напрямую через объект класса мы к ним обратиться не сможем:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | BaseAction baseAction1 = new BaseAction();    // baseAction1.Move(); // ! Ошибка - в BaseAction нет метода Move  // необходимо приведение к типу IAction  // небезопасное приведение  ((IAction)baseAction1).Move();  // безопасное приведение  if (baseAction1 is IAction action) action.Move();  // или так  IAction baseAction2 = new BaseAction();  baseAction2.Move(); |

В какой ситуации может действительно понадобиться явная реализация интерфейса? Например, когда класс применяет несколько интерфейсов, но они имеют один и тот же метод с одним и тем же возвращаемым результатом и одним и тем же набором параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Person : ISchool, IUniversity  {  public void Study() => Console.WriteLine("Учеба в школе или в университете");  }  interface ISchool  {  void Study();  }  interface IUniversity  {  void Study();  } |

Класс Person определяет один метод Study(), создавая одну общую реализацию для обоих примененных интерфейсов. И вне зависимости от того, будем ли мы рассматривать объект Person как объект типа ISchool или IUniversity, результат метода будет один и тот же.

Чтобы разграничить реализуемые интерфейсы, надо явным образом применить интерфейс:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Person : ISchool, IUniversity  {  void ISchool.Study() => Console.WriteLine("Учеба в школе");  void IUniversity.Study() => Console.WriteLine("Учеба в университете");  } |

Использование:

|  |  |
| --- | --- |
| 1  2  3  4 | Person person = new Person();    ((ISchool)person).Study();  ((IUniversity)person).Study(); |

Другая ситуация, когда в базовом классе уже реализован интерфейс, но необходимо в производном классе по-своему реализовать интерфейс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface IAction  {  void Move();  }  class BaseAction : IAction  {  public void Move() =>Console.WriteLine("Move in BaseAction");  }  class HeroAction : BaseAction, IAction  {  void IAction.Move() => Console.WriteLine("Move in HeroAction");  } |

Несмотря на то, что базовый класс BaseAction уже реализовал интерфейс IAction, но производный класс по-своему реализует его. Применение классов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | HeroAction action1 = new HeroAction();  action1.Move(); // Move in BaseAction  ((IAction)action1).Move(); // Move in HeroAction    IAction action2 = new HeroAction();  action2.Move(); // Move in HeroAction |

### **Модификаторы доступа**

Члены интерфейса могут иметь разные модификаторы доступа. Если модификатор доступа не public, а какой-то другой, то при реализации метода, свойства или события интерфейса в классах и структурах мы можем применять два способа. Во-первых, можно также использовать явную реализацию интерфейса, реализовав все необходимые интерфейса без модификаторов доступа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | IMovable tom = new Person("Tom");  // подписываемся на событие  tom.MoveEvent += () => Console.WriteLine($"{tom.Name} is moving");  tom.Move();    delegate void MoveHandler(); // делегат перемещения  interface IMovable  {  protected internal void Move();  protected internal string Name { get;}  protected internal event MoveHandler MoveEvent;  }  class Person : IMovable  {  string name;  // явная реализация события - дополнительно создается переменная  MoveHandler? moveEvent;  event MoveHandler IMovable.MoveEvent  {  add => moveEvent += value;  remove => moveEvent -= value;  }  // явная реализация свойства - в виде автосвойства  string IMovable.Name { get => name; }  public Person(string name) => this.name = name;  // явная реализация метода  void IMovable.Move()  {  Console.WriteLine($"{name} is walking");  moveEvent?.Invoke();  }  } |

В данном случае опять же надо учитывать, что напрямую мы можем обратиться к подобным методам, свойствам и событиям через переменную интерфейса, но не переменную класса.

Альтернативный вариант представляет неявная реализация с модификатором **public**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | Person tom = new Person("Tom");  // подписываемся на событие  tom.MoveEvent += () => Console.WriteLine($"{tom.Name} is moving");  tom.Move();    delegate void MoveHandler();    interface IMovable  {  protected internal void Move();  protected internal string Name { get;}  protected internal event MoveHandler MoveEvent;  }  class Person : IMovable  {  string name;  // явная реализация события - дополнительно создается переменная  MoveHandler? moveEvent;  // неявная реализация события с модификатором public  public event MoveHandler MoveEvent  {  add => moveEvent += value;  remove => moveEvent -= value;  }  // неявная реализация свойства - в виде автосвойства, но с модификатором public  public string Name { get => name; }  public Person(string name) => this.name = name;  // неявная реализация метода, но с модификатором public  public void Move()  {  Console.WriteLine($"{name} is walking");  moveEvent?.Invoke();  }  } |

В этом случае к методам, свойствам и событиям интейрфейса можно обращаться через переменную класса.

## **Реализация интерфейсов в базовых и производных классах**

Если класс применяет интерфейс, то этот класс должен реализовать все методы и свойства интерфейса, которые не имеют реализации по умолчанию. Однако также можно и не реализовать методы, сделав их абстрактными, переложив право их реализации на производные классы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface IMovable  {  void Move();  }  abstract class Person : IMovable  {  public abstract void Move();  }  class Driver : Person  {  public override void Move() => Console.WriteLine("Шофер ведет машину");  } |

При реализации интерфейса учитываются также методы и свойства, унаследованные от базового класса. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | IAction action = new HeroAction();  action.Move(); // Move in BaseAction    interface IAction  {  void Move();  }  class BaseAction  {  public void Move() => Console.WriteLine("Move in BaseAction");  }  class HeroAction : BaseAction, IAction{ } |

Здесь класс HeroAction реализует интерфейс IAction, однако для реализации метода Move из интерфейса применяется метод Move, унаследованный от базового класса BaseAction. Таким образом, класс HeroAction может не реализовать метод Move, так как этот метод уже определен в базовом классе BaseAction.

Следует отметить, что если класс одновременно наследует другой класс и реализует интерфейс, как в примере выше класс HeroAction, то название базового класса должно быть указано до реализуемых интерфейсов:

**class HeroAction : BaseAction, IAction**

### **Изменение реализации интерфейсов в производных классах**

Может сложиться ситуация, что базовый класс реализовал интерфейс, но в классе-наследнике необходимо изменить реализацию этого интерфейса. Что в этом случае делать? В этом случае мы можем использовать либо переопределение, либо скрытие метода или свойства интерфейса.

Первый вариант - переопределение виртуальных/абстрактных методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface IAction  {  void Move();  }  class BaseAction : IAction  {  public virtual void Move() => Console.WriteLine("Move in BaseAction");  }  class HeroAction : BaseAction  {  public override void Move() => Console.WriteLine("Move in HeroAction");  } |

В базовом классе BaseAction реализованный метод интерфейса определен как виртуальный (можно было бы также сделать его абстрактным), а в производном классе он переопределен.

При вызове метода через переменную интерфейса, если она ссылается на объект производного класса, будет использоваться реализация из производного класса:

|  |  |
| --- | --- |
| 1  2  3  4  5 | BaseAction action1 = new HeroAction();  action1.Move(); // Move in HeroAction    IAction action2 = new HeroAction();  action2.Move(); // Move in HeroAction |

Второй вариант - скрытие метода в производном классе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | interface IAction  {  void Move();  }  class BaseAction : IAction  {  public void Move() => Console.WriteLine("Move in BaseAction");    }  class HeroAction : BaseAction  {  public new void Move() => Console.WriteLine("Move in HeroAction");  } |

Также используем эти классы:

|  |  |
| --- | --- |
| 1  2  3  4  5 | BaseAction action1 = new HeroAction();  action1.Move(); // Move in BaseAction    IAction action2 = new HeroAction();  action2.Move(); // Move in BaseAction |

Так как интерфейс реализован именно в классе BaseAction, то через переменную action2 можно обратиться только к реализации метода Move из базового класса BaseAction.

Третий вариант - повторная реализация интерфейса в классе-наследнике:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface IAction  {  void Move();  }  class BaseAction : IAction  {  public void Move() => Console.WriteLine("Move in BaseAction");  }  class HeroAction : BaseAction, IAction  {  public new void Move() => Console.WriteLine("Move in HeroAction");  } |

В этом случае реализации этого метода из базового класса будет игнорироваться:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | BaseAction action1 = new HeroAction();  action1.Move(); // Move in BaseAction    IAction action2 = new HeroAction();  action2.Move(); // Move in HeroAction    HeroAction action3 = new HeroAction();  action3.Move(); // Move in HeroAction |

Также стоит отметить, что в случае с переменной action1 по-прежнему действует ранее связывание, в силу которого через эту переменную можно вызвать реализацию метода Move только из базового класса, который эта переменная представляет.

Четвертый вариант: явная реализация интерфейса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | interface IAction  {  void Move();  }  class BaseAction : IAction  {  public void Move() => Console.WriteLine("Move in BaseAction");  }  class HeroAction : BaseAction, IAction  {  public new void Move() => Console.WriteLine("Move in HeroAction");  // явная реализация интерфейса  void IAction.Move() => Console.WriteLine("Move in IAction");  } |

В этом случае для переменной IAction будет использоваться явная реализация интерфейса IAction, а для переменной HeroAction по прежнему будет использоваться неявная реализация:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | BaseAction action1 = new HeroAction();  action1.Move(); // Move in BaseAction    IAction action2 = new HeroAction();  action2.Move(); // Move in IAction    HeroAction action3 = new HeroAction();  action3.Move(); // Move in HeroAction |

## **Наследование интерфейсов**

Интерфейсы, как и классы, могут наследоваться:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | interface IAction  {  void Move();  }  interface IRunAction : IAction  {  void Run();  }  class BaseAction : IRunAction  {  public void Move()  {  Console.WriteLine("Move");  }  public void Run()  {  Console.WriteLine("Run");  }  } |

При применении этого интерфейса класс BaseAction должен будет реализовать как методы и свойства интерфейса IRunAction, так и методы и свойства базового интерфейса IAction, если эти методы и свойства не имеют реализации по умолчанию.

Однако в отличие от классов мы не можем применять к интерфейсам модификатор **sealed**, чтобы запретить наследование интерфейсов.

Также мы не можем применять к интерфейсам модификатор **abstract**, поскольку интерфейс фактически итак, как правило, предоставляет абстрактный функционал, который должен быть реализован в классе или структуре (за исключением методов и свойств с реализацией по умолчанию).

Однако методы интерфейсов могут использовать ключевое слово **new** для скрытия методов из базового интерфейса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | IAction action1 = new RunAction();  action1.Move(); // I am moving    IRunAction action2 = new RunAction();  action2.Move(); // I am running    interface IAction  {  void Move() => Console.WriteLine("I am moving");  }  interface IRunAction : IAction  {  // скрываем реализацию из IAction  new void Move() => Console.WriteLine("I am running");  }  class RunAction : IRunAction { } |

Здесь метод Move из IRunAction скрывает метод Move из базового интерфейса IAction. Это имеет смысл, если в базовом интерфейсе определена реализация по умолчанию, как в случае выше, которую нужно переопределить. И в случае выше, если переменная представляет тип IRunAction, то для метода Move вызывается реализация этого интерфейса:

|  |  |
| --- | --- |
| 1  2 | IRunAction action2 = new RunAction();  action2.Move(); // I am running |

Иначе если переменная представляет тип IAction, то для метода Move применяется реализация этого интерфейса:

|  |  |
| --- | --- |
| 1  2 | IAction action1 = new RunAction();  action1.Move(); // I am moving |

Но класс RunAction может переопределить метод Move сразу для обоих интерфейсов.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | IAction action1 = new RunAction();  action1.Move(); // I am tired    IRunAction action2 = new RunAction();  action2.Move(); // I am tired    interface IAction  {  void Move() => Console.WriteLine("I am moving");  }  interface IRunAction : IAction  {  new void Move() => Console.WriteLine("I am running");  }  class RunAction : IRunAction  {  public void Move() => Console.WriteLine("I am tired");  } |

При наследовании интерфейсов следует учитывать, что, как и при наследовании классов, производный интерфейс должен иметь тот же уровень доступа или более строгий, чем базовый интерфейс. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public interface IAction  {  void Move();  }  internal interface IRunAction : IAction  {  void Run();  } |

Но не наоборот. Например, в следующем случае мы получим ошибку, и программа не скомпилируется, так как производный интерфейс имеет менее строгий уровень доступа, нежели базовый:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | internal interface IAction  {  void Move();  }  public interface IRunAction : IAction // ошибка IRunAction может быть только internal  {  void Run();  } |

## **Интерфейсы в обобщениях**

### **Интерфейсы как ограничения обобщений**

Интерфейсы могут выступать в качестве ограничений обобщений. При этом если в качестве ограничения можно указаь только один класс, то интерфейсов можно указать несколько.

Допустим, у нас есть следующие интерфейсы и класс, который их реализует:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | interface IMessage  {  string Text { get; } // текст сообщения  }  interface IPrintable  {  void Print();  }  class Message : IMessage, IPrintable  {  public string Text { get; }  public Message(string text) => Text = text;    public void Print() => Console.WriteLine(Text);  } |

Интерфейс IMessage представляет интерфейс сообщения и определяет свойство Text для хранения текста сообщения. Интерфейс IPrintable определяет метод Print для условной печати сообщения. И непосредственно класс сообщения - класс Message реализует эти интерфейсы.

Используем выше перечисленные интерфейсы в качестве ограничений обобщенного класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Messenger<T> where T: IMessage, IPrintable  {  public void Send(T message)  {  Console.WriteLine("Отправка сообщения:");  message.Print();  }  } |

В данном случае класс условного мессенджера использует параметр T - тип, который который реализует сразу два интерфейса IMessage и IPrintable. Например, выше определен класс Message, который реализует оба интерфейса, поэтому мы можем данным типом типизировать объекты Messenger:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // создаем мессенджер  var telegram = new Messenger<Message>();  // создаем сообщение  var message = new Message("Hello World!");  // отправляем сообщение  telegram.Send(message); |

Также параметр T может представлять интерфейс, который наследуется от обоих интерфейсов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | interface IPrintableMessage: IPrintable, IMessage { }  class PrintableMessage : IPrintableMessage  {  public string Text { get; }  public PrintableMessage(string text) => Text = text;  public void Print() => Console.WriteLine(Text);  } |

В этом случае объекты Messenger мы можем типизировать типом IPrintableMessage:

|  |  |
| --- | --- |
| 1  2  3 | var telegram = new Messenger<IPrintableMessage>();  var message = new PrintableMessage("Hello World!");  telegram.Send(message); |

### **Обобщенные интерфейсы**

Как и классы, интерфейсы могут быть обобщенными:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | interface IUser<T>  {  T Id { get; }  }  class User<T> : IUser<T>  {  public T Id { get; }  public User(T id) => Id = id;  } |

Интерфейс IUser типизирован параметром T, который при реализации интерфейса используется в классе User. В частности, переменная \_id определена как T, что позволяет нам использовать для id различные типы.

Определим две реализации: одна в качестве параметра будет использовать тип int, а другая - тип string:

|  |  |
| --- | --- |
| 1  2  3  4  5 | IUser<int> user1 = new User<int>(6789);  Console.WriteLine(user1.Id); // 6789    IUser<string> user2 = new User<string>("12345");  Console.WriteLine(user2.Id); // 12345 |

Также при реализации интерфейса мы можем явным образом указать, какой тип будет использоваться для параметра T:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class IntUser : IUser<int>  {  public int Id { get; }  public IntUser(int id) => Id = id;  } |

Применение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | IUser<int> user1 = new IntUser(2345);  Console.WriteLine(user1.Id); // 2345    IntUser user2 = new IntUser(9840);  Console.WriteLine(user2.Id); // 9840 |

## **Копирование объектов. Интерфейс ICloneable**

Поскольку классы представляют ссылочные типы, то это накладывает некоторые ограничения на их использование. В частности, допустим, у нас есть следующий класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Person  {  public string Name { get; set; }  public int Age { get; set; }  public Person(string name, int age)  {  Name = name;  Age = age;  }  } |

Создадим один объект Person и попробуем скопировать его данные в другой объект Person:

|  |  |
| --- | --- |
| 1  2  3  4 | var tom = new Person("Tom", 23);  var bob = tom;  bob.Name = "Bob";  Console.WriteLine(tom.Name); // Bob |

В данном случае объекты tom и bob будут указывать на один и тот же объект в памяти, поэтому изменения свойств для переменной bob затронут также и переменную tom.

Чтобы переменная bob указывала на новый объект, но при этом имела значения из переменной tom, мы можем применить клонирование с помощью реализации интерфейса **ICloneable**:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface ICloneable  {  object Clone();  } |

### **Поверхностное копирование**

Реализация интерфейса в классе Person могла бы выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Person : ICloneable  {  public string Name { get; set; }  public int Age { get; set; }  public Person(string name, int age)  {  Name = name;  Age = age;  }  public object Clone()  {  return new Person(Name, Age);  }  } |

Использование:

|  |  |
| --- | --- |
| 1  2  3  4 | var tom = new Person("Tom", 23);  var bob = (Person)tom.Clone();  bob.Name = "Bob";  Console.WriteLine(tom.Name); // Tom |

Теперь все нормально копируется, изменения в свойствах переменной bob не сказываются на свойствах из переменной tom.

Для сокращения кода копирования мы можем использовать специальный метод **MemberwiseClone()**, который возвращает копию объекта:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Person : ICloneable  {  public string Name { get; set; }  public int Age { get; set; }  public Person(string name, int age)  {  Name = name;  Age = age;  }  public object Clone()  {  return MemberwiseClone();  }  } |

Этот метод реализует **поверхностное (неглубокое) копирование**. Однако данного копирования может быть недостаточно. Например, пусть класс Person содержит ссылку на объект класса Company:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class Person : ICloneable  {  public string Name { get; set; }  public int Age { get; set; }  public Company Work { get; set; }  public Person(string name, int age, Company company)  {  Name = name;  Age = age;  Work = company;  }  public object Clone() => MemberwiseClone();  }  class Company  {  public string Name { get; set; }  public Company(string name) => Name = name;  } |

В этом случае при копировании новая копия будет указывать на тот же объект Company:

|  |  |
| --- | --- |
| 1  2  3  4 | var tom = new Person("Tom", 23, new Company("Microsoft"));  var bob = (Person)tom.Clone();  bob.Work.Name = "Google";  Console.WriteLine(tom.Work.Name); // Google - а должно быть Microsoft |

### **Глубокое копирование**

Поверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять **глубокое копирование**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class Person : ICloneable  {  public string Name { get; set; }  public int Age { get; set; }  public Company Work { get; set; }  public Person(string name, int age, Company company)  {  Name = name;  Age = age;  Work = company;  }  public object Clone() => new Person(Name, Age, new Company(Work.Name));  }  class Company  {  public string Name { get; set; }  public Company(string name) => Name = name;  } |

## **Сортировка объектов. Интерфейс IComparable**

Большинство встроенных в .NET классов коллекций и массивы поддерживают сортировку. С помощью одного метода, который, как правило, называется Sort() можно сразу отсортировать по возрастанию весь набор данных. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[] numbers = new int[] { 97, 45, 32, 65, 83, 23, 15 };  Array.Sort(numbers);  foreach (int n in numbers)  Console.WriteLine(n);  // 15 23 32 45 65 83 97 |

Однако метод Sort по умолчанию работает только для наборов примитивных типов, как int или string. Для сортировки наборов сложных объектов применяется интерфейс **IComparable**. Он имеет всего один метод:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface IComparable  {  int CompareTo(object? o);  } |

Метод CompareTo предназначен для сравнения текущего объекта с объектом, который передается в качестве параметра object? o. На выходе он возвращает целое число, которое может иметь одно из трех значений:

* Меньше нуля. Значит, текущий объект должен находиться перед объектом, который передается в качестве параметра
* Равен нулю. Значит, оба объекта равны
* Больше нуля. Значит, текущий объект должен находиться после объекта, передаваемого в качестве параметра

Например, имеется класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Person : IComparable  {  public string Name { get;}  public int Age { get; set; }  public Person(string name, int age)  {  Name = name; Age = age;  }  public int CompareTo(object? o)  {  if(o is Person person) return Name.CompareTo(person.Name);  else throw new ArgumentException("Некорректное значение параметра");  }  } |

Здесь в качестве критерия сравнения выбрано свойство Name объекта Person. Поэтому при сравнении здесь фактически идет сравнение значения свойства Name текущего объекта и свойства Name объекта, переданного через параметр. Если вдруг объект не удастся привести к типу Person, то выбрасывается исключение.

Применение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var tom = new Person("Tom", 37);  var bob = new Person("Bob", 41);  var sam = new Person("Sam", 25);    Person[] people = { tom, bob, sam};  Array.Sort(people);    foreach (Person person in people)  {  Console.WriteLine($"{person.Name} - {person.Age}");  } |

И в данном случае мы получим следующий консольный вывод:

Bob - 41

Sam - 25

Tom - 37

Интерфейс IComparable имеет обобщенную версию, поэтому мы могли бы сократить и упростить его применение в классе Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Person : IComparable<Person>  {  public string Name { get;}  public int Age { get; set; }  public Person(string name, int age)  {  Name = name; Age = age;  }  public int CompareTo(Person? person)  {  if(person is null) throw new ArgumentException("Некорректное значение параметра");  return Name.CompareTo(person.Name);  }  } |

Аналогичным образом мы можем сравнивать по возрасту:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Person : IComparable<Person>  {  public string Name { get;}  public int Age { get; set; }  public Person(string name, int age)  {  Name = name; Age = age;  }  public int CompareTo(Person? person)  {  if(person is null) throw new ArgumentException("Некорректное значение параметра");  return Age - person.Age;  }  } |

### **Применение компаратора**

Кроме интерфейса IComparable платформа .NET также предоставляет интерфейс IComparer:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface IComparer<in T>  {  int Compare(T? x, T? y);  } |

Метод Compare предназначен для сравнения двух объектов o1 и o2. Он также возвращает три значения, в зависимости от результата сравнения: если первый объект больше второго, то возвращается число больше 0, если меньше - то число меньше нуля; если оба объекта равны, возвращается ноль.

Создадим компаратор объектов Person. Пусть он сравнивает объекты в зависимости от длины строки - значения свойства Name:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class PeopleComparer : IComparer<Person>  {  public int Compare(Person? p1, Person? p2)  {  if(p1 is null || p2 is null)  throw new ArgumentException("Некорректное значение параметра");  return p1.Name.Length - p2.Name.Length;  }  }    class Person  {  public string Name { get;}  public int Age { get; set; }  public Person(string name, int age)  {  Name = name; Age = age;  }  } |

В данном случае используется обобщенная версия интерфейса IComparer, чтобы не делать излишних преобразований типов. Применение компаратора:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var alice = new Person("Alice", 41);  var tom = new Person("Tom", 37);  var kate = new Person("Kate", 25);    Person[] people = { alice, tom, kate};  Array.Sort(people, new PeopleComparer());    foreach (Person person in people)  {  Console.WriteLine($"{person.Name} - {person.Age}");  } |

Объект компаратора указывается в качестве второго параметра метода Array.Sort(). При этом не важно, реализует ли класс Person интерфейс IComparable или нет. Правила сортировки, установленные компаратором, будут иметь больший приоритет. В начале будут идти объекты Person, у которых имена меньше, а в конце - у которых имена длиннее:

Tom - 37

Kate - 25

Alice - 41

## **Ковариантность и контравариантность обобщенных интерфейсов**

Понятия ковариантности и контравариантности связаны с возможностью использовать в приложении вместо некоторого типа другой тип, который находится ниже или выше в иерархии наследования.

Имеется три возможных варианта поведения:

* **Ковариантность**: позволяет использовать более конкретный тип, чем заданный изначально
* **Контравариантность**: позволяет использовать более универсальный тип, чем заданный изначально
* **Инвариантность**: позволяет использовать только заданный тип

C# позволяет создавать ковариантные и контравариантные обобщенные интерфейсы. Эта функциональность повышает гибкость при использовании обобщенных интерфейсов в программе. По умолчанию все обобщенные интерфейсы являются инвариантными.

Для рассмотрения ковариантных и контравариантных интерфейсов возьмем следующие классы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Message  {  public string Text { get; set; }  public Message(string text) => Text = text;  }  class EmailMessage : Message  {  public EmailMessage(string text): base(text) { }  } |

Здесь определен класс сообщения Message, который получает через конструктор текст и сохраняет его в свойство Text. А класс EmailMessage представляет условное email-сообщение и просто вызывает конструктор базового класса, передавая ему текст сообщения.

### **Ковариантные интерфейсы**

Обобщенные интерфейсы могут быть ковариантными, если к универсальному параметру применяется ключевое слово **out**. Такой параметр должен представлять тип объекта, который возвращается из метода. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | interface IMessenger<out T>  {  T WriteMessage(string text);  }  class EmailMessenger : IMessenger<EmailMessage>  {  public EmailMessage WriteMessage(string text)  {  return new EmailMessage($"Email: {text}");  }  } |

Здесь обобщенный интерфейс IMessenger представляет интерфейс мессенджера и определяет метод WriteMessage() для создания сообщения. При этом на момент определения интерфейса мы не знаем, объект какого типа будет возвращаться в этом методе. Ключевое слово **out** в определении интерфейса указывает, что данный интерфейс будет ковариантным.

Класс EmailMessenger, который представляет условную программу для отправки email-сообщений, реализует этот интерфейс и возвращает из метода WriteMessage() объект EmailMessage.

Применим данные типы в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | IMessenger<Message> outlook = new EmailMessenger();  Message message = outlook.WriteMessage("Hello World");  Console.WriteLine(message.Text); // Email: Hello World      IMessenger<EmailMessage> emailClient = new EmailMessenger();  IMessenger<Message> messenger = emailClient;  Message emailMessage = messenger.WriteMessage("Hi!");  Console.WriteLine(emailMessage.Text); // Email: Hi! |

То есть мы можем присвоить более общему типу IMessenger<Message> объект более конкретного типа EmailMessenger или IMessenger<EmailMessage>.

В то же время если бы мы не использовали ключевое слово out:

|  |  |
| --- | --- |
| 1 | interface IMessenger<T> |

то мы столкнулись бы с ошибкой в строке

|  |  |
| --- | --- |
| 1  2  3  4 | IMessenger<Message> outlook = new EmailMessenger(); // ! Ошибка    IMessenger<EmailMessage> emailClient = new EmailMessenger();  IMessenger<Message> messenger = emailClient; // ! Ошибка |

Поскольку в этом случае невозможно было бы привести объект IMessenger<EmailMessage> к типу IMessenger<Message>

При создании ковариантного интерфейса надо учитывать, что универсальный параметр может использоваться только в качестве типа значения, возвращаемого методами интерфейса. Но не может использоваться в качестве типа аргументов метода или ограничения методов интерфейса.

### **Контравариантные интерфейсы**

Для создания контравариантного интерфейса надо использовать ключевое слово **in**. Например, возьмем те же классы Message и EmailMessage и определим следующие типы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | interface IMessenger<in T>  {  void SendMessage(T message);  }  class SimpleMessenger : IMessenger<Message>  {  public void SendMessage(Message message)  {  Console.WriteLine($"Отправляется сообщение: {message.Text}");  }  } |

Здесь опять же интерфейс IMessenger представляет интерфейс мессенджера и определяет метод SendMessage() для отправки условного сообщения. Ключевое слово **in** в определении интерфейса указывает, что этот интерфейс - **контравариантный**.

Класс SimpleMessenger представляет условную программу отправки сообщений и реализует этот интерфейс. Причем в качестве типа используемого этот класс использует тип Message. То есть SimpleMessenger фактически представляет тип IMessenger<Message>.

Применим эти типы в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | IMessenger<EmailMessage> outlook = new SimpleMessenger();  outlook.SendMessage(new EmailMessage("Hi!"));    IMessenger<Message> telegram = new SimpleMessenger();  IMessenger<EmailMessage> emailClient = telegram;  emailClient.SendMessage(new EmailMessage("Hello")); |

Так как интерфейс IMessenger использует универсальный параметр с ключевым словом **in**, то он является контравариантным, поэтому в коде мы можем переменной типа IMessenger<EmailMessage> передать объект IMessenger<Message> или SimpleMessenger

Если бы ключевое слово **in** не использовалось бы, то мы не смогли бы это сделать. То есть объект интерфейса с более универсальным типом приводится к объекту интерфейса с более конкретным типом.

При создании контрвариантного интерфейса надо учитывать, что универсальный параметр контрвариантного типа может применяться только к аргументам метода, но не может применяться к возвращаемому результату метода.

### **Совмещение ковариантности и контравариантности**

Также мы можем совместить ковариантность и контравариантность в одном интерфейсе. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | interface IMessenger<in T, out K>  {  void SendMessage(T message);  K WriteMessage(string text);  }  class SimpleMessenger : IMessenger<Message, EmailMessage>  {  public void SendMessage(Message message)  {  Console.WriteLine($"Отправляется сообщение: {message.Text}");  }  public EmailMessage WriteMessage(string text)  {  return new EmailMessage($"Email: {text}");  }  } |

Фактически здесь объединены два предыдущих примера. Благодаря ковариантности/контравариантности объект класса SimpleMessenger может представлять типы IMessenger<EmailMessage, Message>, IMessenger<Message, EmailMessage>, IMessenger<Message, Message> и IMessenger<EmailMessage, EmailMessage>. Применение классов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | IMessenger<EmailMessage, Message> messenger = new SimpleMessenger();  Message message = messenger.WriteMessage("Hello World");  Console.WriteLine(message.Text);  messenger.SendMessage(new EmailMessage("Test"));    IMessenger<EmailMessage, EmailMessage> outlook = new SimpleMessenger();  EmailMessage emailMessage = outlook.WriteMessage("Message from Outlook");  outlook.SendMessage(emailMessage);    IMessenger<Message, Message> telegram = new SimpleMessenger();  Message simpleMessage = telegram.WriteMessage("Message from Telegram");  telegram.SendMessage(simpleMessage); |

# **Дополнительные возможности ООП в C#**

## **Определение операторов**

Наряду с методами в классах и структурах мы можем также определять операторы. Например, пусть у нас есть следующий класс Counter:

|  |  |
| --- | --- |
| 1  2  3  4 | class Counter  {  public int Value { get; set; }  } |

Данный класс представляет некоторый счетчик, значение которого хранится в свойстве Value.

И допустим, у нас есть два объекта класса Counter - два счетчика, которые мы хотим сравнивать или складывать на основании их свойства Value, используя стандартные операции сравнения и сложения:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Counter counter1 = new Counter { Value = 23 };  Counter counter2 = new Counter { Value = 45 };    bool result = counter1 > counter2;  Counter c3 = counter1 + counter2; |

Но на данный момент ни операция сравнения, ни операция сложения для объектов Counter не доступны. Эти операции могут использоваться для ряда примитивных типов. Например, по умолчанию мы можем складывать числовые значения, но как складывать объекты комплексных типов - классов и структур компилятор не знает. И для этого нам надо выполнить перегрузку нужных нам операторов.

Определение операторов заключается в определении в классе, для объектов которого мы хотим определить оператор, специального метода:

|  |  |
| --- | --- |
| 1  2 | public static возвращаемый\_тип operator оператор(параметры)  { } |

Этот метод должен иметь модификаторы **public static**, так как перегружаемый оператор будет использоваться для всех объектов данного класса. Далее идет название возвращаемого типа. Возвращаемый тип представляет тот тип, объекты которого мы хотим получить. К примеру, в результате сложения двух объектов Counter мы ожидаем получить новый объект Counter. А в результате сравнения двух мы хотим получить объект типа bool, который указывает истинно ли условное выражение или ложно. Но в зависимости от задачи возвращаемые типы могут быть любыми.

Затем вместо названия метода идет ключевое слово **operator** и собственно сам оператор. И далее в скобках перечисляются параметры. Бинарные операторы принимают два параметра, унарные - один параметр. И в любом случае один из параметров должен представлять тот тип - класс или структуру, в котором определяется оператор.

Например, перегрузим ряд операторов для класса Counter:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class Counter  {  public int Value { get; set; }    public static Counter operator +(Counter counter1, Counter counter2)  {  return new Counter { Value = counter1.Value + counter2.Value };  }  public static bool operator >(Counter counter1, Counter counter2)  {  return counter1.Value > counter2.Value;  }  public static bool operator <(Counter counter1, Counter counter2)  {  return counter1.Value < counter2.Value;  }  } |

Поскольку все определенные операторы - бинарные - то есть проводятся над двумя объектами, то для каждой перегрузки предусмотрено по два параметра.

Так как в случае с операцией сложения мы хотим сложить два объекта класса Counter, то оператор принимает два объекта этого класса. И так как мы хотим в результате сложения получить новый объект Counter, то данный класс также используется в качестве возвращаемого типа. Все действия этого оператора сводятся к созданию, нового объекта, свойство Value которого объединяет значения свойства Value обоих параметров:

|  |  |
| --- | --- |
| 1  2  3  4 | public static Counter operator +(Counter counter1, Counter counter2)  {  return new Counter { Value = counter1.Value + counter2.Value };  } |

Также определены две операции сравнения. Если мы определяем одну из этих операций сравнения, то мы также должны определить вторую из этих операций. Сами операторы сравнения сравнивают значения свойств Value и в зависимости от результата сравнения возвращают либо true, либо false.

Теперь используем перегруженные операторы в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Counter counter1 = new Counter { Value = 23 };  Counter counter2 = new Counter { Value = 45 };  bool result = counter1 > counter2;  Console.WriteLine(result); // false    Counter counter3 = counter1 + counter2;  Console.WriteLine(counter3.Value); // 23 + 45 = 68 |

Стоит отметить, что так как по сути определение оператора представляет собой метод, то этот метод мы также можем перегрузить, то есть создать для него еще одну версию. Например, добавим в класс Counter еще один оператор:

|  |  |
| --- | --- |
| 1  2  3  4 | public static int operator +(Counter counter1, int val)  {  return counter1.Value + val;  } |

Данный метод складывает значение свойства Value и некоторое число, возвращая их сумму. И также мы можем применить этот оператор:

|  |  |
| --- | --- |
| 1  2  3 | Counter counter1 = new Counter { Value = 23 };  int result = counter1 + 27; // 50  Console.WriteLine(result); |

Следует учитывать, что не все операторы можно определить. В частности, мы можем определить логику для следующих операторов:

* унарные операторы +x, -x, !x, ~x, ++, --, true, false
* бинарные операторы +, -, \*, /, %
* операции сравнения ==, !=, <, >, <=, >=
* поразрядные операторы &, |, ^, <<, >>
* логические операторы &&, ||

Кроме того, есть несколько операторов, которые надо определять парами:

* == и !=
* < и >
* <= и >=

И есть ряд операторов, которые нельзя перегрузить, например, операцию равенства = или тернарный оператор ?:, а также ряд других. Полный список перегружаемых операторов можно найти в [документации msdn](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/operator-overloading" \l "overloadable-operators)

### **Определение инкремента и декремента**

Следует учитывать, что в коде оператора не должны изменяться те объекты, которые передаются в оператор через параметры. Например, мы можем определить для класса Counter оператор инкремента:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static Counter operator ++(Counter counter1)  {  counter1.Value += 10;  return counter1;  } |

Поскольку оператор унарный, он принимает только один параметр - объект того класса, в котором данный оператор определен. Но это **неправильное** определение инкремента, так как оператор не должен менять значения своих параметров.

И более корректное определение оператора инкремента будет выглядеть так:

|  |  |
| --- | --- |
| 1  2  3  4 | public static Counter operator ++(Counter counter1)  {  return new Counter { Value = counter1.Value + 10 };  } |

То есть возвращается новый объект, который содержит в свойстве Value инкрементированное значение. При этом нам не надо определять отдельно операторы для префиксного и для постфиксного инкремента (а также декремента), так как одна реализация будет работать в обоих случаях.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Counter counter1 = new Counter() { Value = 10 };  Counter counter2 = counter1++;  Console.WriteLine(counter1.Value); // 20  Console.WriteLine(counter2.Value); // 10    Counter counter3 = ++counter1;  Console.WriteLine(counter1.Value); // 30  Console.WriteLine(counter3.Value); // 30 |

При операции постфиксного инкремента (counter1++) компилятор сначала создает временную переменную, в которую сохраняет текущий объект. Затем текущий объект замещает значением, полученным из функции оператора. В качестве результата операции возвращается значение временной переменной. При префиксном инкременте (++counter1) компилятор возвращает новое значение, полученное из функции оператора.

### **Определение операций true и false**

Отдельно стоит отметить определение операторов **true** и **false**. Эти операторы определяются, когда мы хотим использовать объект типа в качестве условия. Например, определим данные операторы в классе Counter:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Counter  {  public int Value { get; set; }    public static bool operator true(Counter counter1)  {  return counter1.Value != 0;  }  public static bool operator false(Counter counter1)  {  return counter1.Value == 0;  }  } |

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Counter counter = new Counter() { Value = 0 };  if (counter)  Console.WriteLine(true);  else  Console.WriteLine(false); |

Также стоит отметить, что если мы хотим использовать операцию отрицания, типа if (!counter), то нам также необходимо определить для типа операцию **!**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | Counter counter = new Counter() { Value = 2 };  if (!counter)  Console.WriteLine(true);  else  Console.WriteLine(false);    class Counter  {  public int Value { get; set; }    public static bool operator !(Counter counter1)  {  return counter1.Value == 0;  }    public static bool operator true(Counter counter1)  {  return counter1.Value != 0;  }  public static bool operator false(Counter counter1)  {  return counter1.Value == 0;  }  } |

Операция отрицания фактически синонимична операции false, поэтому содержит аналогичное условие.

## **Перегрузка операций преобразования типов**

В прошлой теме была рассмотрена тема перегрузки операторов. И с этой темой тесно связана тема перегрузки операторов преобразования типов.

Ранее мы рассматривали явные и неявные преобразования примитивных типов. Например:

|  |  |
| --- | --- |
| 1  2  3 | int x = 50;  byte y = (byte)x; // явное преобразование от int к byte  int z = y; // неявное преобразование от byte к int |

И было бы не плохо иметь возможность определять логику преобразования одних типов в другие. И с помощью перегрузки операторов мы можем это делать. Для этого в классе определяется метод следующей формы:

|  |  |
| --- | --- |
| 1  2  3  4 | public static implicit|explicit operator Тип\_в\_который\_надо\_преобразовать(исходный\_тип param)  {  // логика преобразования  } |

После модификаторов **public static** идет ключевое слово **explicit** (если преобразование явное, то есть нужна операция приведения типов) или **implicit** (если преобразование неявное). Затем идет ключевое слово **operator** и далее возвращаемый тип, в который надо преобразовать объект. В скобках в качестве параметра передается объект, который надо преобразовать.

Например, пусть у нас есть следующий класс Counter, который представляет счетчик-секундомер и который хранит количество секунд в свойстве Seconds:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Counter  {  public int Seconds { get; set; }    public static implicit operator Counter(int x)  {  return new Counter { Seconds = x };  }  public static explicit operator int(Counter counter)  {  return counter.Seconds;  }  } |

Первый оператор преобразует число - объект типа int к типу Counter. Его логика проста - создается новый объект Counter, у которого устанавливается свойство Seconds.

Второй оператор преобразует объект Counter к типу int, то есть получает из Counter число.

Примение операторов преобразования в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Counter counter1 = new Counter { Seconds = 23 };    int x = (int)counter1;  Console.WriteLine(x); // 23    Counter counter2 = x;  Console.WriteLine(counter2.Seconds); // 23 |

Поскольку операция преобразования из Counter в int определена с ключевым словом explicit, то есть как явное преобразование, то в этом случае необходимо применить операцию приведения типов:

|  |  |
| --- | --- |
| 1 | int x = (int)counter1; |

В случае с операцией преобразования от int к Counter ничего подобного делать не надо, поскольку данная операция определена с ключевым словом implicit, то есть как неявная. Какие операции преобразования делать явными, а какие неявные, в данном случае не столь важно, это решает разработчик по своему усмотрению.

Следует учитывать, что оператор преобразования типов должен преобразовывать из типа или в тип, в котором этот оператор определен. То есть оператор преобразования, определенный в типе Counter, должен либо принимать в качестве параметра объект типа Counter, либо возвращать объект типа Counter.

Рассмотрим также более сложные преобразования, к примеру, из одного составного типа в другой составной тип. Допустим, у нас есть еще класс Timer:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | class Timer  {  public int Hours { get; set; }  public int Minutes { get; set; }  public int Seconds { get; set; }  }  class Counter  {  public int Seconds { get; set; }    public static implicit operator Counter(int x)  {  return new Counter { Seconds = x };  }  public static explicit operator int(Counter counter)  {  return counter.Seconds;  }  public static explicit operator Counter(Timer timer)  {  int h = timer.Hours \* 3600;  int m = timer.Minutes \* 60;  return new Counter { Seconds = h + m + timer.Seconds };  }  public static implicit operator Timer(Counter counter)  {  int h = counter.Seconds / 3600;  int m = (counter.Seconds % 3600) / 60;  int s = counter.Seconds % 60;  return new Timer { Hours = h, Minutes = m, Seconds = s };  }  } |

Класс Timer представляет условный таймер, который хранит часы, минуты и секунды. Класс Counter представляет условный счетчик-секундомер, который хранит количество секунд. Исходя из этого мы можем определить некоторую логику преобразования из одного типа к другому, то есть получение из секунд в объекте Counter часов, минут и секунд в объекте Timer. Например, 3675 секунд по сути это 1 час, 1 минута и 15 секунд

Применение операций преобразования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Counter counter1 = new Counter { Seconds = 115 };    Timer timer = counter1;  Console.WriteLine($"{timer.Hours}:{timer.Minutes}:{timer.Seconds}"); // 0:1:55    Counter counter2 = (Counter)timer;  Console.WriteLine(counter2.Seconds); //115 |

## **Индексаторы**

**Индексаторы** позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

Формальное определение индексатора:

|  |  |
| --- | --- |
| 1  2  3  4  5 | возвращаемый\_тип this [Тип параметр1, ...]  {  get { ... }  set { ... }  } |

В отличие от свойств индексатор не имеет названия. Вместо него указывается ключевое слово **this**, после которого в квадратных скобках идут параметры. Индексатор должен иметь как минимум один параметр.

Посмотрим на примере. Допустим, у нас есть класс **Person**, который представляет человека, и класс **Company**, который представляет некоторую компанию, где работают люди. Используем индексаторы для определения класса Company:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Person  {  public string Name { get;}  public Person(string name) => Name=name;  }  class Company  {  Person[] personal;  public Company(Person[] people) => personal = people;  // индексатор  public Person this[int index]  {  get => personal[index];  set => personal[index] = value;  }  } |

Для хранения персонала компании в классе определен массив personal, который состоит из объектов Person. Для доступа к этим объектам определен индексатор:

|  |  |
| --- | --- |
| 1 | public Person this[int index] |

Индексатор в принципе подобен стандартному свойству. Во-первых, для индексатора определяется тип в данном случае тип Person. Тип индексатора определяет, какие объекты будет получать и возвращать индексатор.

Во-вторых, для индексатора определен параметр int index, через который обращаемся к элементам внутри объекта Company.

Для возвращения объекта в индексаторе определен блок **get**:

|  |  |
| --- | --- |
| 1 | get => personal[index]; |

Поскольку индексатор имеет тип Person, то в блоке get нам надо возвратить объект этого типа с помощью оператора return. Здесь мы можем определить разнообразную логику. В данном случае просто возвращаем объект из массива personal.

В блоке **set**, как и в обычном свойстве, получаем через параметр **value** переданный объект Person и сохраняем его в массив по индексу.

|  |  |
| --- | --- |
| 1 | set => personal[index] = value; |

После этого мы можем работать с объектом Company как с набором объектов Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var microsoft = new Company(new[]  {  new Person("Tom"), new Person("Bob"), new Person("Sam"), new Person("Alice")  });  // получаем объект из индексатора  Person firstPerson = microsoft[0];  Console.WriteLine(firstPerson.Name); // Tom  // переустанавливаем объект  microsoft[0] = new Person("Mike");  Console.WriteLine(microsoft[0].Name); // Mike |

Стоит отметить, что если индексатору будет передан некорректный индекс, который отсутствует в массиве person, то мы получим исключение, как и в случае обращения напрямую к элементам массива. В этом случае можно предусмотреть какую-то дополнительную логику. Например, проверять переданный индекс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class Company  {  Person[] personal;  public Company(Person[] people) => personal = people;  // индексатор  public Person this[int index]  {  get  {  // если индекс имеется в массиве  if (index >= 0 && index < personal.Length)  return personal[index]; // то возвращаем объект Person по индексу  else  throw new ArgumentOutOfRangeException(); // иначе генерируем исключение  }  set  {  // если индекс есть в массиве  if (index >= 0 && index < personal.Length)  personal[index] = value; // переустанавливаем значение по индексу  }  }  } |

Здесь в блоке get если переданный индекс имеется в массиве, то возвращаем объект по индексу. Если индекса нет в массиве, то генерируем исключение. Аналогично в блоке set устанавливаем значение по индексу, если индекс есть в массиве.

### **Индексы**

Индексатор получает набор индексов в виде параметров. Однако индексы необязательно должны представлять тип int, устанавливаемые/возвращаемые значения необязательно хранить в массиве. Например, мы можем рассматривать объект как хранилище атрибутов/свойств и передавать имя атрибута в виде строки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | User tom = new User();  // устанавливаем значения  tom["name"] = "Tom";  tom["email"] = "tom@gmail.ru";  tom["phone"] = "+1234556767";    // получаем значение  Console.WriteLine(tom["name"]); // Tom    class User  {  string name = "";  string email = "";  string phone = "";  public string this[string propname]  {  get  {  switch (propname)  {  case "name": return name;  case "email": return email;  case "phone": return phone;  default: throw new Exception("Unknown Property Name");  }  }  set  {  switch (propname)  {  case "name":  name = value;  break;  case "email":  email = value;  break;  case "phone":  phone = value;  break;  }  }  }  } |

В данном случае индексатор в классе User в качестве индекса получает строку, которая хранит название атрибута (в данном случае название поля класса).

В блоке **get** в зависимости от значения строкового индекса возвращается значение того или иного поля класса. Если передано неизвестное название, то генерируется исключение. В блоке **set** похожая логика - по индексу узнаем, для какого поля надо установить значение.

### **Применение нескольких параметров**

Также индексатор может принимать несколько параметров. Допустим, у нас есть класс, в котором хранилище определено в виде двухмерного массива или матрицы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Matrix  {  int[,] numbers = new int[,] { { 1, 2, 4 }, { 2, 3, 6 }, { 3, 4, 8 } };  public int this[int i, int j]  {  get => numbers[i, j];  set => numbers[i, j] = value;  }  } |

Теперь для определения индексатора используются два индекса - i и j. И в программе мы уже должны обращаться к объекту, используя два индекса:

|  |  |
| --- | --- |
| 1  2  3  4 | Matrix matrix = new Matrix();  Console.WriteLine(matrix[0, 0]);  matrix[0, 0] = 111;  Console.WriteLine(matrix[0, 0]); |

Следует учитывать, что индексатор не может быть статическим и применяется только к экземпляру класса. Но при этом индексаторы могут быть виртуальными и абстрактными и могут переопределяться в произодных классах.

### **Блоки get и set**

Как и в свойствах, в индексаторах можно опускать блок get или set, если в них нет необходимости. Например, удалим блок set и сделаем индексатор доступным только для чтения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Matrix  {  int[,] numbers = new int[,] { { 1, 2, 4 }, { 2, 3, 6 }, { 3, 4, 8 } };  public int this[int i, int j]  {  get => numbers[i, j];  }  } |

Также мы можем ограничивать доступ к блокам get и set, используя модификаторы доступа. Например, сделаем блок set приватным:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Matrix  {  int[,] numbers = new int[,] { { 1, 2, 4 }, { 2, 3, 6 }, { 3, 4, 8 } };  public int this[int i, int j]  {  get => numbers[i, j];  private set => numbers[i, j] = value;  }  } |

### **Перегрузка индексаторов**

Подобно методам индексаторы можно перегружать. В этом случае также индексаторы должны отличаться по количеству, типу или порядку используемых параметров. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | var microsoft = new Company(new Person[] { new("Tom"), new("Bob"), new("Sam") });    Console.WriteLine(microsoft[0].Name); // Tom  Console.WriteLine(microsoft["Bob"].Name); // Bob  class Person  {  public string Name { get;}  public Person(string name) => Name=name;  }  class Company  {  Person[] personal;  public Company(Person[] people) => personal = people;  // индексатор  public Person this[int index]  {  get => personal[index];  set => personal[index] = value;  }    public Person this[string name]  {  get  {  foreach (var person in personal)  {  if (person.Name == name) return person;  }  throw new Exception("Unknown name");  }  }  } |

В данном случае класс Company содержит две версии индексатора. Первая версия получает и устанавливает объект Person по индексу, а вторая - только получае объект Person по его имени.

## **Переменные-ссылки и возвращение ссылки**

Кроме параметров метода, которые с помощью модификатора **ref** позволяют передавать значение по ссылке, C# также позволяет с помощью ключевого слова **ref** возвращать ссылку из метода и определять переменную, которая будет хранить ссылку.

### **Переменная-ссылка**

Для определения локальной переменной-ссылки (ref local) перед ее типом ставится ключевое слово **ref**:

|  |  |
| --- | --- |
| 1  2 | int x = 5;  ref int xRef = ref x; |

Здесь переменная xRef указывает не просто на значение переменной x, а на область в памяти, где располагается эта переменная. Для этого перед x также указывается ref.

При этом мы не можем просто определить переменную-ссылку, нам обязательно надо присвоить ей некоторое значение. Так, следующий код вызовет ошибку:

|  |  |
| --- | --- |
| 1 | ref int xRef; // ошибка |

Получив ссылку, мы можем манипулировать значением по этой ссылке. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int x = 5;  ref int xRef = ref x;  Console.WriteLine(x); // 5  xRef = 125;  Console.WriteLine(x); // 125  x = 625;  Console.WriteLine(xRef); // 625 |

### **Ссылка как результат функции**

Для возвращения из функции ссылки в сигнатуре функции перед возвращаемым типом, а также после оператора **return** следует указать ключевое слово **ref**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int[] numbers = { 1, 2, 3, 4, 5, 6, 7 };  ref int numberRef = ref Find(4, numbers); // ищем число 4 в массиве  numberRef = 9; // заменяем 4 на 9  Console.WriteLine(numbers[3]); // 9    ref int Find(int number, int[] numbers)  {  for (int i = 0; i < numbers.Length; i++)  {  if (numbers[i] == number)  {  return ref numbers[i]; // возвращаем ссылку на адрес, а не само значение  }  }  throw new IndexOutOfRangeException("число не найдено");  } |

В методе Find ищем число в массиве, но вместо самого значения числа возвращаем ссылку на него в памяти. Для этого в сигнатуре метода в качестве типа результата функции указывается не просто **int**, а **ref int**.

Кроме того, в самом методе после слова return также ставится ref:

|  |  |
| --- | --- |
| 1 | return ref numbers[i]; |

Тем самым мы получаем не просто значение, а ссылку на объект в памяти.

В методе Main для определения переменной, которая будет содержать ссылку, используется ключевое слово ref. При вызове метода также указывается слово ref:

|  |  |
| --- | --- |
| 1 | ref int numberRef = ref Find(7, numbers); |

В итоге переменная numberRef будет содержать ссылку на объект int, и через данную переменную в последствиии мы можем изменить объект по этой ссылке.

Другой пример - возвращение ссылки на максимальное число из двух:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int a = 5;  int b = 8;  ref int pointer = ref Max(ref a, ref b);  pointer = 34; // меняем значением максимального числа  Console.WriteLine($"a: {a} b: {b}"); // a: 5 b: 34    ref int Max(ref int n1, ref int n2)  {  if (n1 > n2)  return ref n1;    else  return ref n2;  } |

Стоит обратить внимание, что параметры метода в этом случае определены с ключевым словом ref.

При определении метода, который возвращает ссылку, следует учитывать, что такой метод естественно не может иметь тип void. Кроме того, такой метод не может возвращать:

* Значение null
* Константу
* Значение перечисления enum
* Свойство класса или структуры
* Поле для чтения (которое имеет модификатор read-only)

## **Методы расширения**

Методы расширения (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса. Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем, поскольку у нас нет доступа к исходному коду. Либо если мы не можем использовать стандартный механизм наследования, например, если классы определенны с модификатором sealed.

Например, нам надо добавить для типа string новый метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | string s = "Привет мир";  char c = 'и';  int i = s.CharCount(c);  Console.WriteLine(i);    public static class StringExtension  {  public static int CharCount(this string str, char c)  {  int counter = 0;  for (int i = 0; i < str.Length; i++)  {  if (str[i] == c)  counter++;  }  return counter;  }  } |

Для того, чтобы создать метод расширения, вначале надо создать статический класс, который и будет содержать этот метод. В данном случае это класс StringExtension. Затем объявляем статический метод. Суть нашего метода расширения - подсчет количества определенных символов в строке.

Собственно метод расширения - это обычный статический метод, который в качестве первого параметра всегда принимает такую конструкцию: this имя\_типа название\_параметра, то есть в нашем случае this string str. Так как наш метод будет относиться к типу string, то мы и используем данный тип.

Затем у всех строк мы можем вызвать данный метод:

|  |  |
| --- | --- |
| 1 | int i = s.CharCount(c); |

Причем нам уже не надо указывать первый параметр. Значения для остальных параметров передаются в обычном порядке.

Применение методов расширения очень удобно, но при этом надо помнить, что метод расширения никогда не будет вызван, если он имеет ту же сигнатуру, что и метод, изначально определенный в типе.

Также следует учитывать, что методы расширения действуют на уровне пространства имен. То есть, если добавить в проект другое пространство имен, то метод не будет применяться к строкам, и в этом случае надо будет подключить пространство имен метода через директиву using.

# **Частичные классы, методы и свойства**

### **Частичные классы**

Классы могут быть частичными. То есть мы можем иметь несколько файлов с определением одного и того же класса, и при компиляции все эти определения будут скомпилированы в одно.

Например, определим в проекте два файла с кодом. Не столь важно как эти файлы будут называться, например, "PersonMove.cs" и "PersonEat.cs". Например, в **PersonMove.cs** определим следующий класс:

|  |  |
| --- | --- |
| 1  2  3  4 | public partial class Person  {  public void Move() => Console.WriteLine("Человек идет");  } |

А в другом файле - в **PersonEat.cs** определим следующий класс:

|  |  |
| --- | --- |
| 1  2  3  4 | public partial class Person  {  public void Eat() => Console.WriteLine("Человек ест");  } |

Таким образом, два файла в проекте содержит определение одного и того же класса Person, которые содержат два разных метода. И оба определенных здесь класса являются частичными. Для этого они определяются с ключевым словом **partial**.

Затем мы можем использовать все методы класса Person:

|  |  |
| --- | --- |
| 1  2  3 | var tom = new Person();  tom.Move();  tom.Eat(); |

Консольный вывод:

Человек идет

Человек ест

Таким образом, несколько определений частичного класса объединяются в одно. Стоит отметить, что частичные классы необязательно надо размещать в разных файлах. Они могут помещаться и в один и тот же файл и даже быть размещены рядом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | var tom = new Person();  tom.Move();  tom.Eat();  // Console.WriteLine(tom.Name);    public partial class Person  {  public void Eat() => Console.WriteLine("Человек ест");  }  public partial class Person  {  public void Move() => Console.WriteLine("Человек идет");  } |

Тем не менее смысл частичных классов состоит в том, что определяя одну версию частичного класса, мы можем не знать, что будет делать другая версия частичного класса. Нередко частичные классы применяются при автогенерации кода, например, по определению пользовательского интерфейса, когда программист пишет основную логику взаимодействия программы с пользователем, а компилятор создает другой частичный класс с логикой отрисовки и позиционирования элементов управления в приложении.

### **Частичные методы**

Частичные классы могут содержать частичные методы (partial methods). Такие методы также опреляются с ключевым словом **partial**. Причем определение частичного метода без тела метода находится в одном частичном классе, а реализация этого же метода - в другом частичном классе. Концепция частичных методов в некотором роде похожа на разделение функций на прототипы и реализации в таких языках как C и C++.

Например, пусть у нас будет следующий класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5 | // объявление частичного класса  public partial class Person  {  public partial void Say(string message); // объявление частичного метода  } |

Частичный метод Say не имеет тела и представляет лишь объявление. То есть компилятор, встретив подобный метод, будет знать, что метод принимает строку и ничего не возвращает. Причем на момент определения этого класса неизвестно, что представляет собой этот метод, какие действия он будет выполнять. Тем не менее мы знаем список его параметров и можем вызвать его в программе.

Также определим второй класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // реализация частичного класса  public partial class Person  {  // реализация частичного метода  public partial void Say(string message)  {  Console.WriteLine(message);  }  } |

Здесь уже определена реализация метода Say(). И когда мы будем вызывать метод Say, будет выполняться эта реализация.

Стоит отметить, что по умолчанию к частичным методам применяется ряд ограничений:

* Они не могут иметь модификаторы доступа
* Они имеют тип **void**
* Они не могут иметь out-параметры
* Они не могут иметь модификаторы virtual, override, sealed, new или extern

Если же они не соответствуют какому-то из этих ограничений, то для них должна быть предоставлена реализация. Как, например, в предыдущем примере частичные методы применяют модификатор public, поэтому предоставляется реализация.

### **Частичные свойства и индексаторы**

Язык C# уже давно позволял определять частичные методы (partial methods). Начиная с версии C# 13 также можно определять частичные свойства и индексаторы. То есть мы можем разделить определение свойства/индексатора на объявление и реализацию и разместить их в разных файлах. При этом сигнатуры объявления и реализации должны совпадать. Единственное ограничение - нельзя определять частичные автосвойства. Если свойство, объявленное со словом partial, не имеет реализации, то оно рассматривается как объявление свойства.

Пример частичного свойства:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | var tom = new Person("Tom");    Console.WriteLine(tom.Name);    // объявление класса  partial class Person{  public partial string Name { get; set; } // объявление частичного свойства  }    // реализация класса  partial class Person{  private string name; // поле для хранения значения свойства    public Person(string name) => this.name = name;    public partial string Name // реализация частичного свойства  {  get => name;  set => name = value;  }  } |

Здесь определен частичный класс Person. Причем сначала идет только объявление класса Person:

|  |  |
| --- | --- |
| 1  2  3 | partial class Person{  public partial string Name { get; set; } // объявление частичного свойства  } |

В этом объявлении таже только объявление частичного свойства - Name. И хотя это объявление похоже на автоматическое свойство, это не автосвойство, а именно объявление свойства, так как к нему применяется модификатор **partial**

Далее идет собственно реализация класса и в том числе реализация свойства:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | partial class Person{  private string name; // поле для хранения значения свойства    public Person(string name) => this.name = name;    public partial string Name // реализация частичного свойства  {  get => name;  set => name = value;  }  } |

Хотя в данном случае для простоты объявление и реализация класса размещены рядом друг с другом, в реальности их можно разместить в разных файлах.

Пример с частичными индексаторами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | var buffer = new Buffer();    // некоторые действия с индексатором  for(var i=0; i < Buffer.COUNT; ++i){  buffer[i] = i \* i;  }    for(var i=0; i < Buffer.COUNT; ++i){  Console.WriteLine(buffer[i]);  }  // объявление класса  partial class Buffer{  public partial int this[int index] { get; set; } // объявление свойства-индексатора  }    // реализация класса  partial class Buffer{  public const int COUNT = 6;  private int[] data = new int[COUNT];  public partial int this[int index] // реализация свойства-индексатора  {  get => data[index];  set => data[index] = value;  }  } |

В данном случае частичный индексатор определен в частичном классе Buffer. Фактически индексатор является надстройкой над массивом data.

## **Анонимные типы**

Анонимные типы позволяют создать объект с некоторым набором свойств без определения класса. Анонимный тип определяется с помощью ключевого слова **var** и инициализатора объектов:

|  |  |
| --- | --- |
| 1  2 | var user = new { Name = "Tom", Age = 34 };  Console.WriteLine(user.Name); |

В данном случае user - это объект анонимного типа, у которого определены два свойства Name и Age. И мы также можем использовать его свойства, как и у обычных объектов классов. Однако тут есть ограничение - свойства анонимных типов доступны только для чтения.

При этом во время компиляции компилятор сам будет создавать для него имя типа и использовать это имя при обращении к объекту. Нередко анонимные типы имеют имя наподобие "<>f\_\_AnonymousType0'2".

Для исполняющей среды CLR анонимные типы будут также, как и классы, представлять ссылочный тип.

Если в программе используются несколько объектов анонимных типов с одинаковым набором свойств, то для них компилятор создаст одно определение анонимного типа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var user = new { Name = "Tom", Age = 34 };  var student = new { Name = "Alice", Age = 21 };  var manager = new { Name = "Bob", Age = 26, Company = "Microsoft" };    Console.WriteLine(user.GetType().Name); // <>f\_\_AnonymousType0'2  Console.WriteLine(student.GetType().Name); // <>f\_\_AnonymousType0'2  Console.WriteLine(manager.GetType().Name); // <>f\_\_AnonymousType1'3 |

Здесь user и student будут иметь одно и то же определение анонимного типа. Однако подобные объекты нельзя преобразовать к какому-нибудь другому типу, например, классу, даже если он имеет подобный набор свойств.

Следует учитывать, что свойства анонимного объекта доступны для установки только в инициализаторе. Вне инициализатора присвоить им значение мы не можем. Поэтому, например, в следующем случае мы столкнемся с ошибкой:

|  |  |
| --- | --- |
| 1  2 | var student = new { Name = "Alice", Age = 21 };  student.Age = 32; // ! Ошибка |

Кроме использованной выше формы инициализации, когда мы присваиваем свойствам некоторые значения, также можно использовать **инициализаторы с проекцией** (projection initializers), когда мы можем передать в инициализатор некоторые идиентификаторы, имена которых будут использоваться как названия свойств:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Person tom = new Person("Tom");  int age = 34;  var student = new { tom.Name, age }; // инициализатор с проекцией  Console.WriteLine(student.Name);  Console.WriteLine(student.age);    class Person  {  public string Name { get; set; }  public Person(string name) => Name = name;  } |

В данном случае определение анонимного объекта фактически будет идентично следующему:

|  |  |
| --- | --- |
| 1 | var student = new { Name = tom.Name, age = age}; |

Названия свойств и переменных (Name и age) будут использоваться в качесте названий свойств объекта.

Также можно определять массивы объектов анонимных типов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var people = new[]  {  new {Name="Tom"},  new {Name="Bob"}  };  foreach(var p in people)  {  Console.WriteLine(p.Name);  } |

Зачем нужны анонимные типы? Иногда возникает задача использовать один тип в одном узком контексте или даже один раз. Создание класса для подобного типа может быть избыточным. Если нам захочется добавить свойство, то мы сразу же на месте анонимного объекта это можем сделать. В случае с классом придется изменять еще и класс, который может больше нигде не использоваться. Типичная ситуация - получение результата выборки из базы данных: объекты используются только для получения выборки, часто больше нигде не используются, и классы для них создавать было бы излишне. А вот анонимный объект прекрасно подходит для временного хранения выборки.

## **Кортежи**

Кортежи предоставляют удобный способ для работы с набором значений, который был добавлен в версии C# 7.0.

Кортеж представляет набор значений, заключенных в круглые скобки:

|  |  |
| --- | --- |
| 1 | var tuple = (5, 10); |

В данном случае определен кортеж tuple, который имеет два значения: 5 и 10. В дальнейшем мы можем обращаться к каждому из этих значений через поля с названиями:

|  |  |
| --- | --- |
| 1 | Item[порядковый\_номер\_поля\_в\_кортеже] |

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var tuple = (5, 10);  Console.WriteLine(tuple.Item1); // 5  Console.WriteLine(tuple.Item2); // 10  tuple.Item1 += 26;  Console.WriteLine(tuple.Item1); // 31 |

В данном случае тип определяется неявно. Но мы также можем явным образом указать для переменной кортежа тип:

|  |  |
| --- | --- |
| 1 | (int, int) tuple = (5, 10); |

Так как кортеж содержит два числа, то в определении типа нам надо указать два числовых типа. Или другой пример определения кортежа:

|  |  |
| --- | --- |
| 1 | (string, int, double) person = ("Tom", 25, 81.23); |

Первый элемент кортежа в данном случае представляет строку, второй элемент - тип int, а третий - тип double.

Мы также можем дать названия полям кортежа:

|  |  |
| --- | --- |
| 1  2  3 | var tuple = (count:5, sum:10);  Console.WriteLine(tuple.count); // 5  Console.WriteLine(tuple.sum); // 10 |

Теперь чтобы обратиться к полям кортежа используются их имена, а не названия Item1 и Item2.

Мы даже можем выполнить декомпозицию кортежа на отдельные переменные:

|  |  |
| --- | --- |
| 1  2  3 | var (name, age) = ("Tom", 23);  Console.WriteLine(name); // Tom  Console.WriteLine(age); // 23 |

Одной из задач, которую позволяет элегантно решить кортеж - это обмен значениями:

|  |  |
| --- | --- |
| 1  2  3  4  5 | string main = "Java";  string second = "C#";  (main, second) = (second, main);  Console.WriteLine(main); // C#  Console.WriteLine(second); // Java |

Что можно использовать, например, при простейшей сортировке массива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | int[] nums = { 54, 7, -41, 2, 4, 2, 89, 33, -5, 12 };    // сортировка  for (int i = 0; i < nums.Length - 1; i++)  {  for (int j = i + 1; j < nums.Length; j++)  {  if (nums[i] > nums[j])  (nums[i], nums[j]) =(nums[j], nums[i]);  }  }    // вывод  Console.WriteLine("Вывод отсортированного массива");  for (int i = 0; i < nums.Length; i++)  {  Console.WriteLine(nums[i]);  } |

### **Кортеж как результат метода**

Кортежи могут выступать в качестве результата функции. Например, одной из распространенных ситуаций является возвращение из функции двух и более значений, в то время как функция может возвращать только одно значение. И кортежи представляют оптимальный способ для решения этой задачи:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var tuple = GetValues();  Console.WriteLine(tuple.Item1); // 1  Console.WriteLine(tuple.Item2); // 3    (int, int) GetValues()  {  var result = (1, 3);  return result;  } |

Здесь определен метод GetValues(), который возвращает кортеж. Кортеж определяется как набор значений, помещенных в круглые скобки. И в данном случае мы возвращаем кортеж из двух элементов типа int, то есть два числа.

Другой пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | var tuple = GetValuesData(new int[] { 1, 2, 3, 4, 5, 6, 7 });  Console.WriteLine(tuple.count);  Console.WriteLine(tuple.sum);    (int sum, int count) GetValuesData(int[] numbers)  {  var result = (sum: 0, count: numbers.Length);  foreach(var n in numbers)  {  result.sum += n;  }  return result;  } |

### **Кортеж как параметр метода**

И также кортеж может передаваться в качестве параметра в метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | PrintPerson(("Tom", 37)); // Tom - 37  PrintPerson(("Bob", 41)); // Bob - 41      void PrintPerson((string name, int age) person)  {  Console.WriteLine($"{person.name} - {person.age}");  } |

Здесь в метод PrintPerson передается кортеж из двух элементов, первый из которых предоставляет строку, а второй - значение типа int.

## **Records**

Records представляют новый ссылочный тип, который появился в C#9. Ключевая особенность records состоит в том, что они могут представлять неизменяемый (immutable) тип, который по умолчанию обладает рядом дополнительных возможностей по сравнению с классами и структурами. Зачем нам нужны неизменяемые типы? Такие типы более безопасны в тех ситуациях, когда нам надо гарантировать, что данные объекта не будут изменяться. В .NET в принципе уже есть неизменяемые типы, например, String.

Стоит отметить, что начиная с версии C# 10 добавлена поддержка структур record, соответственно мы можем создавать record-классы и record-структуры.

Для определения records используется ключевое слово **record**. Если определяется класс record, то ключевое слово class можно неиспользовать при определении типа:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public record Person  {  public string Name { get; set; }  public Person(string name) => Name = name;  } |

или так

|  |  |
| --- | --- |
| 1  2  3  4  5 | public record class Person  {  public string Name { get; set; }  public Person(string name) => Name = name;  } |

При определении структуры record при объявлении типа надо использовать ключевое слово **struct**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public record struct Person  {  public string Name { get; set; }  public Person(string name) => Name = name;  } |

Хотя типы record предназначены для создания неизменяемых типов, однако одно только применение ключевого слова **record** не гарантирует неизменяемость объектов record. Они являются неизменяемыми (immutable) только при определенных условиях. Например, мы можем написать так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var person = new Person("Tom");  person.Name = "Bob";  Console.WriteLine(person.Name); // Bob - данные изменились    public record Person  {  public string Name { get; set; }  public Person(string name) => Name = name;  } |

При выполнении этого кода не возникнет никакой ошибки, мы спокойно сможем изменять значения свойств объекта Person. Чтобы сделать его действительно неизменяемым, для свойств вместо обычных сеттеров надо использовать модификатор **init**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var person = new Person("Tom");  person.Name = "Bob"; // ! ошибка - свойство изменить нельзя    public record Person  {  public string Name { get; init; }  public Person(string name) => Name = name;  } |

В данном случае мы получим ошибку при попытке изменить значение свойств объекта Person.

Во многим records похожи на обычные классы и структуры, например, они могут абстрактными, их также можно наследовать либо запрещать наследование с помощью оператора sealed. Тем не менее есть и ряд отличий. Рассмотрим некоторые основные отличия records от стандартных классов и структур.

### **Сравнение на равенство**

При определении record компилятор генерирует метод **Equals()** для сравнения с другим объектом. При этом сравнение двух records производится на основе их значений. Например, рассмотрим следующий пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | var person1 = new Person("Tom");  var person2 = new Person("Tom");  Console.WriteLine(person1.Equals(person2)); // true    var user1 = new User("Tom");  var user2 = new User("Tom");  Console.WriteLine(user1.Equals(user2)); // false    public record Person  {  public string Name { get; init; }    public Person(string name) => Name = name;  }  public class User  {  public string Name { get; init; }  public User(string name) => Name = name;  } |

В данном случае при сравнении двух объектов record Person мы увидим, что они равны, так как их значения (значения свойств Name) равны. Однако в случае с объектами класса User, которые имеют те же одинаковые значения мы увидим, что они не равны. Так как сравнение records производится **по значению**.

Кроме того, для record уже по умолчанию реализованы операторы **==** и **!=**, которые также сравнивают две record по значению:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var person1 = new Person("Tom");  var person2 = new Person("Tom");  Console.WriteLine(person1 == person2); // true    var user1 = new User("Tom");  var user2 = new User("Tom");  Console.WriteLine(user1 == user2); // false |

### **Оператор with**

В отличие от классов records поддерживают инициализацию с помощью оператора **with**. Он позволяет создать одну record на основе другой record:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | var tom = new Person("Tom", 37);  var sam = tom with { Name = "Sam" };  Console.WriteLine($"{sam.Name} - {sam.Age}"); // Sam - 37    public record Person  {  public string Name { get; init; }  public int Age { get; init; }  public Person(string name, int age)  {  Name = name; Age = age;  }  } |

После record, значения которой мы хотим скопировать, указывается оператор **with**, после которого в фигурных скобках указываются значения для тех свойств, которые мы хотим изменить. Так, в данном случае переменная sam получает для свойства Age значение из tom, а свойство Name изменяется.

Эта возможность может быть особенно актуальна, если в record, которую мы хотим скопировать, множество свойств, из которых мы хотим поменять одно-два.

Если надо скопировать значения всех свойств, то можно оставить пустые фигурные скобки:

|  |  |
| --- | --- |
| 1  2 | var person1 = new Person("Tom", 37);  var person2 = person1 with { }; |

### **Позиционные records**

Records могут принимать данные для свойств через конструктор, и в этом случае мы можем сократить их определение. Например, пусть у нас есть следующая record Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public record Person  {  public string Name { get; init; }  public int Age { get; init; }  public Person(string name, int age)  {  Name = name; Age = age;  }  public void Deconstruct(out string name, out int age) => (name, age) = (Name, Age);  } |

Кроме конструктора здесь реализован деконструктор, который позволяет разложить объект Person на кортеж значений. И мы могли бы применить ее, например, следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var person = new Person ("Tom", 37);  Console.WriteLine(person.Name); // Tom    var (personName, personAge) = person;    Console.WriteLine(personAge); // 37  Console.WriteLine(personName); // Tom |

Выше определенную record Person можно сократить до позиционной record:

|  |  |
| --- | --- |
| 1 | public record Person(string Name, int Age); |

Это все определение типа. То есть мы говорим, что для типа Person будет создаваться конструктор, который принимает два параметра и присваивает их значения соответственно свойствам Name и Age, и что также автоматически будет создаваться деконструктор. Ее использование будет аналогично:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var person = new Person("Tom", 37);  Console.WriteLine(person); // Tom    var (personName, personAge) = person;    Console.WriteLine(personAge); // 37  Console.WriteLine(personName); // Tom    public record Person(string Name, int Age); |

При необходимости также можно совмещать стандартное определение свойств и определение свойств через конструктор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var person = new Person("Tom", 37) { Company = "Google"};  Console.WriteLine(person.Company); // Google  person.Company = "Microsoft";  Console.WriteLine(person.Company); // Microsoft    public record Person(string Name, int Age)  {  public string Company { get; set; } = "";  } |

### **Позиционные структуры для чтения**

Следует отметить различие между позиционными классами и структурами record. Свойства класса record, которые устанавливаются через параметры конструктора, по умолчанию будут иметь модификатор **init**. То есть после установки их значений через конструктор, мы больше не сможем их изменить:

|  |  |
| --- | --- |
| 1  2  3  4 | var person = new Person("Tom", 37);  person.Name = "Bob"; // ! Ошибка - значение нельзя изменить    public record Person(string Name, int Age); |

Стоит отметить, что это относится только к тем свойствам, которые устанавливаются через конструктор.

Однако для позиционных структур record свойства будут иметь стандартные сеттеры, которые позволят изменять значения свойств:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var person = new Person("Tom", 37);  person.Name = "Bob";  Console.WriteLine(person.Name); // Bob - значение изменилось  // структура record  public record struct Person(string Name, int Age); |

Чтобы для подобных свойств структуры record использовался модификатор **init** вместо обычных сеттеров, такую структуру надо определить с ключевым словом **readonly**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var person = new Person("Tom", 37);  person.Name = "Bob"; // ! Ошибка - значение свойства нельзя изменить    // структура record доступна только для чтения  public readonly record struct Person(string Name, int Age); |

### **ToString**

Небольшим преимуществом типов record также является то, что для них уже по умолчанию реализован метод **ToString()**, который выводит состояние объекта в отформатированном виде:

|  |  |
| --- | --- |
| 1  2  3  4 | var person = new Person("Tom", 37);  Console.WriteLine(person); // Person {Name = Tom, Age = 37}    public record Person(string Name, int Age); |

### **Наследование**

Как и обычные классы record-классы могут наследоваться:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var tom = new Person("Tom", 37);  var bob = new Employee("Bob", 41, "Microsoft");  Console.WriteLine(tom); // Person {Name = Tom, Age = 37}  Console.WriteLine(bob); // Person {Name = Bob, Age = 41, Company = Microsoft}    public record Person(string Name, int Age);  public record Employee(string Name, int Age, string Company) : Person(Name, Age); |

В данном случае класс record Employee наследуется от Person.

## **ref-структуры**

Структуры с модификатором **ref** (далее ref-структуры) представляют такие типы, которые размещаются в стеке. В то время как обычные структуры могут быть упакованы в ссылочные типы и помещены в управляемой куче (хипе), ref-структуры размещаются только в стеке. Это делает ref-структуры удобным инструментом в различных сценариях, где нужна высокая производительность. Среди встроенных типов .NET к ref-структурам, в частности, относят ReadOnlySpan<char> и ReadOnlySpan<char>.

Основные особенности ref-структур:

* Память под ref-структуры выделяется только в стеке, в кучу они не перемещаются
* ref-структуры не могут быть упакованы в ссылочные типы (иначе это потребовало бы размещение такиз структур в куче). Поэтому их нельзя преобразовать в ссылочный тип, например, в тип интерфейса, даже если они реализуют этот интерфейс
* ref-структуры характеризуются большей безопасностью, поскольку размещаются в стеке и соответственно не страдают от утечек памяти и других проблем, связанных с управляемой кучей.
* В силу предыдущих пунктов часто применяются, где важна скорость и производительность. Поскольку типы ref-структуры всегда выделяются в стеке, выделение и освобождение происходят намного быстрее, чем выделения на основе кучи. Это снижает нагрузку на сборщик мусора, что приводит к более предсказуемой производительности.
* Детерминированное время жизни: поскольку объекты ref-структуры выделяются в стеке, их время жизни является детерминированным — они автоматически освобождаются, когда метод, в котором они были созданы, завершает работу. Это делает ref struct идеальным для временных объектов, которым не нужно находиться вне области действия метода

В то же время ref-структуры имеют ряд ограничений:

* ref-структура не может представлять элемент массива
* ref-структура не может представлять поле класса или обычной структуры
* ref-структура не может быть упакована в System.ValueType или System.Object
* ref-структура не может быть захвачена в лямбда-выражении или локальной функции

### **Определение ref-структур**

ref-структуры определяются как обычные структуры только с модификатором **ref**. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var tom = new Person("Tom");  Console.WriteLine(tom.Name);    ref struct Person  {  public Person(string name) => Name = name;  public string Name {get;}  } |

Более практический пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};    Sum summator = new Sum(numbers, 1..3);  Console.WriteLine(summator.Get()); // 5    summator = new Sum(numbers, 2..4);  Console.WriteLine(summator.Get()); // 7    ref struct Sum  {  ReadOnlySpan<int> data;    public Sum(int[] numbers, Range range)  {  data = numbers[range];  }  public int Get()  {  int result = 0;  foreach(var n in data)  {  result += n;  }  return result;  }  } |

Здесь структура Sum вычисляет сумму значений коллекции на некотором диапазоне. Диапазон и коллекция в виде массива передаются через конструктор. Диапазон представляет тип **Range**. В конструкторе получаем нужный диапазон массива в объект типа ReadOnlySpan<int>. Для передачи диапазона в конструкторе применяется выражение 1..3, то есть получаем иапазон из элементов от 1 индекса до 3-го (не включая). В методе Get в цикле вычисляем сумму значений на указанном диапазоне.

### **Реализация интерфейсов в ref-структурах**

До C# 13 ref-структуры не могли реализовывать интерфейсы. Начиная с C# 13, они могут. Однако, в силу ограничений ref-структур, такая структура не может быть преобразован в тип интерфейса. Это преобразование является преобразованием упаковки и может нарушить безопасность ref. ref-структуры также не могут объявлять методы, которые явно реализуют методы интерфейса. Кроме того, ref-структуры должны реализовывать все методы, объявленные в интерфейсе, включая методы с реализацией по умолчанию. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | Shape myShape = new();  myShape.Draw(); // Draw some shape    // Draw\_Shape(myShape); // ! Ошибка - Так нельзя  // var shape2 = (IShape)shape; // ! Ошибка - Так нельзя    void Draw\_Shape(IShape shape)  {  shape.Draw();  }    interface IShape  {  void Draw();  }    ref struct Shape : IShape  {  public void Draw()  {  Console.WriteLine("Draw some shape");  }  } |

Обратите внимание, что мы не можем преобразовать значение ref-структуры (в данном случае Shape) к типу интерфейса (IShape):

|  |  |
| --- | --- |
| 1 | var shape2 = (IShape)shape |

Аналогично мы не можем использовать значение ref-структуры там, где требуется объект интерфейса (так как все равно идет преобразование, только неявно), как в случае с методом Draw\_Shape:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // var shape2 = (IShape)shape; // ! Ошибка - Так нельзя    void Draw\_Shape(IShape shape)  {  shape.Draw();  } |

# **Pattern matching**

## **Паттерн типов**

**Pattern matching** фактически выполняет сопоставление некоторого значения с некоторым шаблоном. И если сопоставление прошло успешно, то выполняются определенные действия. Язык C# позволяет выполнять различные типы сопоставлений.

Паттерн типов или **type pattern** позволяет проверить некоторое значение на соответствие некоторому типу:

|  |  |
| --- | --- |
| 1 | значение is тип переменная\_типа |

Например, у нас есть следующие классы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Employee  {  public virtual void Work() => Console.WriteLine("Employee works");  }    class Manager : Employee  {  public override void Work() => Console.WriteLine("Manager works");  public bool IsOnVacation { get; set; }  } |

Класс Employee представляет работника, а класс Manager - менеджера. Оба класса реализуют метод Work. Кроме того, класс Manager определяет свойство IsOnVacation.

С помощью паттерна типов проверим, представляет ли объект Employee класс Manager:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Employee tom = new Manager();  UseEmployee(tom); // Manager works    void UseEmployee(Employee emp)  {  if (emp is Manager manager && manager.IsOnVacation == false)  {  manager.Work();  }  else  {  Console.WriteLine("Преобразование не допустимо");  }  } |

Здесь в методе UseEmployee значение emp сопоставляется с типом Manager. То есть в данном случае в качестве шаблона выступает тип Manager. Если сопоставление прошло успешно (то есть значение emp представляет тип Manager), в переменной manager оказывается объект emp. И далее мы можем вызвать у него методы и свойства.

Также мы можем использовать **constant pattern** - сопоставление с некоторой константой:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var message = "hello";    // проверяем, соответствует ли значение message строке "hello"  if (message is "hello")  {  Console.WriteLine("hello");  } |

Подобным образом, например, можно проверить значение на null:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Employee? bob = new Employee();  Employee? tom = null;    UseEmployee(bob);  UseEmployee(tom);    void UseEmployee(Employee? emp)  {  if (emp is not null)  emp.Work();  } |

Кроме конструкции if сопоставление паттернов может применяться в конструкции switch:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Employee bob = new Employee();  Employee tom = new Manager();  UseEmployee(tom); // Manager works  UseEmployee(bob); // Object is not manager    void UseEmployee(Employee? emp)  {  switch (emp)  {  case Manager manager:  manager.Work();  break;  case null:  Console.WriteLine("Object is null");  break;  default:  Console.WriteLine("Object is not manager");  break;  }  } |

С помощью выражения **when** можно вводить дополнительные условия в конструкцию case:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Employee bob = new Manager() { IsOnVacation = true };  Employee tom = new Manager() { IsOnVacation = false };  UseEmployee(tom); // Manager works  UseEmployee(bob); // Employee does not work    void UseEmployee(Employee? emp)  {  switch (emp)  {  case Manager manager when !manager.IsOnVacation:  manager.Work();  break;  case null:  Console.WriteLine("Employee is null");  break;  default:  Console.WriteLine("Employee does not work");  break;  }  } |

В этом случае опять же преобразуем объект emp в объект типа Manager и в случае удачного преобразования смотрим на значение свойства IsOnVacation: если оно равно false, то выполняется данный блок case.

## **Паттерн свойств**

Паттерн свойств позволяет сравнивать со значениями определенных свойств объекта. Например, пусть у нас будет следующий класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class Person  {  public string Name { get; set; } = ""; // имя пользователя  public string Status { get; set; } = ""; // статус пользователя  public string Language { get; set; } = ""; // язык пользователя  } |

Например, в зависимости от языка пользователя выведем ему определенное сообщение, применив паттерн свойств:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Person tom = new Person { Language = "english", Status = "user", Name = "Tom" };  Person pierre = new Person { Language = "french", Status = "user", Name = "Pierre" };    SayHello(tom); // Hello  SayHello(pierre); // Salut    void SayHello(Person person)  {  if(person is Person { Language: "french" })  {  Console.WriteLine("Salut");  }  else  {  Console.WriteLine("Hello");  }  } |

Здесь метод SayHello в качестве параметра принимает объект Person и сопоставляет его с некоторым паттерном. В качестве паттерна выступает выражение Person { Language: "french" }. То есть параметр person должен представлять объект Person, у которого значение свойства Language равно "french".

При этом можно задействовать набор свойств. Например, добавим проверку по свойству Status:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | Person tom = new Person { Language = "english", Status = "user", Name = "Tom" };  Person pierre = new Person { Language = "french", Status = "user", Name = "Pierre" };  Person admin = new Person { Language = "english", Status = "admin", Name = "Admin" };    SayHello(admin); // Hello, admin  SayHello(tom); // Hello  SayHello(pierre); // Salut    void SayHello(Person person)  {  if(person is Person { Language: "english", Status: "admin" })  {  Console.WriteLine("Hello, admin");  }  else if (person is Person { Language: "french"})  {  Console.WriteLine("Salut");  }  else  {  Console.WriteLine("Hello");  }  } |

Теперь выражение if проверяет, соответствует ли параметр person объекту Person, у которого свойства Language и Status имеют определенные значения.

Подобным образом можно применять паттерн свойств в конструкции switch:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | string GetMessage(Person? p) => p switch  {  { Language: "english" } => "Hello!",  { Language: "german", Status: "admin" } => "Hallo, admin!",  { Language: "french" } => "Salut!",  { } => "undefined",  null => "null" // если Person p = null  }; |

Паттерны свойств предполагают использование фигурных скобок, внутри которых указываются свойства и через двоеточие их значение {свойство: значение}. И со значением свойства в фигурных скобках сравнивается свойство передаваемого объекта. При этом в фигурных скобках мы можем указать несколько свойств и их значений { Language: "german", Status: "admin" } - тогда свойства передаваемого объекта должны соответствовать всем этим значениям.

Можно оставить пустые фигурные скобки, как в последнем случае { } => "undefined!" - передаваемый объект будет соответствовать пустым фигурным скобкам, если он не соответствует всем предыдущим значениям, или например, если его свойства не указаны или имеют значение null.

То есть в данном случае, если у объекта Person p выполняется равенство Language = "english", будет возвращаться строка "Hello!".

Если у объекта Person p одновременно выполняются два равенства Language = "german" и Role="admin", будет возвращаться строка "Hallo, admin!".

Если у объекта Person p выполняется равенство Language = "french", будет возвращаться строка "Salut!".

Если объект Person будет сопоставляться с пустыми фигурными скобками {}, и будет возвращаться строка "undefined".

Последняя проверка проверяет значение на null.

Применение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Person pierre = new Person { Language = "french", Status = "user", Name = "Pierre" };  string message = GetMessage(pierre);  Console.WriteLine(message); // Salut!    Person tomas = new Person { Language = "german", Status = "admin", Name = "Tomas" };  Console.WriteLine(GetMessage(tomas)); // Hallo, admin!    Person pablo = new Person { Language = "spanish", Status = "user", Name = "Pablo" };  Console.WriteLine(GetMessage(pablo)); // undefined    Console.WriteLine(GetMessage(null)); // null |

Кроме того, мы можем определять в паттерных свойств переменные, передавать этим переменным значения объекта и использовать при возвращении значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | string GetMessage(Person? p) => p switch  {  { Language: "german", Status: "admin" } => "Hallo, admin!",  { Language: "french", Name: var name } => $"Salut, {name}!",  { Language: var lang} => $"Unknown language: {lang}",  null => "null"  }; |

Так, подвыражение Name: var name говорит, что надо передать в переменную name значение свойства Name. Затем ее можно применить при генерации выходного значения: => $"Salut, {name}!"

Применение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Person pierre = new Person { Language = "french", Status = "user", Name = "Pierre" };  string message = GetMessage(pierre);  Console.WriteLine(message); // Salut, Pierre!    Person tomas = new Person { Language = "german", Status = "admin", Name = "Tomas" };  Console.WriteLine(GetMessage(tomas)); // Hallo, admin!    Person pablo = new Person { Language = "spanish", Status = "user", Name = "Pablo" };  Console.WriteLine(GetMessage(pablo)); // Unknown language: spanish    Person? bob = null;  Console.WriteLine(GetMessage(bob)); // null |

Стоит отметить, что начиная с версии C# 10 было упрощено сопоставление со свойствами вложенных объектов. Допустим, у нас есть следующие классы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Employee  {  public string Name { get;}  public Company Company { get; set; }  public Employee(string name, Company company)  {  Name = name;  Company = company;  }    }  class Company  {  public string Title { get;}  public Company(string title) => Title = title;  } |

Класс Company определяет свойство Title, которое хранит название компании. Класс Employee определяет сотрудника компании и в свойстве Company хранит компанию. Применим паттерн свойств на основе свойств вложенного объекта Company:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | var microsoft = new Company("Microsoft");  var google = new Company("Google");  var tom = new Employee("Tom", microsoft);  var bob = new Employee("Bob", google);    PrintCompany(tom); //  PrintCompany(bob); //    void PrintCompany(Employee employee)  {  if (employee is Employee { Company:{Title: "Microsoft" } })  {  Console.WriteLine($"{employee.Name} works in Microsoft");  }  else  {  Console.WriteLine($"{employee.Name} works someware");  }  } |

В методе PrintCompany объект employee сопоставляется с паттерном Employee { Company:{Title: "Microsoft" } }. То есть сотрудник компании должен представлять объект Employee, у которого название компании равно "Microsoft"

Однако мы также можем сократить данный паттерн следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | void PrintCompany(Employee employee)  {  if (employee is Employee { Company.Title: "Microsoft" })  {  Console.WriteLine($"{employee.Name} works in Microsoft");  }  else  {  Console.WriteLine($"{employee.Name} works someware");  }  } |

## **Паттерны кортежей**

Паттерны кортежей позволяют сравнивать значения кортежей. Например, передадим в конструкцию switch кортеж с названием языка и времени дня и в зависимости от переданных данных возвратим определенное сообщение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | string GetWelcome(string lang, string daytime) => (lang, daytime) switch  {  ("english", "morning") => "Good morning",  ("english", "evening") => "Good evening",  ("german", "morning") => "Guten Morgen",  ("german", "evening") => "Guten Abend",  \_ => "Здрасьть"  }; |

Здесь в метод передаются два значения, из которых создается кортеж (можно и сразу передать в метод кортеж). Далее в конструкции switch с помощью круглых скобок определяются значения, которым должны соответствовать элементы кортежа. Например, выражение ("english", "morning") => "Good morning" будет выполняться, если одновременно lang="english" и datetime="morning".

Применение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | string message = GetWelcome("english", "evening");  Console.WriteLine(message); // Good evening    message = GetWelcome("french", "morning");  Console.WriteLine(message); // Здрасьть |

Нам не обязательно сравнивать все значения кортежа, мы можем использовать только некоторые элементы кортежа. В случае, если мы не хотим использовать элемент кортежа, то вместо него ставим прочерк:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string GetWelcome(string lang, string daytime, string status) => (lang, daytime, status) switch  {  ("english", "morning", \_) => "Good morning",  ("english", "evening", \_) => "Good evening",  ("german", "morning", \_) => "Guten Morgen",  ("german", "evening", \_) => "Guten Abend",  (\_, \_, "admin") => "Hello, Admin",  \_ => "Здрасьть"  }; |

Теперь кортеж состоит из трех элементов. Но первые четыре выражения не используют последний элемент кортежа, допустим, он не важен, поэтому вместо него ставится прочерк ("english", "morning", \_).

А в предпоследнем примере, наоборот, не важны первые два элемента, а важен третий элемент: (\_, \_, "admin") =>.

Но в любом случае нам надо указать конкретные значения или прочерки для всех элементов кортежа.

|  |  |
| --- | --- |
| 1  2  3  4  5 | string message = GetWelcome("english", "evening", "user");  Console.WriteLine(message); // Good evening    message = GetWelcome("french", "morning", "admin");  Console.WriteLine(message); // Hello, Admin |

## **Позиционный паттерн**

Позиционный паттерн применяется к типу, у которого определен метод деконструктора. Например, определим следующий класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class MessageDetails  {  public string Language { get; set; } = ""; // язык пользователя  public string DateTime { get; set; } = ""; // время суток  public string Status { get; set; } = ""; // статус пользователя    public void Deconstruct(out string lang, out string datetime, out string status)  {  lang = Language;  datetime = DateTime;  status = Status;  }  } |

Теперь используем позиционный паттерн и в зависимости от значений объекта MessageDetails возвратим определенное сообщение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string GetWelcome(MessageDetails details) => details switch  {  ("english", "morning", \_) => "Good morning",  ("english", "evening", \_) => "Good evening",  ("german", "morning", \_) => "Guten Morgen",  ("german", "evening", \_) => "Guten Abend",  (\_, \_, "admin") => "Hello, Admin",  \_ => "Здрасьть"  }; |

Фактически этот паттерн похож на пример с кортежами выше, только теперь вместо кортежа в конструкцию switch передается объект MessageDetails. Через метод деконструктора мы можем получить набор выходных параметров в виде кортежа и опять же по позиции сравнивать их с некоторыми значениями в конструкции switch.

Применение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | MessageDetails details1 = new MessageDetails { Language = "english", DateTime = "evening", Status = "user" };  string message = GetWelcome(details1);  Console.WriteLine(message); // Good evening    MessageDetails details2 = new MessageDetails { Language = "french", DateTime = "morning", Status = "admin" };  message = GetWelcome(details2);  Console.WriteLine(message); // Hello, Admin |

Также мы можем взять значения объекта MessageDetails и использовать их при создании результата метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | string GetWelcome(MessageDetails details) => details switch  {  ("english", "morning", \_) => "Good morning",  ("english", "evening", \_) => "Good evening",  ("german", "morning", \_) => "Guten Morgen",  ("german", "evening", \_) => "Guten Abend",  (\_, \_, "admin") => "Hello, Admin",  (var lang, var datetime, var status) => $"{lang} not found, {datetime} unknown, {status} undefined",  \_ => "Здрасьть"  }; |

В предпоследней инструкции в конструкции switch получаем по позиции значения из MessageDetails в переменные lang, datetime и status и используем их для создания сообщения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | MessageDetails details1 = new MessageDetails  {  Language = "chinese",  DateTime = "night",  Status = "moderator"  };  string message = GetWelcome(details1);  Console.WriteLine(message); // chinese not found, night unknown, moderator undefined |

## **Реляционный и логический паттерны**

В C# 9.0 в язык были добавлены дополнительные паттерны - реляционный (relational pattern) и логический (logical pattern) паттерны.

Реляционный паттерн позволяет сравнить передаваемое в конструкцию значение с некоторыми значениями с помощью операций сравнения. Например, в зависимости от суммы вклада проценты по этому вкладу могут отличаться. Рассчитаем сумму процентов в зависимости от суммы вклада с использованием реляционного паттерна:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | decimal Calculate(decimal sum)  {  return sum switch {  <= 0 => 0, // если sum меньше или равно 0, возвращаем 0  < 50000 => sum \* 0.05m, // если sum меньше 50000, возвращаем sum \* 0.05m  < 100000 => sum \* 0.1m, // если sum меньше 100000, возвращаем sum \* 0.1m  \_ => sum \* 0.2m // в остальных случаях возвращаем sum \* 0.2m  };  } |

Применение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Console.WriteLine(Calculate(-200)); // 0  Console.WriteLine(Calculate(0)); // 0  Console.WriteLine(Calculate(10000)); // 500  Console.WriteLine(Calculate(60000)); // 6000  Console.WriteLine(Calculate(200000)); // 40000 |

Логический паттерн позволяет использовать логические операторы **and** (логическое умножение или операция логического И) и **or** (логическое сложение или операция логического ИЛИ) для объединения операций сравнения. Например, передадим в метод возраст и возвратим соответствующее сообщение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string CheckAge(int age)  {  return age switch  {  <1 or >110 => "Недействительный возраст", // если age больше 110 и меньше 1  >=1 and <18 => "Доступ запрещен", // если age равно или больше 1 и меньше 18  \_ => "Доступ разрешен" // в остальных случаях  };  } |

Применение:

|  |  |
| --- | --- |
| 1  2  3  4 | Console.WriteLine(CheckAge(200)); // Недействительный возраст  Console.WriteLine(CheckAge(0)); // Недействительный возраст  Console.WriteLine(CheckAge(17)); // Доступ запрещен  Console.WriteLine(CheckAge(18)); // Доступ разрешен |

Еще один логический оператор - **not** используется для отрицания и возвращает true, если его аргумент равен false:

|  |  |
| --- | --- |
| 1  2  3  4  5 | string CheckAge(int age) => age switch  {  not 33 => "Обычный возраст", // если age НЕ равен 33  \_ => "Вам 33 года" // в остальных случаях, то есть если age = 33  }; |

## **Паттерны списков**

Паттерны списков (list pattern) позволяют сопоставлять выражения со списками и массивами. Данный паттерн пока доступен начиная с версии C# 11.

Полное совпадение с массивом/списком:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Console.WriteLine(GetNumber(new[] { 1, 2, 3, 4, 5 })); // 1  Console.WriteLine(GetNumber(new[] { 1, 2})); // 3  Console.WriteLine(GetNumber(new int[] {})); // 4  Console.WriteLine(GetNumber(new[] { 1, 2, 5 })); // 5    int GetNumber(int[] values)=> values switch  {  [1, 2, 3, 4, 5] => 1,  [1, 2, 3] => 2,  [1, 2] => 3,  [] => 4,  \_ => 5  }; |

Аналогично вместо массивов можно применять списки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | List<int> numbers = new List<int>{1, 2, 3};    Console.WriteLine(GetNumber(numbers)); // 2    int GetNumber(List<int> values) => values switch  {  [1, 2, 3, 4, 5] => 1,  [1, 2, 3] => 2,  [1, 2] => 3,  [] => 4,  \_ => 5  }; |

Аналогичным образом паттерны списков можно использовать в конструкции **if**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[] numbers = { 1, 2, 3, 4, 5 };  if(numbers is [1, 2, 3, 4, 5])  {  Console.WriteLine("[1, 2, 3, 4, 5]");  } |

#### **Подстановка \_**

С помощью паттерна **\_** можно обозначить одиночный элемент, который имеет любое значение. Например, паттерн **[2, \_, 5]** соответствует любому массиву из трех элементов, в котором между 2 и 5 располагается произвольное значение. А массив **[\_, \_]** соответствует любому массиву из двух произвольных элементов

Несколько примеров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Console.WriteLine(GetNumber(new[] { 2, 3, 5})); // 1  Console.WriteLine(GetNumber(new[] { 2, 4, 6 })); // 2  Console.WriteLine(GetNumber(new[] { 1, 2, 5 })); // 3  Console.WriteLine(GetNumber(new[] { 1, 2, 3})); // 4  Console.WriteLine(GetNumber(new int[] { })); // 5    int GetNumber(int[] values) => values switch  {  [2, \_, 5] => 1,  [2, \_, \_] => 2,  [\_, \_, 5] => 3,  [\_, \_, \_] => 4,  \_ => 5  }; |

#### **slice-паттерн**

Для передачи произвольного количества элементов массива/списка применяется slice-паттерн **..**. Например, паттерн **[1, 2, .., 5]** соответствует массиву, в который начинается на 1, за которым идет 2. А последний элемент в массиве - 5. При этом между 2 и 5 может располагаться произвольное количество произвольных целых чисел. То есть паттерн **[1, 2, .., 5]** будет соответствовать таким массивам как

|  |  |
| --- | --- |
| 1  2  3 | int[] arr1 = { 1, 2, 3, 4, 5 };  int[] arr2 = { 1, 2, 5 };  int[] arr3 = { 1, 2, 66, 77, 88, 5 }; |

С помощью паттерна **..** можно задавать произвольное количество элементов как в начале, так и в конце массива/списка. Например, паттерн **[2,..]** представляет массив, который начинается на 2. А паттерн **[.., 5]** представляет массив, который заканчивается элементом 5. Паттерн **[..]** представляет массив, который содержит произвольное количество элементов. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Console.WriteLine(GetNumber(new[] { 2, 5 })); // 1  Console.WriteLine(GetNumber(new[] { 2, 3, 4, 5 })); // 1    Console.WriteLine(GetNumber(new[] { 2})); // 2  Console.WriteLine(GetNumber(new[] { 2, 3, 4})); // 2    Console.WriteLine(GetNumber(new[] {3, 4, 5})); // 3  Console.WriteLine(GetNumber(new[] { 5 })); // 3    Console.WriteLine(GetNumber(new int[] { })); // 4  Console.WriteLine(GetNumber(new[] { 1 })); // 4  Console.WriteLine(GetNumber(new[] { 1, 2, 3 })); // 4    int GetNumber(int[] values)=> values switch  {  [2, .., 5] => 1, // если первый элемент - 2, а последний - 5  [2,..] => 2, // если первый элемент - 2  [.., 5] => 3, // если последний элемент - 5  [..] => 4 // произвольное количество элементов  }; |

slice-паттерн можно сочетать с символов подстановки \_, например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int GetNumber(int[] values) => values switch  {  [\_, .., \_] => 1,  [..] => 2  }; |

В данном случае паттерн **[\_, .., \_]** предполагает массив, который состоит как минимум из двух произвольных элементов, и между первым и последним элементром может находиться произвольное количество других элементов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Console.WriteLine(GetNumber(new[] { 1, 2, 3, 4})); // 1  Console.WriteLine(GetNumber(new[] { 1, 2, 3})); // 1  Console.WriteLine(GetNumber(new[] { 1, 2})); // 1  Console.WriteLine(GetNumber(new[] { 1 })); // 2  Console.WriteLine(GetNumber(new int[] { })); // 2    int GetNumber(int[] values) => values switch  {  [\_, .., \_] => 1,  [..] => 2  }; |

### **Получение элементов в переменные**

Отдельные значения массива/списка можно получить в переменные, например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[] numbers = { 2, 3, 5 };  if(numbers is [var first, var second, .., var last])  {  Console.WriteLine($"first: {first}, second: {second} last: {last}");  } |

В данном случае получаем первый элемент массива в переменную first, второй элемент - в переменную second, а последний элемент - в переменную last.

Пример с различными массивами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Console.WriteLine(GetData(new[] { 1, 2, 3 })); // First: 1 Second: 2 Last: 3  Console.WriteLine(GetData(new[] { 2, 4, 6, 8 })); // First: 2 Second: 4 Last: 8  Console.WriteLine(GetData(new[] { 1, 2 })); // Array has less than 3 elements    string GetData(int[] values) => values switch  {  [var first, var second, .., var last] => $"First: {first} Second: {second} Last: {last}",  [..] => "Array has less than 3 elements"  }; |

В данном случае получаем первый элемент массива в переменную first, второй элемент - в переменную second, а последний элемент - в переменную last.

При этом значения, которые проектируются на паттерн **..**, также можно получить в переменную. Например, в паттерне **[2, .. var middle, 5]** элементы, которые проектируются на **..**, можно передаются в переменную middle. Несколько примеров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Console.WriteLine(GetSlice(new[] { 2, 3, 4, 5})); // Middle: 3, 4  Console.WriteLine(GetSlice(new[] { 2, 4, 6, 8 })); // End: 4, 6, 8  Console.WriteLine(GetSlice(new[] { 1, 2, 3, 5 })); // Start: 1, 2, 3  Console.WriteLine(GetSlice(new[] { 1, 2, 3, 4 })); // All: 1, 2, 3, 4  Console.WriteLine(GetSlice(new int[] { })); // All:    string GetSlice(int[] values) => values switch  {  [2, .. var middle, 5] => $"Middle: {string.Join(", ", middle)}",  [2, .. var end] => $"End: {string.Join(", ", end)}",  [.. var start, 5] => $"Start: {string.Join(", ", start)}",  [.. var all] => $"All: {string.Join(", ", all)}"  }; |

### **Свойства коллекций**

Стоит отметить, что, поскольку массивы и списки - обычные классы C#, которые имеют свойства, то для них мы также можем применять паттерн свойств. Объединение паттерна свойств и паттерна списков позволяет упростить решение некоторых задач. Например, у нас есть задача: если массив имеет три элемента, то разложить его на три переменных:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[] numbers = { 2, 3, 5 };  if(numbers is { Length:3} and [var first, var second, var third])  {  Console.WriteLine($"first: {first}, second: {second} third: {third}");  } |

# **Коллекции**

## **Список List<T>**

Хотя в языке C# есть массивы, которые хранят в себе наборы однотипных объектов, но работать с ними не всегда удобно. Например, массив хранит фиксированное количество объектов, однако что если мы заранее не знаем, сколько нам потребуется объектов. И в этом случае намного удобнее применять коллекции. Еще один плюс коллекций состоит в том, что некоторые из них реализует стандартные структуры данных, например, стек, очередь, словарь, которые могут пригодиться для решения различных специальных задач. Большая часть классов коллекций содержится в пространстве имен **System.Collections.Generic**.

Класс List<T> из пространства имен System.Collections.Generic представляет простейший список однотипных объектов. Класс List типизируется типом, объекты которого будут хранится в списке.

Мы можем создать пустой список:

|  |  |
| --- | --- |
| 1 | List<string> people = new List<string>(); |

В данном случае объект List типизируется типом **string**. А это значит, что хранить в этом списке мы можем только строки.

Можно сразу при создании списка инициализировать его начальными значениями. В этом случае элементы списка помещаются после вызова конструктора в фигурных скобках

|  |  |
| --- | --- |
| 1 | List<string> people = new List<string>() { "Tom", "Bob", "Sam" }; |

В данном случае в список помещаются три строки

Также можно при создании списка инициализировать его элементами из другой коллекции, например, другого списка:

|  |  |
| --- | --- |
| 1  2 | var people = new List<string>() { "Tom", "Bob", "Sam" };  var employees = new List<string>(people); |

Можно совместить оба способа:

|  |  |
| --- | --- |
| 1  2 | var people = new List<string>() { "Tom", "Bob", "Sam" };  var employees = new List<string>(people){"Mike"}; |

В данном случае в списке employees будет четыре элемента ({ "Tom", "Bob", "Sam", "Mike" }) - три добавляются из списка people и один элемент задается при инициализации.

Начиная с версии **C# 12** для определения списков можно использовать выражения коллекций, которые предполагают заключение элементов коллекции в квадратные скобки:

|  |  |
| --- | --- |
| 1  2 | List<string> people = ["Tom", "Bob", "Sam"];  List<string> employees = [];// пустой список |

Подобным образом можно работать со списками других типов, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | List<Person> people = new List<Person>()  {  new Person("Tom"),  new Person("Bob"),  new Person("Sam")  };    class Person  {  public string Name { get;}  public Person(string name) => Name = name;  } |

### **Установка начальной емкости списка**

Еще один конструктор класса List принимает в качестве параметра начальную емкость списка:

|  |  |
| --- | --- |
| 1 | List<string> people = new List<string>(16); |

Указание начальной емкости списка позволяет в будущем увеличить производительность и уменьшить издержки на выделение памяти при добавлении элементов. Поскольку динамическое добавление в список может приводить на низком уровне к дополнительному выделению памяти, что снижает производительность. Если же мы знаем, что список не будет превышать некоторый размер, то мы можем передать этот размер в качестве емкости списка и избежать дополнительных выделений памяти.

Также начальную емкость можно установить с помощью свойства Capacity, которое имеется у класса List.

### **Обращение к элементам списка**

Как и массивы, списки поддерживают индексы, с помощью которых можно обратиться к определенным элементам:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var people = new List<string>() { "Tom", "Bob", "Sam" };    string firstPerson = people[0]; // получаем первый элемент  Console.WriteLine(firstPerson); // Tom  people[0] = "Mike"; // изменяем первый элемент  Console.WriteLine(people[0]); // Mike |

### **Длина списка**

С помощью свойства **Count** можно получить длину списка:

|  |  |
| --- | --- |
| 1  2 | var people = new List<string>() { "Tom", "Bob", "Sam" };  Console.WriteLine(people.Count); // 3 |

### **Перебор списка**

C# позволяет осуществить перебор списка с помощью стандартного цикла **foreach**:/p>

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var people = new List<string>() { "Tom", "Bob", "Sam" };    foreach (var person in people)  {  Console.WriteLine(person);  }  // Вывод программы:  // Tom  // Bob  // Sam |

Также можно использовать другие типы циклов и в комбинации с индексами перебирать списки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var people = new List<string>() { "Tom", "Bob", "Sam" };    for (int i = 0; i < people.Count; i++)  {  Console.WriteLine(people[i]);  } |

### **Методы списка**

Среди его методов можно выделить следующие:

* **void Add(T item)**: добавление нового элемента в список
* **void AddRange(IEnumerable<T> collection)**: добавление в список коллекции или массива
* **int BinarySearch(T item)**: бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован.
* **void CopyTo(T[] array)**: копирует список в массив array
* **void CopyTo(int index, T[] array, int arrayIndex, int count)**: копирует из списка начиная с индекса index элементы, количество которых равно count, и вставляет их в массив array начиная с индекса arrayIndex
* **bool Contains(T item)**: возвращает true, если элемент item есть в списке
* **void Clear()**: удаляет из списка все элементы
* **bool Exists(Predicate<T> match)**: возвращает true, если в списке есть элемент, который соответствует делегату match
* **T? Find(Predicate<T> match)**: возвращает первый элемент, который соответствует делегату match. Если элемент не найден, возвращается null
* **T? FindLast(Predicate<T> match)**: возвращает последний элемент, который соответствует делегату match. Если элемент не найден, возвращается null
* **List<T> FindAll(Predicate<T> match)**: возвращает список элементов, которые соответствуют делегату match
* **int IndexOf(T item)**: возвращает индекс первого вхождения элемента в списке
* **int LastIndexOf(T item)**: возвращает индекс последнего вхождения элемента в списке
* **List<T> GetRange(int index, int count)**: возвращает список элементов, количество которых равно count, начиная с индекса index.
* **void Insert(int index, T item)**: вставляет элемент item в список по индексу index. Если такого индекса в списке нет, то генерируется исключение
* **void InsertRange(int index, collection)**: вставляет коллекцию элементов collection в текущий список начиная с индекса index. Если такого индекса в списке нет, то генерируется исключение
* **bool Remove(T item)**: удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true. Если в списке несколько одинаковых элементов, то удаляется только первый из них
* **void RemoveAt(int index)**: удаление элемента по указанному индексу index. Если такого индекса в списке нет, то генерируется исключение
* **void RemoveRange(int index, int count)**: параметр index задает индекс, с которого надо удалить элементы, а параметр count задает количество удаляемых элементов.
* **int RemoveAll((Predicate<T> match))**: удаляет все элементы, которые соответствуют делегату match. Возвращает количество удаленных элементов
* **void Reverse()**: изменяет порядок элементов
* **void Reverse(int index, int count)**: изменяет порядок на обратный для элементов, количество которых равно count, начиная с индекса index
* **void Sort()**: сортировка списка
* **void Sort(IComparer<T>? comparer)**: сортировка списка с помощью объекта comparer, который передается в качестве параметра

### **Добавление в список**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | List<string> people = new List<string> () { "Tom" };    people.Add("Bob"); // добавление элемента  // people = { "Tom", "Bob" };    people.AddRange(new[] { "Sam", "Alice" }); // добавляем массив  // people = { "Tom", "Bob", "Sam", "Alice" };  // также можно было бы добавить другой список  // people.AddRange(new List<string>(){ "Sam", "Alice" });    people.Insert(0, "Eugene"); // вставляем на первое место  // people = { "Eugene", "Tom", "Bob", "Sam", "Alice" };    people.InsertRange(1, new string[] {"Mike", "Kate"}); // вставляем массив с индекса 1  // people = { "Eugene", "Mike", "Kate", "Tom", "Bob", "Sam", "Alice" };    // также можно было бы добавить другой список  // people.InsertRange(1, new List<string>(){ "Mike", "Kate" }); |

### **Удаление из списка**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | var people = new List<string> () { "Eugene", "Mike", "Kate", "Tom", "Bob", "Sam", "Tom", "Alice" };    people.RemoveAt(1); // удаляем второй элемент  // people = { "Eugene", "Kate", "Tom", "Bob", "Sam", "Tom", "Alice" };    people.Remove("Tom"); // удаляем элемент "Tom"  // people = { "Eugene", "Kate", "Bob", "Sam", "Tom", "Alice" };    // удаляем из списка все элементы, длина строки которых равна 3  people.RemoveAll(person => person.Length == 3);  // people = { "Eugene", "Kate", "Alice" };    // удаляем из списка 2 элемента начиная с индекса 1  people.RemoveRange(1, 2);  // people = { "Eugene"};    // полностью очищаем список  people.Clear();  // people = { }; |

### **Поиск и проверка элемента**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | var people = new List<string> () { "Eugene", "Mike", "Kate", "Tom", "Bob", "Sam" };    var containsBob = people.Contains("Bob"); // true  var containsBill = people.Contains("Bill"); // false    // проверяем, есть ли в списке строки с длиной 3 символа  var existsLength3 = people.Exists(p => p.Length == 3); // true    // проверяем, есть ли в списке строки с длиной 7 символов  var existsLength7 = people.Exists(p => p.Length == 7); // false    // получаем первый элемент с длиной в 3 символа  var firstWithLength3 = people.Find(p => p.Length == 3); // Tom    // получаем последний элемент с длиной в 3 символа  var lastWithLength3 = people.FindLast(p => p.Length == 3); // Sam    // получаем все элементы с длиной в 3 символа в виде списка  List<string> peopleWithLength3 = people.FindAll(p => p.Length == 3);  // peopleWithLength3 { "Tom", "Bob", "Sam"} |

### **Получение диапазона и копирование в массив**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | List<string> people = new List<string>() {"Eugene", "Tom", "Mike", "Sam", "Bob" };    // получаем диапазон со второго по четвертый элемент  var range = people.GetRange(1, 3);  // range = { "Tom", "Mike", "Sam"};    // копируем в массив первые три элемента  string[] partOfPeople = new string[3];  people.CopyTo(0, partOfPeople, 0, 3);  // partOfPeople = { "Eugene", "Tom", "Mike"}; |

### **Расположение элементов в обратном порядке**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var people = new List<string> () { "Eugene", "Tom", "Mike", "Sam", "Bob" };    // переворачиваем весь список  people.Reverse();  // people = { "Bob","Sam", "Mike", "Tom", "Eugene"};    var people2 = new List<string>() { "Eugene", "Tom", "Mike", "Sam", "Bob" };  // переворачиваем часть только 3 элемента с индекса 1  people2.Reverse(1, 3);  // people2 = { "Eugene","Sam", "Mike", "Tom", "Bob" }; |

## **Двухсвязный список LinkedList<T>**

Класс **LinkedList<T>** представляет двухсвязный список, в котором каждый элемент хранит ссылку одновременно на следующий и на предыдущий элемент.

### **Создание связанного списка**

Для создания связного списка можно принименять один из его конструктора. Например, создадим пустой связный список:

|  |  |
| --- | --- |
| 1 | LinkedList<string> people = new LinkedList<string>(); |

В данном случае связанный список people предназначен для хранения строк.

Также можно в конструктор передать коллекцию элементов, например, список List, по которому будет создан связный список:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var employees = new List<string> { "Tom", "Sam", "Bob" };    LinkedList<string> people = new LinkedList<string>(employees);  foreach (string person in people)  {  Console.WriteLine(person);  } |

### **LinkedListNode**

Если в простом списке List<T> каждый элемент представляет объект типа T, то в LinkedList<T> каждый узел представляет объект класса **LinkedListNode<T>**. А добавляемые в связанный список элементы T фактически обертываются в объект LinkedListNode.

Класс LinkedListNode имеет следующие свойства:

* **Value**: возвращает или устанавливает само значение узла, представленное типом T
* **Next**: возвращает ссылку на следующий элемент типа LinkedListNode<T> в списке. Если следующий элемент отсутствует, то имеет значение null
* **Previous**: возвращает ссылку предыдущий элемент типа LinkedListNode<T> в списке. Если предыдущий элемент отсутствует, то имеет значение null

### **Свойства LinkedList**

Класс LinkedList определяет следующие свойства:

* **Count**: количество элементов в связанном списке
* **First**: первый узел в списке в виде объекта LinkedListNode<T>
* **Last**: последний узел в списке в виде объекта LinkedListNode<T>

Используем эти свойства:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var employees = new List<string> { "Tom", "Sam", "Bob" };    LinkedList<string> people = new LinkedList<string>(employees);  Console.WriteLine(people.Count); // 3  Console.WriteLine(people.First?.Value); // Tom  Console.WriteLine(people.Last?.Value); // Bob |

Используя свойства LinkedList и LinkedListNode, можно пройтись по всем элементам списка в прямом или обратном порядке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | LinkedList<string> people = new LinkedList<string>(new[] { "Tom", "Sam", "Bob" });    // от начала до конца списка  var currentNode = people.First;  while(currentNode != null)  {  Console.WriteLine(currentNode.Value);  currentNode = currentNode.Next;  }    // с конца до начала списка  currentNode = people.Last;  while (currentNode != null)  {  Console.WriteLine(currentNode.Value);  currentNode = currentNode.Previous;  } |

### **Методы LinkedList**

Используя методы класса LinkedList<T>, можно обращаться к различным элементам, как в конце, так и в начале списка:

* **AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode)**: вставляет узел newNode в список после узла node.
* **AddAfter(LinkedListNode<T> node, T value)**: вставляет в список новый узел со значением value после узла node.
* **AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode)**: вставляет в список узел newNode перед узлом node.
* **AddBefore(LinkedListNode<T> node, T value)**: вставляет в список новый узел со значением value перед узлом node.
* **AddFirst(LinkedListNode<T> node)**: вставляет новый узел в начало списка
* **AddFirst(T value)**: вставляет новый узел со значением value в начало списка
* **AddLast(LinkedListNode<T> node)**: вставляет новый узел в конец списка
* **AddLast(T value)**: вставляет новый узел со значением value в конец списка
* **RemoveFirst()**: удаляет первый узел из списка. После этого новым первым узлом становится узел, следующий за удаленным
* **RemoveLast()**: удаляет последний узел из списка

Применим некоторые из этих методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var people = new LinkedList<string>();  people.AddLast("Tom"); // вставляем узел со значением Tom на последнее место  //так как в списке нет узлов, то последнее будет также и первым  people.AddFirst("Bob"); // вставляем узел со значением Bob на первое место    // вставляем после первого узла новый узел со значением Mike  if (people.First != null) people.AddAfter(people.First, "Mike");    // теперь у нас список имеет следующую последовательность: Bob Mike Tom  foreach (var person in people) Console.WriteLine(person); |

Подобным образом можно создавать связанные списки и других типов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | var company = new LinkedList<Person>();    company.AddLast(new Person("Tom"));  company.AddLast(new Person("Sam"));  company.AddFirst(new Person("Bill"));    foreach (var person in company) Console.WriteLine(person.Name);    class Person  {  public string Name { get; }  public Person(string name) => Name = name;  } |

## **Очередь Queue**

Класс **Queue<T>** представляет обычную очередь, которая работает по алгоритму FIFO ("первый вошел - первый вышел").

### **Создание очереди**

Для создания очереди можно использовать один из трех ее конструкторов. Прежде всего можно создать пустую очередь:

|  |  |
| --- | --- |
| 1 | Queue<string> people = new Queue<string>(); |

При создании пустой очереди можно указать емкость очереди:

|  |  |
| --- | --- |
| 1 | Queue<string> people = new Queue<string>(16); |

Также можно инициализировать очередь элементами из другой коллекции или массивом:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var employees = new List<string> { "Tom", "Sam", "Bob" };  Queue<string> people = new Queue<string>(employees);  foreach (var person in people) Console.WriteLine(person);    Console.WriteLine(people.Count); // 3 |

Для перебора очереди можно использовать стандартный цикл **foreach**.

Для получения количества элементов в очереди в классе определено свойство **Count**.

### **Методы Queue**

У класса Queue<T> можно отметить следующие методы:

* **void Clear()**: очищает очередь
* **bool Contains(T item)**: возвращает true, если элемент item имеется в очереди
* **T Dequeue()**: извлекает и возвращает первый элемент очереди
* **void Enqueue(T item)**: добавляет элемент в конец очереди
* **T Peek()**: просто возвращает первый элемент из начала очереди без его удаления

Посмотрим применение очереди на практике:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | var people = new Queue<string>();    // добавляем элементы  people.Enqueue("Tom"); // people = { Tom }  people.Enqueue("Bob"); // people = { Tom, Bob }  people.Enqueue("Sam"); // people = { Tom, Bob, Sam }    // получаем элемент из самого начала очереди  var firstPerson = people.Peek();  Console.WriteLine(firstPerson); // Tom    // удаляем элементы  var person1 = people.Dequeue(); // people = { Bob, Sam }  Console.WriteLine(person1); // Tom  var person2 = people.Dequeue(); // people = { Sam }  Console.WriteLine(person2); // Bob  var person3 = people.Dequeue(); // people = { }  Console.WriteLine(person3); // Sam |

Стоит отметить, что если с помощью методов Peek или Dequeue мы попытаемся получить первый элемент очереди, которая пуста, то программа выдаст исключение. Соответственно перед получением элемента мы можем проверять количество элементов в очереди:

|  |  |
| --- | --- |
| 1  2  3  4  5 | if(people.Count > 0)  {  var person = people.Peek();  people.Dequeue();  } |

Либо можно использовать пару методов:

* **bool TryDequeue(out T result)**: передает в переменную result первый элемент очереди с его удалением из очереди, возвращает true, если очередь не пуста и элемент успешно получен.
* **bool TryPeek(out T result)**: передает в переменную result первый элемент очереди без его извлечения из очереди, возвращает true, если очередь не пуста и элемент успешно получен.

Применение методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var people = new Queue<string>();    // добавляем элементы  people.Enqueue("Tom"); // people = { Tom }    // удаляем элементы  var success1 = people.TryDequeue(out var person1); // success1 = true  if (success1) Console.WriteLine(person1); // Tom    var success2 = people.TryPeek(out var person2); // success2 = false  if (success2) Console.WriteLine(person2); |

Очереди - довольно часто встречаемая стуктура в реальной жизни. Например, очередь пациентов на прием к врачу. Реализуем данную ситуацию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | var patients = new Queue<Person>();  patients.Enqueue(new Person("Tom"));  patients.Enqueue(new Person("Bob"));  patients.Enqueue(new Person("Sam"));    var practitioner = new Doctor();  practitioner.TakePatients(patients);    class Person  {  public string Name { get; }  public Person(string name) => Name = name;  }    class Doctor  {  public void TakePatients(Queue<Person> patients)  {  while(patients.Count > 0)  {  var patient = patients.Dequeue();  Console.WriteLine($"Осмотр пациента {patient.Name}");  }  Console.WriteLine("Доктор закончил осматривать пациентов");  }  } |

Здесь класс врача - класс Doctor в методе TakePatients принимает очередь пациентов в виде объектов Person. И пока в очереди есть объекты извлекает по одному объекту. Консольный вывод:

|  |  |
| --- | --- |
| 1  2  3  4 | Осмотр пациента Tom  Осмотр пациента Bob  Осмотр пациента Sam  Доктор закончил осматривать пациентов |

## **Коллекция Stack<T>**

Класс Stack<T> представляет коллекцию, которая использует алгоритм LIFO ("последний вошел - первый вышел"). При такой организации каждый следующий добавленный элемент помещается поверх предыдущего. Извлечение из коллекции происходит в обратном порядке - извлекается тот элемент, который находится выше всех в стеке.

Стек - довольно часто встречаемая структура данных в реальной жизни. Банальные примеры стеков - стопка книг или тарелок, где каждую новую книгу или тарелку помещают поверх предыдущей. А извлекают из этой стопки книги/тарелки в обратном порядке - сначала самую верхнюю и так далее. Другой пример - одежда: допустим, человек выходит на улицу в зимнюю погоду и для этого сначала одевает майку, потом рубашку, затем свитер, и в конце куртку. Когда человек снимает с себя одежду - он делает это в обратном порядке: сначала снимает куртку, потом свитер и так далее.

### **Создание стека**

Для создания стека можно использовать один из трех конструкторов. Прежде всего можно создать пустой стек:

|  |  |
| --- | --- |
| 1 | Stack<string> people = new Stack<string>(); |

При создании пустого стека можно указать емкость стека:

|  |  |
| --- | --- |
| 1 | Stack<string> people = new Stack<string>(16); |

Также можно инициализировать стек элементами из другой коллекции или массивом:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var employees = new List<string> { "Tom", "Sam", "Bob" };  Stack<string> people = new Stack<string>(employees);  foreach (var person in people) Console.WriteLine(person);    Console.WriteLine(people.Count); // 3 |

Для перебора стека можно использовать стандартный цикл **foreach**. Причем в цикле в соответствии с аалгоритмом стека LIFO данные извлекаются в порядке, обратном их добавлению. Консольный вывод в данном случае:

Bob

Sam

Tom

3

Для получения количества элементов стека применяется свойство **Count**.

### **Методы Stack**

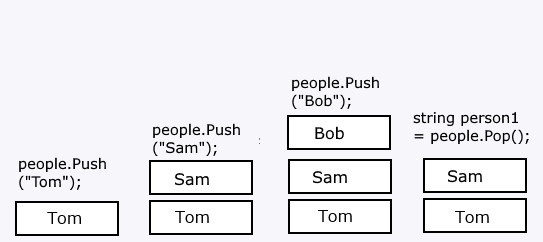
В классе Stack можно выделить следующие методы:

* **Clear**: очищает стек
* **Contains**: проверяет наличие в стеке элемента и возвращает true при его наличии
* **Push**: добавляет элемент в стек в верхушку стека
* **Pop**: извлекает и возвращает первый элемент из стека
* **Peek**: просто возвращает первый элемент из стека без его удаления

Посмотрим на примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | var people = new Stack<string>();  people.Push("Tom");  // people = { Tom }  people.Push("Sam");  // people = { Sam, Tom }  people.Push("Bob");  // people = { Bob, Sam, Tom }    // получаем первый элемент стека без его удаления  string headPerson = people.Peek();  Console.WriteLine(headPerson); // Bob    string person1 = people.Pop();  // people = { Sam, Tom }  Console.WriteLine(person1); // Bob    string person2 = people.Pop();  // people = { Tom }  Console.WriteLine(person2); // Sam    string person3 = people.Pop();  // people = { }  Console.WriteLine(person3); // Tom |

Работу стека можно представить следующей иллюстрацией:



Стоит отметить, что если с помощью методов Peek или Pop мы попытаемся получить первый элемент стека, который пуст, то программа выдаст исключение. Соответственно перед получением элемента мы можем проверять количество элементов в стеке:

|  |  |
| --- | --- |
| 1  2  3  4  5 | if(people.Count > 0)  {  var person = people.Peek();  people.Pop();  } |

Либо можно использовать пару методов:

* **bool TryPop(out T result)**: удаляет из стека первый элемент и передает его в переменную result, возвращает true, если очередь не пуста и элемент успешно получен.
* **bool TryPeek(out T result)**: передает в переменную result первый элемент стека без его извлечения, возвращает true, если элемент успешно получен.

Применение методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var people = new Stack<string>();  people.Push("Tom");  // people = { Tom }    // удаляем элементы  var success1 = people.TryPop(out var person1); // success1 = true  if (success1) Console.WriteLine(person1); // Tom    var success2 = people.TryPeek(out var person2); // success2 = false  if (success2) Console.WriteLine(person2); |

## **Коллекция Dictionary<K, V>**

Еще один распространенный тип коллекции представляют словари. Словарь хранит объекты, которые представляют пару ключ-значение. Класс словаря **Dictionary<K, V>** типизируется двумя типами: параметр **K** представляет тип ключей, а параметр **V** предоставляет тип значений.

### **Создания и инициализация словаря**

Класс Dictionary предоставляет ряд конструкторов для создания словаря. Например, мы можем создать пустой словарь:

|  |  |
| --- | --- |
| 1 | Dictionary<int, string> people = new Dictionary<int, string>(); |

Здесь словарь people в качестве ключей принимает значения типа int, а в качестве значений - строки.

При определении словаря его сразу же можно инициализировать значениями:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var people = new Dictionary<int, string>()  {  { 5, "Tom"},  { 3, "Sam"},  { 11, "Bob"}  }; |

При инициализации применяется инициализитор - в фигурных скобках после вызова конструктора объекту передаются начальные данные. В случае со словаем мы можем передать в инициализаторе набор элементов, где каждый элемент заключается в фигурные скобки, например:

|  |  |
| --- | --- |
| 1 | { 5, "Tom"} |

Каждый элемент представляет два значения: первое значение представляет ключ, а второе значение - собственно значение элемента. Поскольку при объявлении словаря people для ключей указан тип int, а для значений - тип string, то в элементе словаря сначала указывается число int, а затем строка. То есть в случае выше элемент имеет ключ 5, а значение - "Tom". Затем по ключу элемента мы сможем получить его значение.

Также мы можем применять другой способ инициализации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var people = new Dictionary<int, string>()  {  [5] = "Tom",  [6] = "Sam",  [7] = "Bob"  }; |

При таком способе инициализации в квадратных скобках указывается ключ и ему присваивается значение элемента. Но в целом этот способ инициализации будет равноценен предыдущему.

### **KeyValuePair**

Стоит отметить, что каждый элемент в словаре представляет структуру **KeyValuePair<TKey, TValue>**, где параметр **TKey** представляет тип ключа, а параметр **TValue** - тип значений элементов. Эта структура предоставляет свойства Key и Value, с помощью которых можно получить соответственно ключ и значение элемента в словаре. И одна из версий конструктора Dictionary позволяет инициализировать словарь коллекцией объектов KeyValuePair:

|  |  |
| --- | --- |
| 1  2  3 | var mike = new KeyValuePair<int, string>(56, "Mike");  var employees = new List<KeyValuePair<int, string>>() { mike};  var people = new Dictionary<int, string>(employees); |

Конструктор типа KeyValuePair принимает два параметра - ключ элемента и его значения. То есть в данном случае создается один такой элемент - mike с ключом 56 и значением "Mike". И этот элемент добавляется в список employees, которым затем инициализируется словарь.

Можно совместить оба способа инициализации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var mike = new KeyValuePair<int, string>(56, "Mike");  var employees = new List<KeyValuePair<int, string>>() { mike };  var people = new Dictionary<int, string>(employees)  {  [5] = "Tom",  [6] = "Sam",  [7] = "Bob",  }; |

В данном случае в словаре people будет четыре элемента.

### **Перебор словаря**

Для перебора словаря можно применять цикл **foreach**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var people = new Dictionary<int, string>()  {  [5] = "Tom",  [6] = "Sam",  [7] = "Bob"  };  foreach(var person in people)  {  Console.WriteLine($"key: {person.Key} value: {person.Value}");  } |

При переборе каждый элемент будет помещаться в переменную, которая представляет тип KeyValuePair, соответственно с помощью свойств Key и Value мы сможем получить ключ и значение элемента. Консольный вывод программы:

key: 5 value: Tom

key: 6 value: Sam

key: 7 value: Bob

### **Получение элементов**

Для обращения к элементам из словаря применяется их ключ, который передается в квадратных скобках:

|  |  |
| --- | --- |
| 1 | словарь[ключ] |

Таким образом мы можем получить и изменить элементы словаря

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | var people = new Dictionary<int, string>()  {  [5] = "Tom",  [6] = "Sam",  [7] = "Bob",  };  // получаем элемент по ключу 6  string sam = people[6]; // Sam  Console.WriteLine(sam); // Sam  // переустанавливаем значение по ключу 6  people[6] = "Mike";  Console.WriteLine(people[6]); // Mike    // добавляем новый элемент по ключу 22  people[22] = "Eugene";  Console.WriteLine(people[22]); // Eugene |

Более того, таким образом мы можем также добавить новый элемент в словарь. При установке значения по ключу, если элемент с таким ключом уже есть в словаре, то значение переустанавливается. Если же элемента с подобным ключом нет в словаре, то элемент добавляется.:

### **Методы и свойства Dictionary**

Среди методов класса Dictionary можно выделить следующие:

* **void Add(K key, V value)**: добавляет новый элемент в словарь
* **void Clear()**: очищает словарь
* **bool ContainsKey(K key)**: проверяет наличие элемента с определенным ключом и возвращает true при его наличии в словаре
* **bool ContainsValue(V value)**: проверяет наличие элемента с определенным значением и возвращает true при его наличии в словаре
* **bool Remove(K key)**: удаляет по ключу элемент из словаря  
  Другая версия этого метода позволяет получить удленный элемент в выходной параметр: **bool Remove(K key, out V value)**
* **bool TryGetValue(K key, out V value)**: получает из словаря элемент по ключу key. При успешном получении передает значение элемента в выходной параметр value и возвращает true
* **bool TryAdd(K key, V value)**: добавляет в словарь элемент с ключом key и значением value. При успешном добавлении возвращает true

Из свойств следует отметить свойство **Count**, которое возвращает количество элементов в словаре.

Применение методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | // условная телефонная книга  var phoneBook = new Dictionary<string, string>();    // добавляем элемент: ключ - номер телефона, значение - имя абонента  phoneBook.Add("+123456", "Tom");  // альтернативное добавление  // phoneBook["+123456"] = "Tom";    // Проверка наличия  var phoneExists1 = phoneBook.ContainsKey("+123456"); // true  Console.WriteLine($"+123456: {phoneExists1}");  var phoneExists2 = phoneBook.ContainsKey("+567456"); // false  Console.WriteLine($"+567456: {phoneExists2}");  var abonentExists1 = phoneBook.ContainsValue("Tom"); // true  Console.WriteLine($"Tom: {abonentExists1}");  var abonentExists2 = phoneBook.ContainsValue("Bob"); // false  Console.WriteLine($"Bob: {abonentExists2}");    // удаление элемента  phoneBook.Remove("+123456");    // проверяем количество элементов после удаления  Console.WriteLine($"Count: {phoneBook.Count}"); // Count: 0 |

## **Класс ObservableCollection**

Кроме стандартных классов коллекций типа списков, очередей, словарей, стеков .NET также предоставляет специальный класс **ObservableCollection<T>**. В отличие от ранее рассмотренных коллекций данный класс определен в пространстве имен **System.Collections.ObjectModel**. По функциональности коллекция ObservableCollection похожа на список List за тем исключением, что позволяет известить внешние объекты о том, что коллекция была изменена.

### **Создание и инициализация ObservableCollection**

Для создания объекта класс ObservableCollection предоставляет ряд конструкторов. Прежде всего мы можем создать пустую коллекцию:

|  |  |
| --- | --- |
| 1  2  3 | using System.Collections.ObjectModel;    ObservableCollection<string> people = new ObservableCollection<string>(); |

В данном случае коллекция people типизируется типом **string**, поэтому может хранить только строки.

Другая версия конструктора позволяет передать в ObservableCollection объекты из другой коллекции или массива:

|  |  |
| --- | --- |
| 1 | var people = new ObservableCollection<string>( new string[] {"Tom", "Bob", "Sam"}); |

Для инициализации можно через инициализатор в фигурных скобках передать значения

|  |  |
| --- | --- |
| 1  2  3  4 | var people = new ObservableCollection<string>  {  "Tom", "Bob", "Sam"  }; |

Также можно сочетать предыдущие два способа:

|  |  |
| --- | --- |
| 1  2  3  4 | var people = new ObservableCollection<string>( new string[] {"Mike", "Alice", "Kate" })  {  "Tom", "Bob", "Sam"  }; |

### **Обращение к элементам коллекции**

Для обращения к элементам ObservableCollection можно применять индексы на манер массивов или списков List:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var people = new ObservableCollection<string>  {  "Tom", "Bob", "Sam"  };    // получаем первый элемент  Console.WriteLine(people[0]); // Tom  // изменяем первый элемент  people[0] = "Tomas";  Console.WriteLine(people[0]); // Tomas |

### **Перебор коллекции**

Для перебора коллекции можно применять стандартные циклы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using System.Collections.ObjectModel;    var people = new ObservableCollection<string>  {  "Tom", "Bob", "Sam"  };    foreach(var person in people)  {  Console.WriteLine(person);  }  for (int i =0; i < people.Count; i++)  {  Console.WriteLine(people[i]);  } |

С помощью свойства **Count** можно получить количество элементов в коллекции.

### **Методы ObservableCollection**

Среди методов класса ObservableCollection можно отметить следующие:

* **void Add(T item)**: добавление нового элемента в коллекцию
* **void CopyTo(T[] array, int index,)**: копирует в массив array элементы из коллекции начиная с индекса index
* **bool Contains(T item)**: возвращает true, если элемент item есть в коллекции
* **void Clear()**: удаляет из коллекции все элементы
* **int IndexOf(T item)**: возвращает индекс первого вхождения элемента в коллекции
* **void Insert(int index, T item)**: вставляет элемент item в коллекцию по индексу index. Если такого индекса в коллекции нет, то генерируется исключение
* **bool Remove(T item)**: удаляет элемент item из коллекции, и если удаление прошло успешно, то возвращает true. Если в коллекции несколько одинаковых элементов, то удаляется только первый из них
* **void RemoveAt(int index)**: удаление элемента по указанному индексу index. Если такого индекса в коллекции нет, то генерируется исключение
* **void Move(int oldIndex, int newIndex)**: перемещает элемент с индекса oldIndex на позицию по индексу newIndex

Применение методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | using System.Collections.ObjectModel;    var people = new ObservableCollection<string>();    // добавляем элемент  people.Add("Bob");  // вставляем элемент по индексу 0  people.Insert(0, "Tom");    // проверка наличия элемента  bool bobExists = people.Contains("Bob"); // true  Console.WriteLine($"Bob exists: {bobExists}");  bool mikeExists = people.Contains("Mike"); // false  Console.WriteLine($"Mike exists: {mikeExists}");    // удаляем элемент  people.Remove("Tom");  // удаляем элемент по индексу 0  people.RemoveAt(0); |

### **Уведомление об измении коллекции**

Класс ObservableCollection определяет событие **CollectionChanged**, подписавшись на которое, мы можем обработать любые изменения коллекции. Данное событие представляет делегат **NotifyCollectionChangedEventHandler**:

|  |  |
| --- | --- |
| 1 | void NotifyCollectionChangedEventHandler(object? sender, NotifyCollectionChangedEventArgs e); |

Второй параметр делегата - объект **NotifyCollectionChangedEventArgs** хранит всю информацию о событии. В частности, его свойство Action позволяет узнать характер изменений. Оно хранит одно из значений из перечисления **NotifyCollectionChangedAction**:

* NotifyCollectionChangedAction.Add: добавление
* NotifyCollectionChangedAction.Remove: удаление
* NotifyCollectionChangedAction.Replace: замена
* NotifyCollectionChangedAction.Move: перемещение объекта внутри коллекции на новую позицию
* NotifyCollectionChangedAction.Reset: сброс содержимого коллекции (например, при ее очистке с помощью метода Clear())

Кроме того, свойства NewItems и OldItems позволяют получить соответственно добавленные и удаленные объекты. Таким образом, мы получаем полный контроль над обработкой добавления, удаления и замены объектов в коллекции.

Допустим, у нас будет следующий класс Person, который представляет пользователя:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Person  {  public string Name { get; }  public Person(string name) => Name = name;  } |

Для управления коллекцией объектов Person определим следующую программу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | using System.Collections.ObjectModel;  using System.Collections.Specialized;    var people = new ObservableCollection<Person>()  {  new Person("Tom"),  new Person("Sam")  };  // подписываемся на событие изменения коллекции  people.CollectionChanged += People\_CollectionChanged;    people.Add(new Person("Bob")); // добавляем новый элемент    people.RemoveAt(1); // удаляем элемент  people[0] = new Person("Eugene"); // заменяем элемент    Console.WriteLine("\nСписок пользователей:");  foreach (var person in people)  {  Console.WriteLine(person.Name);  }  // обработчик изменения коллекции  void People\_CollectionChanged(object? sender, NotifyCollectionChangedEventArgs e)  {  switch (e.Action)  {  case NotifyCollectionChangedAction.Add: // если добавление  if(e.NewItems?[0] is Person newPerson)  Console.WriteLine($"Добавлен новый объект: {newPerson.Name}");  break;  case NotifyCollectionChangedAction.Remove: // если удаление  if (e.OldItems?[0] is Person oldPerson)  Console.WriteLine($"Удален объект: {oldPerson.Name}");  break;  case NotifyCollectionChangedAction.Replace: // если замена  if ((e.NewItems?[0] is Person replacingPerson) &&  (e.OldItems?[0] is Person replacedPerson))  Console.WriteLine($"Объект {replacedPerson.Name} заменен объектом {replacingPerson.Name}");  break;  }  } |

Здесь в качестве обработчика изменений коллекции выступает метод People\_CollectionChanged, в котором с помощью параметра NotifyCollectionChangedEventArgs получаем информацию об изменении. Консольный вывод программы:

Добавлен новый объект: Bob

Удален объект: Sam

Объект Tom заменен объектом Eugene

Список пользователей:

Eugene

Bob

## **Интерфейсы IEnumerable и IEnumerator**

Как мы увидели, основной для большинства коллекций является реализация интерфейсов IEnumerable и IEnumerator. Благодаря такой реализации мы можем перебирать объекты в цикле foreach:

|  |  |
| --- | --- |
| 1  2  3  4 | foreach(var item in перечислимый\_объект)  {    } |

Перебираемая коллекция должна реализовать интерфейс IEnumerable.

Интерфейс IEnumerable имеет метод, возвращающий ссылку на другой интерфейс - перечислитель:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface IEnumerable  {  IEnumerator GetEnumerator();  } |

А интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public interface IEnumerator  {  bool MoveNext(); // перемещение на одну позицию вперед в контейнере элементов  object Current {get;} // текущий элемент в контейнере  void Reset(); // перемещение в начало контейнера  } |

Метод **MoveNext()** перемещает указатель на текущий элемент на следующую позицию в последовательности. Если последовательность еще не закончилась, то возвращает true. Если же последовательность закончилась, то возвращается false.

Свойство **Current** возвращает объект в последовательности, на который указывает указатель.

Метод **Reset()** сбрасывает указатель позиции в начальное положение.

Каким именно образом будет осуществляться перемещение указателя и получение элементов зависит от реализации интерфейса. В различных реализациях логика может быть построена различным образом.

Например, без использования цикла foreach перебирем массив с помощью интерфейса IEnumerator:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | using System.Collections;    string[] people = {"Tom", "Sam", "Bob"};    IEnumerator peopleEnumerator = people.GetEnumerator(); // получаем IEnumerator  while (peopleEnumerator.MoveNext()) // пока не будет возвращено false  {  string item = (string)peopleEnumerator.Current; // получаем элемент на текущей позиции  Console.WriteLine(item);  }  peopleEnumerator.Reset(); // сбрасываем указатель в начало массива |

### **Реализация IEnumerable и IEnumerator**

Рассмотрим простешую реализацию **IEnumerable** на примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using System.Collections;    Week week = new Week();  foreach (var day in week)  {  Console.WriteLine(day);  }    class Week : IEnumerable  {  string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",  "Friday", "Saturday", "Sunday" };  public IEnumerator GetEnumerator() => days.GetEnumerator();  } |

В данном случае класс Week, который представляет неделю и хранит все дни недели, реализует интерфейс IEnumerable. Однако в данном случае мы поступили очень просто - вместо реализации IEnumerator мы просто возвращаем в методе GetEnumerator объект IEnumerator для массива.

|  |  |
| --- | --- |
| 1 | public IEnumerator GetEnumerator() => days.GetEnumerator(); |

Благодаря этому мы можем перебрать все дни недели в цикле foreach.

В то же время стоит отметить, что для перебора коллекции через foreach в принципе необязательно реализовать интерфейс IEnumerable. Достаточно в классе определить публичный метод **GetEnumerator**, который бы возвращал объект IEnumerator. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class Week  {  string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",  "Friday", "Saturday", "Sunday" };  public IEnumerator GetEnumerator() =>days.GetEnumerator();  } |

Однако это было довольно просто - мы просто используем уже готовый перчислитель массива. Однако, возможно, потребуется задать свою собственную логику перебора объектов. Для этого реализуем интерфейс **IEnumerator**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | using System.Collections;    class WeekEnumerator : IEnumerator  {  string[] days;  int position = -1;  public WeekEnumerator(string[] days) => this.days = days;  public object Current  {  get  {  if (position == -1 || position >= days.Length)  throw new ArgumentException();  return days[position];  }  }  public bool MoveNext()  {  if (position < days.Length - 1)  {  position++;  return true;  }  else  return false;  }  public void Reset() => position = -1;  }  class Week  {  string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",  "Friday", "Saturday", "Sunday" };  public IEnumerator GetEnumerator() => new WeekEnumerator(days);  } |

Здесь теперь класс Week использует не встроенный перечислитель, а WeekEnumerator, который реализует IEnumerator.

Ключевой момент при реализации перечислителя - перемещения указателя на элемент. В классе WeekEnumerator для хранения текущей позиции определена переменная position. Следует учитывать, что в самом начале (в исходном состоянии) указатель должен указывать на позицию условно перед первым элементом. Когда будет производиться цикл foreach, то данный цикл вначале вызывает метод MoveNext и фактически перемещает указатель на одну позицию в перед и только затем обращается к свойству Current для получения элемента в текущей позиции.

Затем в программе мы можем аналогичным образом перебирать объект Week с помощью цикла foreach:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Week week = new Week();  foreach(var day in week)  {  Console.WriteLine(day);  } |

### **Обобщенная версия IEnumerator**

В примерах выше использовались необобщенные версии интерфейсов, однако мы также можем использовать их обобщенные двойники:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | using System.Collections;    class WeekEnumerator : IEnumerator<string>  {  string[] days;  int position = -1;  public WeekEnumerator(string[] days) => this.days = days;  public string Current  {  get  {  if (position == -1 || position >= days.Length)  throw new ArgumentException();  return days[position];  }  }  object IEnumerator.Current => Current;  public bool MoveNext()  {  if (position < days.Length - 1)  {  position++;  return true;  }  else  return false;  }  public void Reset() => position = -1;  public void Dispose() { }  }  class Week  {  string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",  "Friday", "Saturday", "Sunday" };  public IEnumerator<string> GetEnumerator() => new WeekEnumerator(days);  } |

В данном случае реализуем интерфейс IEnumerator<string>, соответственно в свойстве Current нам надо возвратить объект string. В этом случае при переборе в цикле foreach перебираемые объекты будут автоматически представлять тип **string**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Week week = new Week();  foreach (string day in week)  {  Console.WriteLine(day);  } |

## **Итераторы и оператор yield**

**Итератор** по сути представляет блок кода, который использует оператор **yield** для перебора набора значений. Данный блок кода может представлять тело метода, оператора или блок get в свойствах.

Итератор использует две специальных инструкции:

* **yield return**: определяет возвращаемый элемент
* **yield break**: указывает, что последовательность больше не имеет элементов

Рассмотрим небольшой пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Numbers numbers = new Numbers();  foreach (int n in numbers)  {  Console.WriteLine(n);  }    class Numbers  {  public IEnumerator<int> GetEnumerator()  {  for (int i = 0; i < 6; i++)  {  yield return i \* i;  }  }  } |

В классе Numbers метод **GetEnumerator()** фактически представляет итератор. С помощью оператора **yield return** возвращается некоторое значение (в данном случае квадрат числа).

В программе с помощью цикла foreach мы можем перебрать объект Numbers как обычную коллекцию. При получении каждого элемента в цикле foreach будет срабатывать оператор yield return, который будет возвращать один элемент и запоминать текущую позицию.

Благодаря итераторам мы можем пойти дальше и легко реализовать перебор числа в цикле foreach:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | foreach(var n in 5) Console.WriteLine(n);  foreach (var n in -5) Console.WriteLine(n);    static class Int32Extension  {  public static IEnumerator<int> GetEnumerator(this int number)  {  int k = (number > 0)? number: 0;  for (int i = number - k; i <= k; i++) yield return i;  }  } |

В данном случае итератор реализован как метод расширения для типа int или System.Int32. В методе итератора фактически возвращаем все целочисленные значения от 0 до текущего числа. Консольный вывод:

0

1

2

3

4

5

-5

-4

-3

-2

-1

0

Другой пример: пусть у нас есть коллекция Company, которая представляет компанию и которая хранит в массиве personnel штат сотрудников - объектов Person. Используем оператор yield для перебора этой коллекции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class Person  {  public string Name { get; }  public Person(string name) =>Name = name;  }  class Company  {  Person[] personnel;  public Company(Person[] personnel) => this.personnel = personnel;  public int Length => personnel.Length;  public IEnumerator<Person> GetEnumerator()  {  for (int i = 0; i < personnel.Length; i++)  {  yield return personnel[i];  }  }  } |

Метод GetEnumerator() представляет итератор. И когда мы будем осуществлять перебор в объекте Company в цикле foreach, то будет идти обращение к вызову yield return personnel[i];. При обращении к оператору yield return будет сохраняться текущее местоположение. И когда метод foreach перейдет к следующей итерации для получения нового объекта, итератор начнет выполнения с этого местоположения.

Ну и в основной программе в цикле foreach выполняется собственно перебор, благодаря реализации итератора:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var people = new Person[]  {  new Person("Tom"),  new Person("Bob"),  new Person("Sam")  };  var microsoft = new Company(people);    foreach(Person employee in microsoft)  {  Console.WriteLine(employee.Name);  } |

Хотя при реализации итератора в методе GetEnumerator() применялся перебор массива в цикле for, но это необязательно делать. Мы можем просто определить несколько вызовов оператора yield return:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public IEnumerator<Person> GetEnumerator()  {  yield return personnel[0];  yield return personnel[1];  yield return personnel[2];  } |

В этом случае при каждом вызове оператора yield return итератор также будет запоминать текущее местоположение и при последующих вызовах начинать с него.

### **Именованный итератор**

Выше для создания итератора мы использовали метод GetEnumerator. Но оператор yield можно использовать внутри любого метода, только такой метод должен возвращать объект интерфейса IEnumerable. Подобные методы еще называют **именованными итераторами**.

Создадим такой именованный итератор в классе Company и используем его:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class Person  {  public string Name { get; }  public Person(string name) =>Name = name;  }  class Company  {  Person[] personnel;  public Company(Person[] personnel) => this.personnel = personnel;  public int Length => personnel.Length;  public IEnumerable<Person> GetPersonnel(int max)  {  for (int i = 0; i < max; i++)  {  if (i == personnel.Length)  {  yield break;  }  else  {  yield return personnel[i];  }  }  }  } |

Определенный здесь итератор - метод IEnumerable GetPersonnel(int max) в качестве параметра принимает количество выводимых объектов. В процессе работы программы может сложиться, что его значение будет больше, чем длина массива personnel. И чтобы не произошло ошибки, используется оператор **yield break**. Этот оператор прерывает выполнение итератора.

Применение итератора:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var people = new Person[]  {  new Person("Tom"),  new Person("Bob"),  new Person("Sam")  };  var microsoft = new Company(people);    foreach(Person employee in microsoft.GetPersonnel(5))  {  Console.WriteLine(employee.Name);  } |

Вызов microsoft.GetPersonnel(5) будет возвращать набор из не более чем 5 объектов Person. Но так как у нас всего три таких объекта, то в методе GetPersonnel после трех операций сработает оператор yield break.

## **Класс Array и массивы**

Все массивы в C# построены на основе класса **Array** из пространства имен System. Этот класс определяет ряд свойств и методов, которые мы можем использовать при работе с массивами. Основные свойства и методы:

* Свойство **Length** возвращает длину массива
* Свойство **Rank** возвращает размерность массива
* **int BinarySearch (Array array, object? value)** выполняет бинарный поиск в отсортированном массиве и возвращает индекс найденного элемента
* **void Clear (Array array)** очищает массив, устанавливая для всех его элементов значение по умолчанию
* **void Copy (Array sourceArray, int sourceIndex, Array destinationArray, int destinationIndex, int length)** копирует из массива sourceArray начиная с индекс sourceIndex length элементов в массив destinationArray начиная с индекса destinationIndex
* **bool Exists<T> (T[] array, Predicate<T> match)** проверяет, содержит ли массив array элементы, которые удовлеворяют условию делегата match
* **void Fill<T> (T[] array, T value)** заполняет массив array значением value
* **T? Find<T> (T[] array, Predicate<T> match)** находит первый элемент, который удовлеворяет определенному условию из делегата match. Если элемент не найден, то возвращается null
* **T? FindLast<T> (T[] array, Predicate<T> match)** находит последний элемент, который удовлеворяет определенному условию из делегата match. Если элемент не найден, то возвращается null
* **int FindIndex<T> (T[] array, Predicate<T> match)** возвращает индекс первого вхождения элемента, который удовлеворяет определенному условию делегата match
* **int FindLastIndex<T> (T[] array, Predicate<T> match)** возвращает индекс последнего вхождения элемента, который удовлеворяет определенному условию
* **T[] FindAll<T> (T[] array, Predicate<T> match)** возвращает все элементы в виде массива, которые удовлеворяет определенному условию из делегата match
* **int IndexOf (Array array, object? value)** возвращает индекс первого вхождения элемента в массив
* **int LastIndexOf (Array array, object? value)** возвращает индекс последнего вхождения элемента в массив
* **void Resize<T> (ref T[]? array, int newSize)** изменяет размер одномерного массива
* **void Reverse (Array array)** располагает элементы массива в обратном порядке
* **void Sort (Array array)** сортирует элементы одномерного массива

Разберем самые используемые методы.

### **Сортировка массива**

Отсортируем массив с помощью метода **Sort()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | string[] people = { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };    Array.Sort(people);    foreach (var person in people)  Console.Write($"{person} ");    // Alice Bob Kate Sam Tom Tom |

Этот метод имеет много перегрузок. Например, одна из версий позволяет отсортировать только часть массива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string[] people = { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };    // сортируем с 1 индекса 3 элемента  Array.Sort(people, 1, 3);    foreach (var person in people)  Console.Write($"{person} ");    // Tom Bob Kate Sam Tom Alice |

### **Поиск индекса элемента**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | string[] people = { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };    // находим индекс первого элемента "Tom"  int tomFirstIndex = Array.IndexOf(people, "Tom");  // находим индекс последнего элемента "Tom"  int tomLastIndex = Array.LastIndexOf(people, "Tom");  // находим индекс первого элемента, у которого длина строки больше 3  int lengthFirstIndex = Array.FindIndex(people, person => person.Length > 3);  // находим индекс последнего элемента, у которого длина строки больше 3  int lengthLastIndex = Array.FindLastIndex(people, person => person.Length > 3);    Console.WriteLine($"tomFirstIndex: {tomFirstIndex}"); // 0  Console.WriteLine($"tomLastIndex: {tomLastIndex}"); // 4  Console.WriteLine($"lengthFirstIndex: {lengthFirstIndex}"); // 3  Console.WriteLine($"lengthLastIndex: {lengthLastIndex}"); // 5      // находим индекс элемента "Bob"  // для бинарного поиска коллекцию нужно отсортировать  Array.Sort(people); // Alice Bob Kate Sam Tom Tom  int bobIndex = Array.BinarySearch(people, "Bob");  Console.WriteLine($"bobIndex: {bobIndex}"); // 1 |

Если элемент не найден в массиве, то методы возвращают -1.

### **Поиск элемента по условию**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | string[] people = { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };    // находим первый и последний элементы  // где длина строки больше 3 символов  string? first = Array.Find(people, person => person.Length > 3);  Console.WriteLine(first); // Kate  string? last = Array.FindLast(people, person => person.Length > 3);  Console.WriteLine(last); // Alice    // находим элементы, у которых длина строки равна 3  string[] group = Array.FindAll(people, person => person.Length == 3);  foreach (var person in group) Console.WriteLine(person);  // Tom Sam Bob Tom |

### **Изменение порядка элементов массива**

Например, изменим порядок элементов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | string[] people = { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };    Array.Reverse(people);    foreach (var person in people)  Console.Write($"{person} ");  // "Alice", "Tom", "Kate", "Bob", "Sam", "Tom" |

Также можно изменить порядок только части элементов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | string[] people = { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };    // изменяем порядок 3 элементов начиная c индекса 1  Array.Reverse(people, 1, 3);    foreach (var person in people)  Console.Write($"{person} ");  // "Tom", "Kate", "Bob", "Sam", "Tom", "Alice" |

В данном случае изменяем порядок только 3 элементов начиная c индекса 1.

### **Изменение размера массива**

Для изменения размера массива применяется метод Resize. Его первый параметр - изменяемый массив, а второй параметр - количество элементов, которые должны быть в массиве. Если второй параметр меньше длины массива, то массив усекается. Если значение параметра, наоборот, больше, то массив дополняется дополнительными элементами, которые имеют значение по умолчанию. Причем первый параметр передается по ссылке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | string[] people = { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };    // уменьшим массив до 4 элементов  Array.Resize(ref people, 4);    foreach (var person in people)  Console.Write($"{person} ");  // "Tom", "Sam", "Bob", "Kate" |

### **Копирование массива**

Метод **Copy** копирует часть одного массива в другой:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | string[] people = { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };    var employees = new string[3];    // копируем 3 элемента из массива people c индекса 1  // и вставляем их в массив employees начиная с индекса 0  Array.Copy(people,1, employees,0, 3);    foreach (var person in employees)  Console.Write($"{person} ");  // Sam Bob Kate |

В данном случае копируем 3 элемента из массива people начиная c индекса 1 и вставляем их в массив employees начиная с индекса 0.Тип **Span** представляет непрерывную область памяти. Цель данного типа - повысить производительность и эффективность использования памяти. Span позволяет избежать дополнительных выделений памяти при операции с наборами данных. Поскольку Span является структурой, то объект этого типа располагаетс в стеке, а не в хипе.

### **Создание Span**

Для создания объекта Span можно применять один из его конструкторов:

* Span(): создает пустой объект Span
* Span(T item): создает объект Span с одним элементом item
* Span(T[] array): создает объект Span из массива array
* Span(void\* pointer, int length): создает объект Span, который получает length байт памяти, начиная с указателя pointer
* Span(T[] array, int start, int length): создает объект Span, который получает из массива array length элементов, начиная с индекса start

Например, простейшее создание Span:

|  |  |
| --- | --- |
| 1 | Span<string> people = ["Tom", "Bob", "Sam"]; |

В данном случае Span будет хранить ссылки на три строки.

Нередко Span создается на основе каких-то других наборов данных:

|  |  |
| --- | --- |
| 1  2 | string[] people = { "Tom", "Alice", "Bob" };  Span<string> peopleSpan = new Span<string>(people); |

Мы также можем непосредственно присвоить массив, и он неявно будет преобразован в Span:

|  |  |
| --- | --- |
| 1  2 | string[] people = { "Tom", "Alice", "Bob" };  Span<string> peopleSpan = people; |

Далее мы можем получать, устанавливать или перебирать данные также, как в массиве:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | string[] people = { "Tom", "Alice", "Bob" };  Span<string> peopleSpan = people;  peopleSpan[1] = "Ann"; // переустановка значения элемента  Console.WriteLine(peopleSpan[2]); // получение элемента  Console.WriteLine(peopleSpan.Length); // получение длины Span    // перебор Span  foreach (var s in peopleSpan)  {  Console.WriteLine(s);  } |

Если Span ведет себя внешне как массив, то в чем его преимущество или когда он нам может пригодиться? Рассмотрим простейшую ситуацию - у нас есть массив со значениями дневных температур воздуха за месяц, и нам надо получить их него два набора - набор температур за первую декаду и за последнюю декаду. Используя массивы, мы могли бы сделать так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int[] temperatures =  {  10, 12, 13, 14, 15, 11, 13, 15, 16, 17,  18, 16, 15, 16, 17, 14, 9, 8, 10, 11,  12, 14, 15, 15, 16, 15, 13, 12, 12, 11  };  int[] firstDecade = new int[10]; // выделяем память для первой декады  int[] lastDecade = new int[10]; // выделяем память для второй декады  Array.Copy(temperatures, 0, firstDecade, 0, 10); // копируем данные в первый массив  Array.Copy(temperatures, 20, lastDecade, 0, 10); // копируем данные во второй массив |

Для хранения данных создаются два дополнительных массива для дневных температур каждой декады. С помощью метода Array.Copy данные из исходного массива temperatures копируются в два остальных массива. Но суть в данном случае в том, что для обоих массивов мы вынуждены выделить память. То есть оба массива по сути содержат те же данные, что и temperatures, однако в отдельных частях памяти.

Span позволяет работать с памятью более эффективно и избежать ненужных выделений памяти. Так, используем вместо массивов Span:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int[] temperatures =  {  10, 12, 13, 14, 15, 11, 13, 15, 16, 17,  18, 16, 15, 16, 17, 14, 9, 8, 10, 11,  12, 14, 15, 15, 16, 15, 13, 12, 12, 11  };  Span<int> temperaturesSpan = temperatures;    Span<int> firstDecade = temperaturesSpan.Slice(0, 10); // нет выделения памяти под данные  Span<int> lastDecade = temperaturesSpan.Slice(20, 10); // нет выделения памяти под данные |

Для создания производных объектов Span применяется метод Slice, который из Spana выделяет часть и возвращает ее в виде другого объекта Span. Теперь объекты Span firstDecade и lastDecade работают с теми же данными, что и temperaturesSpan, а дополнительно память не выделяется. То есть во всех трех случаях мы работаем с тем же массивом temperatures. Мы даже можем в одном Span изменить данные, и данные изменятся в другом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | int[] temperatures =  {  10, 12, 13, 14, 15, 11, 13, 15, 16, 17,  18, 16, 15, 16, 17, 14, 9, 8, 10, 11,  12, 14, 15, 15, 16, 15, 13, 12, 12, 11  };  Span<int> temperaturesSpan = temperatures;    Span<int> firstDecade = temperaturesSpan.Slice(0, 10);    temperaturesSpan[0] = 25; // меняем в temperatureSpan  Console.WriteLine(firstDecade[0]); //25 |

За счет чего это достигается? Для понимания работы Span можно обратиться к исходному коду типа. В частности, мы можем в нем увидеть следующее свойство:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public readonly ref struct Span<T>  {  //....  public ref T this[int index] { get { ... } }  //....  } |

Здесь мы видим, что индексатор возвращает ref-ссылку, благодаря чем мы получаем доступ непосредственно к объекту и можем его изменять.

В данном случае, конечно, преимущества от отсутствия выделения дополнительной памяти под хранение объектов минимальны. Но при более интенсивной работе с данными выигрыш в производительности неизбежно должен возрастать.

### **Методы Span**

Основные методы Span:

* void Fill(T value): заполняет все элементы Span значением value
* T[] ToArray(): преобразует Span в массив
* Span<T> Slice(int start, int length): выделяет из Span length элементов начиная с индекса start в виде другого Span
* void Clear(): очищает Span
* void CopyTo(Span<T> destination): копирует элементы текущего Span в другой Span
* bool TryCopyTo(Span<T> destination): копирует элементы текущего Span в другой Span, но при этом также возвращает значение bool, которое указывает, удачно ли прошла операция копирования

### **ReadOnlySpan**

Структура **ReadOnlySpan** аналогична Span, только предназначена для неизменяемых данных. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | string text = "hello, world";  string worldString = text.Substring(startIndex: 7, length: 5); // есть выделение памяти под символы  ReadOnlySpan<char> worldSpan = text.AsSpan().Slice(start: 7, length: 5); // нет выделения памяти под символы  //worldSpan[0] = 'a'; // Нельзя изменить  Console.WriteLine(worldSpan[0]); // выводим первый символ    // перебор символов  foreach(var c in worldSpan)  {  Console.Write(c);  } |

В данном случае с помощью метода **AsSpan()** преобразуем строку в объект ReadOnlySpan<char> и затем выделяем из него диапазон символов "world". Поскольку ReadOnlySpan предназначен только для чтения, то соответственно мы не можем изменить через него данные, но получить можем. В остальном работа с ReadOnlySpan идет так же, как с Span.

### **Ограничения Span**

Как структура, определенная с модификатором ref, Span имеет ряд ограничений: она не может быть присвоена переменной типа Object, dynamic или переменной типа интерфейса. Она не может быть полем в объекте ссылочного типа (а только внутри ref-структур). Она не может использоваться в пределах операций await или yield.

## **Индексы и диапазоны**

В C# 8.0 была добавлена новая функциональность - индексы и диапазоны, которые упрощают получение из массивов подмассивов. Для этого в C# есть два типа: **System.Range** и **System.Index**. Оба типа являются структурами. Тип Range представляет некоторый диапазон значений в некоторой последовательность, а тип Index - индекс в последовательности.

### **Индексы**

Индекс фактически представляет числовое значение, и при определении индекса мы можем указать это значение:

|  |  |
| --- | --- |
| 1 | Index myIndex = 2; |

В данном случае индекс представляет третий элемент последовательности (индексация начинается с 0).

С помощью специального оператора **^** можно задать индекс относительно конца последовательности.

|  |  |
| --- | --- |
| 1 | Index myIndex = ^2; |

Теперь индекс представляет второй элемент с конца последовательности, то есть предпоследний элемент.

Используем индексы для получения элементов массива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Index myIndex1 = 2; // третий элемент  Index myIndex2 = ^2; // предпоследний элемент    string[] people = { "Tom", "Bob", "Sam", "Kate", "Alice" };  string selected1 = people[myIndex1]; // Sam  string selected2 = people[myIndex2]; // Kate  Console.WriteLine(selected1);  Console.WriteLine(selected2); |

Фактически для данной задачи индексы не нужны, и мы можем воспользоваться стандартными возможностями массивов:

|  |  |
| --- | --- |
| 1  2  3  4  5 | string[] people = { "Tom", "Bob", "Sam", "Kate", "Alice" };  string selected1 = people[2]; // Sam  string selected2 = people[people.Length - 2]; // Kate  Console.WriteLine(selected1);  Console.WriteLine(selected2); |

То есть в подобных ситуациях плюсом индексов является большая удобочитаемость. Так, people[^2] более читабельно, чем people[people.Length - 2].

### **Диапазон**

Диапазон представляет часть последовательности, которая ограничена двумя индексами. Начальный индекс включается в диапазон, а конечный индекс НЕ входит в диапазон. Для определения диапазона применяется оператор **..**:

|  |  |
| --- | --- |
| 1 | Range myRange1 = 1..4; // по 1-го индекса включая по 4-й индекс не включая |

В данном случае диапазон myRange1 включает элементы с 1 индекса по 4-й индекс (не включая). При этом элемент по 4-му индексу не включается в диапазон. При этом границы диапазона задаются не просто числами, а именно объектами Index. То есть следующие определения диапазонов будут равноценны:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Index start = 1;  Index end = 4;  Range myRange1 = start..end;    Range myRange2 = 1..4; |

Практическое применение диапазонов - получим со второго по четвертый элементы массива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | string[] people = { "Tom", "Bob", "Sam", "Kate", "Alice" };  string[] peopleRange = people[1..4]; // получаем 2, 3 и 4-й элементы из массива  foreach(var person in peopleRange)  {  Console.WriteLine(person);  } |

Результатом операции people[1..4] является подмассив элементов с 1 по 3 индексы (включая). Консольный вывод:

Bob

Sam

Kate

Мы можем задать для диапазона только конечный индекс. В этом случае начальным индексом по умолчанию будет 0.

|  |  |
| --- | --- |
| 1  2 | string[] people = { "Tom", "Bob", "Sam", "Kate", "Alice" };  string[] peopleRange = people[..4]; // Tom, Bob, Sam, Kate |

Либо, наоборот, задать только начальный индекс, тогда конечным индексом будет последний индекс последовательности:

|  |  |
| --- | --- |
| 1  2 | string[] people = { "Tom", "Bob", "Sam", "Kate", "Alice" };  string[] peopleRange = people[1..]; // Bob, Sam, Kate, Alice |

Используя индексы относительно конца последовательности, можно получать диапазон относительно конца последовательности:

|  |  |
| --- | --- |
| 1  2  3  4 | string[] people = { "Tom", "Bob", "Sam", "Kate", "Alice" };  string[] peopleRange1 = people[^2..]; // два последних - Kate, Alice  string[] peopleRange2 = people[..^1]; // начиная с предпоследнего - Tom, Bob, Sam, Kate  string[] peopleRange3 = people[^3..^1]; // два начиная с предпоследнего - Sam, Kate |

Кроме массивов индексы и диапазоны также применяются к объектам Span и ReadOnlySpan:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | string[] people = { "Tom", "Bob", "Sam", "Kate", "Alice" };  Span<string> peopleSpan = people;  Span<string> selectedPeopleSpan = peopleSpan[1..4];  foreach (var person in selectedPeopleSpan)  {  Console.WriteLine(person);  } |

# **Работа со строками**

## **Строки и класс String**

Довольно большое количество задач, которые могут встретиться при разработке приложений, так или иначе связано с обработкой строк - парсинг веб-страниц, поиск в тексте, какие-то аналитические задачи, связанные с извлечением нужной информации из текста и т.д. Поэтому в этом плане работе со строками уделяется особое внимание.

В языке C# строковые значения представляет тип **string**, а вся функциональность работы с данным типом сосредоточена в классе **System.String**. Собственно string является псевдонимом для класса String. Объекты этого класса представляют текст как последовательность символов Unicode. Максимальный размер объекта String может составлять в памяти 2 ГБ, или около 1 миллиарда символов.

### **Создание строк**

Создавать строки можно, как используя переменную типа string и присваивая ей значение, так и применяя один из конструкторов класса String:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string s1 = "hello";  string s2 = new String('a', 6); // результатом будет строка "aaaaaa"  string s3 = new String(new char[] { 'w', 'o', 'r', 'l', 'd' });  string s4 = new String(new char[] { 'w', 'o', 'r', 'l', 'd' }, 1, 3); // orl    Console.WriteLine(s1); // hello  Console.WriteLine(s2); // aaaaaaa  Console.WriteLine(s3); // world  Console.WriteLine(s4); // orl |

Конструктор String имеет различное число версий. Так, вызов конструктора

|  |  |
| --- | --- |
| 1 | new String('a', 6) |

6 раз повторит объект из первого параметра, то есть фактически создаст строку "aaaaaa".

Еще один конструктор принимает массив символов, из которых создается строка

|  |  |
| --- | --- |
| 1 | string s3 = new String(new char[] { 'w', 'o', 'r', 'l', 'd' }); |

Третий использованный выше в примере конструктор позволяет создать строку из части массива символов. Второй параметр передает начальный индекс, с которого извлкаются символы, а третий параметр указывает на количество символов:

|  |  |
| --- | --- |
| 1 | string s4 = new String(new char[] { 'w', 'o', 'r', 'l', 'd' }, 1, 3); // orl |

### **Строка как набор символов**

Так как строка хранит коллекцию символов, в ней определен индексатор для доступа к этим символам:

|  |  |
| --- | --- |
| 1 | public char this[int index] {get;} |

Применяя индексатор, мы можем обратиться к строке как к массиву символов и получить по индексу любой из ее символов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | string message = "hello";  // получаем символ  char firstChar = message[1]; // символ 'e'  Console.WriteLine(firstChar); //e    Console.WriteLine(message.Length); // длина строки |

Используя свойство **Length**, как и в обычном массиве, можно получить длину строки.

### **Перебор строк**

Класс String реализует интерфейс IEnumerable, благодаря чему строку можно перебрать в цикле foreach как набор объектов char. Также можно с помощью других типов циклов перебрать строку, применяя обращение к символам по индексу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | string message = "hello";    for(var i =0; i < message.Length; i++)  {  Console.WriteLine(message[i]);  }  foreach(var ch in message)  {  Console.WriteLine(ch);  } |

### **Сравнение строк**

В отличие от других классов строки сравниваются по значению их символов, а не по ссылкам:

|  |  |
| --- | --- |
| 1  2  3  4 | string message1 = "hello";  string message2 = "hello";    Console.WriteLine(message1 == message2); // true |

### **Многострочные строки**

Начиная с C# 11 с помощью трех пар двойных кавычек можно оформить многострочный текст, в том числе с применением интерполяции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | Print();  PrintValue("hello");    void Print()  {  string text = """  <element attr="content">  <body>  </body>  </element>  """;  Console.WriteLine(text);  }    void PrintValue(string val)  {  string text = $"""  <element attr="content">  <body>  {val}  </body>  </element>  """;  //// или так  //string text = $$"""  // <element attr="content">  // <body>  // {{val}}  // </body>  // </element>  // """;  Console.WriteLine(text);  } |

### **Основные методы строк**

Основная функциональность класса String раскрывается через его методы, среди которых можно выделить следующие:

* **Compare**: сравнивает две строки с учетом текущей культуры (локали) пользователя
* **CompareOrdinal**: сравнивает две строки без учета локали
* **Contains**: определяет, содержится ли подстрока в строке
* **Concat**: соединяет строки
* **CopyTo**: копирует часть строки, начиная с определенного индекса в массив
* **EndsWith**: определяет, совпадает ли конец строки с подстрокой
* **Format**: форматирует строку
* **IndexOf**: находит индекс первого вхождения символа или подстроки в строке
* **Insert**: вставляет в строку подстроку
* **Join**: соединяет элементы массива строк
* **LastIndexOf**: находит индекс последнего вхождения символа или подстроки в строке
* **Replace**: замещает в строке символ или подстроку другим символом или подстрокой
* **Split**: разделяет одну строку на массив строк
* **Substring**: извлекает из строки подстроку, начиная с указанной позиции
* **ToLower**: переводит все символы строки в нижний регистр
* **ToUpper**: переводит все символы строки в верхний регистр
* **Trim**: удаляет начальные и конечные пробелы из строки

В следующих статьях разберем работу этих методов.

## **Операции со строками**

### **Объединение строк**

Конкатенация строк или объединение может производиться как с помощью операции +, так и с помощью метода **Concat**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | string s1 = "hello";  string s2 = "world";  string s3 = s1 + " " + s2; // результат: строка "hello world"  string s4 = string.Concat(s3, "!!!"); // результат: строка "hello world!!!"    Console.WriteLine(s4); |

Метод Concat является статическим методом класса string, принимающим в качестве параметров две строки. Также имеются другие версии метода, принимающие другое количество параметров.

Для объединения строк также может использоваться метод **Join**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string s5 = "apple";  string s6 = "a day";  string s7 = "keeps";  string s8 = "a doctor";  string s9 = "away";  string[] values = new string[] { s5, s6, s7, s8, s9 };    string s10 = string.Join(" ", values);  Console.WriteLine(s10); // apple a day keeps a doctor away |

Метод Join также является статическим. Использованная выше версия метода получает два параметра: строку-разделитель (в данном случае пробел) и массив строк, которые будут соединяться и разделяться разделителем.

### **Сравнение строк**

Для сравнения строк применяется статический метод **Compare**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | string s1 = "hello";  string s2 = "world";    int result = string.Compare(s1, s2);  if (result<0)  {  Console.WriteLine("Строка s1 перед строкой s2");  }  else if (result > 0)  {  Console.WriteLine("Строка s1 стоит после строки s2");  }  else  {  Console.WriteLine("Строки s1 и s2 идентичны");  }  // результатом будет "Строка s1 перед строкой s2" |

Данная версия метода Compare принимает две строки и возвращает число. Если первая строка по алфавиту стоит выше второй, то возвращается число меньше нуля. В противном случае возвращается число больше нуля. И третий случай - если строки равны, то возвращается число 0.

В данном случае так как символ h по алфавиту стоит выше символа w, то и первая строка будет стоять выше.

### **Поиск в строке**

С помощью метода **IndexOf** мы можем определить индекс первого вхождения отдельного символа или подстроки в строке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | string s1 = "hello world";  char ch = 'o';  int indexOfChar = s1.IndexOf(ch); // равно 4  Console.WriteLine(indexOfChar);    string substring = "wor";  int indexOfSubstring = s1.IndexOf(substring); // равно 6  Console.WriteLine(indexOfSubstring); |

Подобным образом действует метод **LastIndexOf**, только находит индекс последнего вхождения символа или подстроки в строку.

Еще одна группа методов позволяет узнать начинается или заканчивается ли строка на определенную подстроку. Для этого предназначены методы **StartsWith** и **EndsWith**. Например, в массиве строк хранится список файлов, и нам надо вывести все файлы с расширением exe:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var files = new string[]  {  "myapp.exe",  "forest.jpg",  "main.exe",  "book.pdf",  "river.png"  };    for (int i = 0; i < files.Length; i++)  {  if (files[i].EndsWith(".exe"))  Console.WriteLine(files[i]);  } |

### **Разделение строк**

С помощью функции **Split** мы можем разделить строку на массив подстрок. В качестве параметра функция Split принимает массив символов или строк, которые и будут служить разделителями. Например, подсчитаем количество слов в сроке, разделив ее по пробельным символам:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | string text = "И поэтому все так произошло";    string[] words = text.Split(new char[] { ' ' });    foreach (string s in words)  {  Console.WriteLine(s);  } |

Это не лучший способ разделения по пробелам, так как во входной строке у нас могло бы быть несколько подряд идущих пробелов и в итоговый массив также бы попадали пробелы, поэтому лучше использовать другую версию метода:

|  |  |
| --- | --- |
| 1 | string[] words = text.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries); |

Второй параметр StringSplitOptions.RemoveEmptyEntries говорит, что надо удалить все пустые подстроки.

### **Обрезка строки**

Для обрезки начальных или концевых символов используется функция **Trim**:

|  |  |
| --- | --- |
| 1  2  3  4 | string text = " hello world ";    text = text.Trim(); // результат "hello world"  text = text.Trim(new char[] { 'd', 'h' }); // результат "ello worl" |

Функция Trim без параметров обрезает начальные и конечные пробелы и возвращает обрезанную строку. Чтобы явным образом указать, какие начальные и конечные символы следует обрезать, мы можем передать в функцию массив этих символов.

Эта функция имеет частичные аналоги: функция **TrimStart** обрезает начальные символы, а функция **TrimEnd** обрезает конечные символы.

Обрезать определенную часть строки позволяет функция **Substring**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string text = "Хороший день";  // обрезаем начиная с третьего символа  text = text.Substring(2);  // результат "роший день"  Console.WriteLine(text);  // обрезаем сначала до последних двух символов  text = text.Substring(0, text.Length - 2);  // результат "роший де"  Console.WriteLine(text); |

Функция Substring также возвращает обрезанную строку. В качестве параметра первая использованная версия применяет индекс, начиная с которого надо обрезать строку. Вторая версия применяет два параметра - индекс начала обрезки и длину вырезаемой части строки.

### **Вставка**

Для вставки одной строки в другую применяется функция **Insert**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | string text = "Хороший день";  string substring = "замечательный ";    text = text.Insert(8, substring);  Console.WriteLine(text); // Хороший замечательный день |

Первым параметром в функции Insert является индекс, по которому надо вставлять подстроку, а второй параметр - собственно подстрока.

### **Удаление строк**

Удалить часть строки помогает метод **Remove**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | string text = "Хороший день";  // индекс последнего символа  int ind = text.Length - 1;  // вырезаем последний символ  text = text.Remove(ind);  Console.WriteLine(text); // Хороший ден    // вырезаем первые два символа  text = text.Remove(0, 2);  Console.WriteLine(text); // роший ден |

Первая версия метода Remove принимает индекс в строке, начиная с которого надо удалить все символы. Вторая версия принимает еще один параметр - сколько символов надо удалить.

### **Замена**

Чтобы заменить один символ или подстроку на другую, применяется метод **Replace**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | string text = "хороший день";    text = text.Replace("хороший", "плохой");  Console.WriteLine(text); // плохой день    text = text.Replace("о", "");  Console.WriteLine(text); // плхй день |

Во втором случае применения функции Replace строка из одного символа "о" заменяется на пустую строку, то есть фактически удаляется из текста. Подобным способом легко удалять какой-то определенный текст в строках.

### **Смена регистра**

Для приведения строки к верхнему и нижнему регистру используются соответственно функции **ToUpper()** и **ToLower()**:

|  |  |
| --- | --- |
| 1  2  3  4 | string hello = "Hello world!";    Console.WriteLine(hello.ToLower()); // hello world!  Console.WriteLine(hello.ToUpper()); // HELLO WORLD! |

## **Форматирование и интерполяция строк**

### **Форматирование строк**

При выводе строк в консоли с помощью метода Console.WriteLine для встраивания значений в строку мы можем применять форматирование вместо конкатенации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | string name = "Tom";  int age = 23;    Console.WriteLine("Имя: {0} Возраст: {1}", name, age);  // консольный вывод  // Имя: Tom Возраст: 23 |

В строке "Имя: {0} Возраст: {1}" на место {0} и {1} затем будут вставляться в порядке следования значения переменныйх name и age

То же самое форматирование в строке мы можем сделать не только в методе Console.WriteLine, но и в любом месте программы с помощью метода **string.Format**:

|  |  |
| --- | --- |
| 1  2  3  4 | string name = "Tom";  int age = 23;  string output = string.Format("Имя: {0} Возраст: {1}", name, age);  Console.WriteLine(output); |

Метод Format принимает строку с плейсхолдерами типа {0}, {1} и т.д., а также набор аргументов, которые вставляются на место данных плейсхолдеров. В итоге генерируется новая строка.

### **Спецификаторы форматирования**

В методе Format могут использоваться различные спецификаторы и описатели, которые позволяют настроить вывод данных. Рассмотрим основные описатели. Все используемые форматы:

|  |  |
| --- | --- |
| **C / c** | Задает формат денежной единицы, указывает количество десятичных разрядов после запятой |
| **D / d** | Целочисленный формат, указывает минимальное количество цифр |
| **E / e** | Экспоненциальное представление числа, указывает количество десятичных разрядов после запятой |
| **F / f** | Формат дробных чисел с фиксированной точкой, указывает количество десятичных разрядов после запятой |
| **G / g** | Задает более короткий из двух форматов: F или E |
| **N / n** | Также задает формат дробных чисел с фиксированной точкой, определяет количество разрядов после запятой |
| **P / p** | Задает отображения знака процентов рядом с число, указывает количество десятичных разрядов после запятой |
| **X / x** | Шестнадцатеричный формат числа |

#### **Форматирование валюты**

Для форматирования валюты используется описатель "C":

|  |  |
| --- | --- |
| 1  2  3  4  5 | double number = 23.7;  string result = string.Format("{0:C0}", number);  Console.WriteLine(result); // 24 р.  string result2 = string.Format("{0:C2}", number);  Console.WriteLine(result2); // 23,70 р. |

Число после описателя указывает, сколько чисел будет использоваться после разделителя между целой и дробной частью. При выводе также добавляется обозначение денежного знака для текущей культуры компьютера. В зависимости от локализации текущей операционной системы результат может различаться. Также обратите внимание на округление в первом примере.

#### **Форматирование целых чисел**

Для форматирования целочисленных значение применяется описатель "d":

|  |  |
| --- | --- |
| 1  2  3  4  5 | int number = 23;  string result = string.Format("{0:d}", number);  Console.WriteLine(result); // 23  string result2 = string.Format("{0:d4}", number);  Console.WriteLine(result2); // 0023 |

Число после описателя указывает, сколько цифр будет в числовом значении. Если в исходном числе цифр меньше, то к нему добавляются нули.

#### **Форматирование дробных чисел**

Для форматирования дробны чисел используется описатель F, число после которого указывает, сколько знаков будет использоваться после разделителя между целой и дробной частью. Если исходное число - целое, то к нему добавляются разделитель и нули.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int number = 23;  string result = string.Format("{0:f}", number);  Console.WriteLine(result); // 23,00    double number2 = 45.08;  string result2 = string.Format("{0:f4}", number2);  Console.WriteLine(result2); // 45,0800    double number3 = 25.07;  string result3 = string.Format("{0:f1}", number3);  Console.WriteLine(result3); // 25,1 |

#### **Формат процентов**

Описатель "P" задает отображение процентов. Используемый с ним числовой спецификатор указывает, сколько знаков будет после запятой:

|  |  |
| --- | --- |
| 1  2 | decimal number = 0.15345m;  Console.WriteLine("{0:P1}", number);// 15,3% |

#### **Настраиваемые форматы**

Используя знак #, можно настроить формат вывода. Например, нам надо вывести некоторое число в формате телефона +х (ххх)ххх-хх-хх:

|  |  |
| --- | --- |
| 1  2  3 | long number = 19876543210;  string result = string.Format("{0:+# (###) ###-##-##}", number);  Console.WriteLine(result); // +1 (987) 654-32-10 |

### **Метод ToString**

Метод ToString() не только получает строковое описание объекта, но и может осуществлять форматирование. Он поддерживает те же описатели, что используются в методе Format:

|  |  |
| --- | --- |
| 1  2  3  4  5 | long number = 19876543210;  Console.WriteLine(number.ToString("+# (###) ###-##-##"));// +1 (987) 654-32-10    double money = 24.8;  Console.WriteLine(money.ToString("C2")); // 24,80 р. |

### **Интерполяция строк**

Интерполяция строк призвана упростить форматирование строк. Так, перепишем пример с выводом значений переменных в строке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | string name = "Tom";  int age = 23;    Console.WriteLine($"Имя: {name} Возраст: {age}");  // консольный вывод  // Имя: Tom Возраст: 23 |

Знак доллара перед строкой указывает, что будет осуществляться интерполяция строк. Внутри строки опять же используются плейсхолдеры {...}, только внутри фигурных скобок уже можно напрямую писать те выражения, которые мы хотим вывести.

Интерполяция по сути представляет более лаконичное форматирование. При этом внутри фигурных скобок мы можем указывать не только свойства, но и различные выражения языка C#:

|  |  |
| --- | --- |
| 1  2  3  4 | int x = 8;  int y = 7;  string result = $"{x} + {y} = {x + y}";  Console.WriteLine(result); // 8 + 7 = 15 |

Также внутри фигурных скобок можно выполнять более сложные выражения, например, вызывать методы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int x = 8;  int y = 7;  string result = $"{x} \* {y} = {Multiply(x, y)}";  Console.WriteLine(result); // 8 \* 7 = 56    int Multiply(int a, int b) => a \* b; |

Уже внутри строки можно применять форматирование. В этом случае мы можем применять все те же описатели, что и в методе Format. Например, выведем номер телефона в формате +x xxx-xxx-xx-xx:

|  |  |
| --- | --- |
| 1  2 | long number = 19876543210;  Console.WriteLine($"{number:+# ### ### ## ##}"); // +1 987 654 32 10 |

Добавляем пространство до и после форматируемого вывода:

|  |  |
| --- | --- |
| 1  2  3  4  5 | string name = "Tom";  int age = 23;    Console.WriteLine($"Имя: {name, -5} Возраст: {age}"); // пробелы после  Console.WriteLine($"Имя: {name, 5} Возраст: {age}"); // пробелы до |

Вывод:

Имя: Том Возраст: 23

Имя: Том Возраст: 23

## **Класс StringBuilder**

Хотя класс System.String предоставляет нам широкую функциональность по работе со строками, все таки он имеет свои недостатки. Прежде всего, объект String представляет собой неизменяемую строку. Когда мы выполняем какой-нибудь метод класса String, система создает новый объект в памяти с выделением ему достаточного места. Удаление первого символа - не самая затратная операция. Однако когда подобных операций множество, а объем текста, для которого надо выполнить данные операции, также не самый маленький, то издержки при потере производительности становятся более существенными.

Чтобы выйти из этой ситуации во фреймворк .NET был добавлен новый класс **StringBuilder**, который находится в пространстве имен System.Text. Этот класс представляет динамическую строку.

### **Создание StringBuilder**

Для создания объекта StringBuilder можно использовать ряд его конструкторов. Прежде всего можно создать пустой StringBuilder:

|  |  |
| --- | --- |
| 1  2  3 | using System.Text;    StringBuilder sb = new StringBuilder(); |

Можно сразу инициализировать объект определенной строкой:

|  |  |
| --- | --- |
| 1 | StringBuilder sb = new StringBuilder("Привет мир"); |

С помощью метода ToString() мы можем получить строку, которая хранится в StringBuilder:

|  |  |
| --- | --- |
| 1  2 | var sb = new StringBuilder("Hello World");  Console.WriteLine(sb.ToString()); // Hello World |

Либо можно просто передать объект StringBuilder:

|  |  |
| --- | --- |
| 1  2 | var sb = new StringBuilder("Hello World");  Console.WriteLine(sb); // Hello World |

### **Длина и емкость StringBuilder**

Для хранения длины строки в классе StringBuilder определенно свойство **Length**. Однако есть и вторая величина - емкость выделенной памяти. Это значение хранится в свойстве Capacity. Емкость - это выделенная память под объект. Установка емкости позволяет уменьшить выделения памяти и тем самым повысить производительность.

Если строка, которая передается в конструктор StringBuilder, имеет длину 16 символов или меньше, то начальная ёмкость в StringBuilder равна 16. Если начальная строка больше 16 символов, то в качестве начальной емкости StringBuilder будет использовать длину строки.

Например, посмотрим, что содержат данные свойства:

|  |  |
| --- | --- |
| 1  2  3  4  5 | using System.Text;    StringBuilder sb = new StringBuilder("Привет мир");  Console.WriteLine($"Длина: {sb.Length}"); // Длина: 10  Console.WriteLine($"Емкость: {sb.Capacity}"); // Емкость: 16 |

Хотя в данном случае длина равна 10 символов, но реально емкость будет составлять по умолчанию 16 символов. То есть мы видим, что при создании строки StringBuilder выделяет памяти больше, чем необходимо этой строке. При увеличении строки в StringBuilder, когда количество символов превосходит начальную емкость, то емкость увеличивается в два и более раз.

Если у нас заранее известен максимальный размер объекта, то мы можем таким образом сразу задать емкость с помощью одного из конструкторов и тем самым избежать последующих издержек при дополнительном выделении памяти.

|  |  |
| --- | --- |
| 1 | var sb = new StringBuilder(32); |

StringBuilder также позволяет сразу задать строку и емкость:

|  |  |
| --- | --- |
| 1 | var sb = new StringBuilder("Привет мир", 32); |

### **Операции со строками в StringBuilder**

Для операций над строками класс StringBuilder определяет ряд методов:

* **Append**: добавляет подстроку в объект StringBuilder
* **Insert**: вставляет подстроку в объект StringBuilder, начиная с определенного индекса
* **Remove**: удаляет определенное количество символов, начиная с определенного индекса
* **Replace**: заменяет все вхождения определенного символа или подстроки на другой символ или подстроку
* **AppendFormat**: добавляет подстроку в конец объекта StringBuilder

Теперь посмотрим на примере метода **Append()** использование и преимущества класса StringBuilder:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | using System.Text;    var sb = new StringBuilder("Название: ");  Console.WriteLine(sb); // Название:  Console.WriteLine($"Длина: {sb.Length}"); // 10  Console.WriteLine($"Емкость: {sb.Capacity}"); // 16    sb.Append(" Руководство");  Console.WriteLine(sb); // Название: Руководство  Console.WriteLine($"Длина: {sb.Length}"); // 22  Console.WriteLine($"Емкость: {sb.Capacity}"); // 32    sb.Append(" по C#");  Console.WriteLine(sb); // Название: Руководство по C#  Console.WriteLine($"Длина: {sb.Length}"); // 28  Console.WriteLine($"Емкость: {sb.Capacity}"); // 32 |

При создании объекта StringBuilder выделяется память по умолчанию для 16 символов, так как длина начальной строки меньше 16.

Дальше применяется метод Append - этот метод добавляет к строке подстроку. Так как при объединении строк их общая длина - 22 символа - превышает начальную емкость в 16 символов, то начальная емкость удваивается - до 32 символов.

Если бы итоговая длина строки была бы больше 32 символов, то емкость расширялась бы до размера длины строки.

Далее опять применяется метод Append, однако финальная длина уже будет 28 символов, что меньше 32 символов, и дополнительная память не будет выделяться.

Используем остальные методы StringBuilder:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | var sb = new StringBuilder("Привет мир");  sb.Append("!");  sb.Insert(7, "компьютерный ");  Console.WriteLine(sb); // Привет компьютерный мир!    // заменяем слово  sb.Replace("мир", "world");  Console.WriteLine(sb); // Привет компьютерный world!    // удаляем 13 символов, начиная с 7-го  sb.Remove(7, 13);  Console.WriteLine(sb); // Привет world!    // получаем строку из объекта StringBuilder  string text = sb.ToString();  Console.WriteLine(text); // Привет world! |

Когда надо использовать класс String, а когда StringBuilder?

Microsoft рекомендует использовать класс String в следующих случаях:

* При небольшом количестве операций и изменений над строками
* При выполнении фиксированного количества операций объединения. В этом случае компилятор может объединить все операции объединения в одну
* Когда надо выполнять масштабные операции поиска при построении строки, например IndexOf или StartsWith. Класс StringBuilder не имеет подобных методов.

Класс StringBuilder рекомендуется использовать в следующих случаях:

* При неизвестном количестве операций и изменений над строками во время выполнения программы
* Когда предполагается, что приложению придется сделать множество подобных операций

## **Регулярные выражения**

Классы StringBuilder и String предоставляют достаточную функциональность для работы со строками. Однако .NET предлагает еще один мощный инструмент - регулярные выражения. Регулярные выражения представляют эффективный и гибкий метод по обработке больших текстов, позволяя в то же время существенно уменьшить объемы кода по сравнению с использованием стандартных операций со строками.

Основная функциональность регулярных выражений в .NET сосредоточена в пространстве имен **System.Text.RegularExpressions**. А центральным классом при работе с регулярными выражениями является класс **Regex**. Например, у нас есть некоторый текст и нам надо найти в нем все словоформы какого-нибудь слова. С классом Regex это сделать очень просто:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using System.Text.RegularExpressions;    string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";  Regex regex = new Regex(@"туп(\w\*)");  MatchCollection matches = regex.Matches(s);  if (matches.Count > 0)  {  foreach (Match match in matches)  Console.WriteLine(match.Value);  }  else  {  Console.WriteLine("Совпадений не найдено");  } |

Здесь мы находим в искомой строке все словоформы слова "туп". В конструктор объекта Regex передается регулярное выражение для поиска. Далее мы разберем некоторые элементы синтаксиса регулярных выражений, а пока достаточно знать, что выражение туп(\w\*) обозначает, найти все слова, которые имеют корень "туп" и после которого может стоять различное количество символов. Выражение \w означает алфавитно-цифровой символ, а звездочка после выражения указывает на неопределенное их количество - их может быть один, два, три или вообще не быть.

Метод Matches класса Regex принимает строку, к которой надо применить регулярные выражения, и возвращает коллекцию найденных совпадений.

Каждый элемент такой коллекции представляет объект **Match**. Его свойство Value возвращает найденное совпадение.

И в данном случае мы получим следующий консольный вывод

тупогуб

тупогубенький

тупа

### **Параметр RegexOptions**

Класс Regex имеет ряд конструкторов, позволяющих выполнить начальную инициализацию объекта. Две версии конструкторов в качестве одного из параметров принимают перечисление RegexOptions. Некоторые из значений, принимаемых данным перечислением:

* **Compiled**: при установке этого значения регулярное выражение компилируется в сборку, что обеспечивает более быстрое выполнение
* **CultureInvariant**: при установке этого значения будут игнорироваться региональные различия
* **IgnoreCase**: при установке этого значения будет игнорироваться регистр
* **IgnorePatternWhitespace**: удаляет из строки пробелы и разрешает комментарии, начинающиеся со знака #
* **Multiline**: указывает, что текст надо рассматривать в многострочном режиме. При таком режиме символы "^" и "$" совпадают, соответственно, с началом и концом любой строки, а не с началом и концом всего текста
* **RightToLeft**: приписывает читать строку справа налево
* **Singleline**: при данном режиме символ "." соответствует любому символу, в том числе последовательности "\n", которая осуществляет переход на следующую строку

Например:

|  |  |
| --- | --- |
| 1 | Regex regex = new Regex(@"туп(\w\*)", RegexOptions.IgnoreCase); |

При необходимости можно установить несколько параметров:

|  |  |
| --- | --- |
| 1 | Regex regex = new Regex(@"туп(\w\*)", RegexOptions.Compiled | RegexOptions.IgnoreCase); |

### **Синтаксис регулярных выражений**

Рассмотрим вкратце некоторые элементы синтаксиса регулярных выражений:

* **^**: соответствие должно начинаться в начале строки (например, выражение @"^пр\w\*" соответствует слову "привет" в строке "привет мир")
* **$**: конец строки (например, выражение @"\w\*ир$" соответствует слову "мир" в строке "привет мир", так как часть "ир" находится в самом конце)
* **.**: знак точки определяет любой одиночный символ (например, выражение "м.р" соответствует слову "мир" или "мор")
* **\***: предыдущий символ повторяется 0 и более раз
* **+**: предыдущий символ повторяется 1 и более раз
* **?**: предыдущий символ повторяется 0 или 1 раз
* **\s**: соответствует любому пробельному символу
* **\S**: соответствует любому символу, не являющемуся пробелом
* **\w**: соответствует любому алфавитно-цифровому символу
* **\W**: соответствует любому не алфавитно-цифровому символу
* **\d**: соответствует любой десятичной цифре
* **\D** : соответствует любому символу, не являющемуся десятичной цифрой

Это только небольшая часть элементов. Более подробное описание синтаксиса регулярных выражений можно найти на msdn в статье [Элементы языка регулярных выражений — краткий справочник](https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference).

Теперь посмотрим на некоторые примеры использования. Возьмем первый пример с скороговоркой "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа" и найдем в ней все слова, где встречается корень "губ":

|  |  |
| --- | --- |
| 1  2 | string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";  Regex regex = new Regex(@"\w\*губ\w\*"); |

Так как выражение \w\* соответствует любой последовательности алфавитно-цифровых символов любой длины, то данное выражение найдет все слова, содержащие корень "губ".

Второй простенький пример - нахождение телефонного номера в формате 111-111-1111:

|  |  |
| --- | --- |
| 1  2 | string s = "456-435-2318";  Regex regex = new Regex(@"\d{3}-\d{3}-\d{4}"); |

Если мы точно знаем, сколько определенных символов должно быть, то мы можем явным образом указать их количество в фигурных скобках: \d{3} - то есть в данном случае три цифры.

Мы можем не только задать поиск по определенным типам символов - пробелы, цифры, но и задать конкретные символы, которые должны входить в регулярное выражение. Например, перепишем пример с номером телефона и явно укажем, какие символы там должны быть:

|  |  |
| --- | --- |
| 1  2 | string s = "456-435-2318";  Regex regex = new Regex("[0-9]{3}-[0-9]{3}-[0-9]{4}"); |

В квадратных скобках задается диапазон символов, которые должны в данном месте встречаться. В итоге данный и предыдущий шаблоны телефонного номера будут эквивалентны.

Также можно задать диапазон для алфавитных символов: Regex regex = new Regex("[a-v]{5}"); - данное выражение будет соответствовать любому сочетанию пяти символов, в котором все символы находятся в диапазоне от a до v.

Можно также указать отдельные значения: Regex regex = new Regex(@"[2]\*-[0-9]{3}-\d{4}");. Это выражение будет соответствовать, например, такому номеру телефона "222-222-2222" (так как первые числа двойки)

С помощью операции | можно задать альтернативные символы, например:

|  |  |
| --- | --- |
| 1 | Regex regex = new Regex(@"(2|3){3}-[0-9]{3}-\d{4}"); |

То есть первые три цифры могут содержать только двойки или тройки. Такой шаблон будет соответствовать, например, строкам "222-222-2222" и "323-435-2318". А вот строка "235-435-2318" уже не подпадает под шаблон, так как одной из трех первых цифр является цифра 5.

Итак, у нас такие символы, как \*, + и ряд других используются в качестве специальных символов. И возникает вопрос, а что делать, если нам надо найти, строки, где содержится точка, звездочка или какой-то другой специальный символ? В этом случае нам надо просто экранировать эти символы слешем:

|  |  |
| --- | --- |
| 1  2 | Regex regex = new Regex(@"(2|3){3}\.[0-9]{3}\.\d{4}");  // этому выражению будет соответствовать строка "222.222.2222" |

### **Проверка на соответствие строки формату**

Нередко возникает задача проверить корректность данных, введенных пользователем. Это может быть проверка электронного адреса, номера телефона, Класс Regex предоставляет статический метод **IsMatch**, который позволяет проверить входную строку с шаблоном на соответствие:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using System.Text.RegularExpressions;    string pattern = @"^(?("")(""[^""]+?""@)|(([0-9a-z]((\.(?!\.))|[-!#\$%&'\\*\+/=\?\^`\{\}\|~\w])\*)(?<=[0-9a-z])@))" +  @"(?(\[)(\[(\d{1,3}\.){3}\d{1,3}\])|(([0-9a-z][-\w]\*[0-9a-z]\*\.)+[a-z0-9]{2,17}))$";  var data = new string[]  {  "tom@gmail.com",  "+12345678999",  "bob@yahoo.com",  "+13435465566",  "sam@yandex.ru",  "+43743989393"  };    Console.WriteLine("Email List");  for(int i = 0; i < data.Length; i++)  {  if (Regex.IsMatch(data[i], pattern, RegexOptions.IgnoreCase))  {  Console.WriteLine(data[i]);  }  } |

Переменная pattern задает регулярное выражение для проверки адреса электронной почты. Данное выражение предлагает нам Microsoft на страницах msdn.

Далее в цикле мы проходим по массиву строк и определяем, какие строки соответствуют этому шаблону, то есть представляют валидный адрес электронной почты. Для проверки соответствия строки шаблону используется метод IsMatch: Regex.IsMatch(data[i], pattern, RegexOptions.IgnoreCase). Последний параметр указывает, что регистр можно игнорировать. И если строка соответствует шаблону, то метод возвращает true.

### **Замена и метод Replace**

Класс Regex имеет метод Replace, который позволяет заменить строку, соответствующую регулярному выражению, другой строкой:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | string text = "Мама мыла раму. ";  string pattern = @"\s+";  string target = " ";  Regex regex = new Regex(pattern);  string result = regex.Replace(text, target);  Console.WriteLine(result); |

Данная версия метода Replace принимает два параметра: строку с текстом, где надо выполнить замену, и сама строка замены. Так как в качестве шаблона выбрано выражение "\s+ (то есть наличие одного и более пробелов), метод Replace проходит по всему тексту и заменяет несколько подряд идущих пробелов ординарными.

Другой пример - на вход подается номер телефона в произвольном формате, и мы хотим оставить в нем только цифры:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | string phoneNumber = "+1(876)-234-12-98";  string pattern = @"\D";  string target = "";  Regex regex = new Regex(pattern);  string result = regex.Replace(phoneNumber, target);  Console.WriteLine(result); // 18762341298 |

В данном случае шаблон "\D" представляет любой символ, который не является цифрой. Любой такой символ заменяется на пустую строку "", то есть в итоге из строки "+1(876)-234-12-98" мы получим строку "18762341298".

# **Работа с датами и временем**

## **Структура DateTime**

Для работы с датами и временем в .NET предназначена структура **DateTime**. Она представляет дату и время от 00:00:00 1 января 0001 года до 23:59:59 31 декабря 9999 года.

Для создания нового объекта DateTime также можно использовать конструктор. Пустой конструктор создает начальную дату:

|  |  |
| --- | --- |
| 1  2 | DateTime dateTime = new DateTime();  Console.WriteLine(dateTime); // 01.01.0001 0:00:00 |

То есть мы получим минимально возможное значение, которое также можно получить следующим образом:

|  |  |
| --- | --- |
| 1 | Console.WriteLine(DateTime.MinValue); |

Чтобы задать конкретную дату, нужно использовать один из конструкторов, принимающих параметры:

|  |  |
| --- | --- |
| 1  2 | DateTime date1 = new DateTime(2015, 7, 20); // год - месяц - день  Console.WriteLine(date1); // 20.07.2015 0:00:00 |

Установка времени:

|  |  |
| --- | --- |
| 1  2 | DateTime date1 = new DateTime(2015, 7, 20, 18, 30, 25); // год - месяц - день - час - минута - секунда  Console.WriteLine(date1); // 20.07.2015 18:30:25 |

Если необходимо получить текущую время и дату, то можно использовать ряд свойств DateTime:

|  |  |
| --- | --- |
| 1  2  3 | Console.WriteLine(DateTime.Now);  Console.WriteLine(DateTime.UtcNow);  Console.WriteLine(DateTime.Today); |

Консольный вывод:

20.07.2015 11:43:33

20.07.2015 8:43:33

20.07.2015 0:00:00

Свойство DateTime.Now берет текущую дату и время компьютера, DateTime.UtcNow - дата и время относительно времени по Гринвичу (GMT) и DateTime.Today - только текущая дата.

При работе с датами надо учитывать, что по умолчанию для представления дат применяется григорианский календарь. Но что будет, если мы захотим получить день недели для 5 октября 1582 года:

|  |  |
| --- | --- |
| 1  2 | DateTime someDate = new DateTime(1582, 10, 5);  Console.WriteLine(someDate.DayOfWeek); |

Консоль высветит значение Tuesday, то есть вторник. Однако, как может быть известно из истории, впервые переход с юлианского календаря на григорианский состоялся в октябре 1582 года. Тогда после даты 4 октября (четверг) (еще по юлианскому календарю) сразу перешли к 15 октября (пятница)(уже по григорианскому календарю). Таким образом, фактически выкинули 10 дней. То есть после 4 октября шло 15 октября.

В большинстве случаев данный факт вряд ли как-то повлияет на вычисления, однако при работе с очень давними датами данный аспект следует учитывать.

### **Операции с DateTime**

Основные операции со структурой DateTime связаны со сложением или вычитанием дат. Например, надо к некоторой дате прибавить или, наоборот, отнять несколько дней.

Для добавления дат используется ряд методов:

* Add(TimeSpan value): добавляет к дате значение TimeSpan
* AddDays(double value): добавляет к текущей дате несколько дней
* AddHours(double value): добавляет к текущей дате несколько часов
* AddMinutes(double value): добавляет к текущей дате несколько минут
* AddMonths(int value): добавляет к текущей дате несколько месяцев
* AddYears(int value): добавляет к текущей дате несколько лет

Например, добавим к некоторой дате 3 часа:

|  |  |
| --- | --- |
| 1  2 | DateTime date1 = new DateTime(2015, 7, 20, 18, 30, 25); // 20.07.2015 18:30:25  Console.WriteLine(date1.AddHours(3)); // 20.07.2015 21:30:25 |

Для вычитания дат используется метод **Subtract(DateTime date)**:

|  |  |
| --- | --- |
| 1  2  3 | DateTime date1 = new DateTime(2015, 7, 20, 18, 30, 25); // 20.07.2015 18:30:25  DateTime date2 = new DateTime(2015, 7, 20, 15, 30, 25); // 20.07.2015 15:30:25  Console.WriteLine(date1.Subtract(date2)); // 03:00:00 |

Здесь даты различаются на три часа, поэтому результатом будет дата "03:00:00".

Метод Substract не имеет возможностей для отдельного вычитания дней, часов и так далее. Но это и не надо, так как мы можем передавать в метод AddDays() и другие методы добавления отрицательные значения:

|  |  |
| --- | --- |
| 1  2  3 | // вычтем три часа  DateTime date1 = new DateTime(2015, 7, 20, 18, 30, 25); // 20.07.2015 18:30:25  Console.WriteLine(date1.AddHours(-3)); // 20.07.2015 15:30:25 |

Кроме операций сложения и вычитания еще есть ряд методов форматирования дат:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | DateTime date1 = new DateTime(2015, 7, 20, 18, 30, 25);  Console.WriteLine(date1.ToLocalTime()); // 20.07.2015 21:30:25  Console.WriteLine(date1.ToUniversalTime()); // 20.07.2015 15:30:25  Console.WriteLine(date1.ToLongDateString()); // 20 июля 2015 г.  Console.WriteLine(date1.ToShortDateString()); // 20.07.2015  Console.WriteLine(date1.ToLongTimeString()); // 18:30:25  Console.WriteLine(date1.ToShortTimeString()); // 18:30 |

Метод ToLocalTime() преобразует время UTC в локальное время, добавляя смещение относительно времени по Гринвичу. Метод ToUniversalTime(), наоборот, преобразует локальное время во время UTC, то есть вычитает смещение относительно времени по Гринвичу. Остальные методы преобразуют дату к определенному формату.

## **Форматирование дат и времени**

Для форматирования вывода дат и времени применяется ряд строковых форматов:

|  |  |
| --- | --- |
| Описатель | Описание |
| D | Полный формат даты. Например, 17 июля 2015 г. |
| d | Краткий формат даты. Например, 17.07.2015 |
| F | Полный формат даты и времени. Например, 17 июля 2015 г. 17:04:43 |
| f | Полный формат даты и краткий формат времени. Например, 17 июля 2015 г. 17:04 |
| G | Краткий формат даты и полный формат времени. Например, 17.07.2015 17:04:43 |
| g | Краткий формат даты и времени. Например, 17.07.2015 17:04 |
| M, m | Формат дней месяца. Например, 17 июля |
| O, o | Формат обратного преобразования даты и времени. Вывод даты и времени в соответствии со стандартом ISO 8601 в формате "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffzzz". Например, 2015-07-17T17:04:43.4092892+03:00 |
| R, r | Время по Гринвичу. Например, Fri, 17 Jul 2015 17:04:43 GMT |
| s | Сортируемый формат даты и времени. Например, 2015-07-17T17:04:43 |
| T | Полный формат времени. Например, 17:04:43 |
| t | Краткий формат времени. Например, 17:04 |
| U | Полный универсальный полный формат даты и времени. Например, 17 июля 2015 г. 17:04:43 |
| u | Краткий универсальный полный формат даты и времени. Например, 2015-07-17 17:04:43Z |
| Y, y | Формат года. Например, Июль 2015 |

Выведем текущею дату и время во всех форматах:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | DateTime now = DateTime.Now;    Console.WriteLine($"D: {now.ToString("D")}");  Console.WriteLine($"d: {now.ToString("d")}");  Console.WriteLine($"F: {now.ToString("F")}");  Console.WriteLine($"f: {now:f}");  Console.WriteLine($"G: {now:G}");  Console.WriteLine($"g: {now:g}");  Console.WriteLine($"M: {now:M}");  Console.WriteLine($"O: {now:O}");  Console.WriteLine($"o: {now:o}");  Console.WriteLine($"R: {now:R}");  Console.WriteLine($"s: {now:s}");  Console.WriteLine($"T: {now:T}");  Console.WriteLine($"t: {now:t}");  Console.WriteLine($"U: {now:U}");  Console.WriteLine($"u: {now:u}");  Console.WriteLine($"Y: {now:Y}"); |

Консольный вывод:

D: 6 января 2022 г.

d: 06.01.2022

F: 6 января 2022 г. 14:45:20

f: 6 января 2022 г. 14:45

G: 06.01.2022 14:45:20

g: 06.01.2022 14:45

M: 6 января

O: 2022-01-06T14:45:20.3942344+04:00

o: 2022-01-06T14:45:20.3942344+04:00

R: Thu, 06 Jan 2022 14:45:20 GMT

s: 2022-01-06T14:45:20

T: 14:45:20

t: 14:45

U: 6 января 2022 г. 10:45:20

u: 2022-01-06 14:45:20Z

Y: январь 2022 г.

### **Настройка формата времени и даты**

Не всегда удобно использование встроенных форматов даты и времени. Иногда бывает необходимо задать сообственную форму отображения объекта DateTime. В этом случае мы можем составить свой формат из описателей:

|  |  |
| --- | --- |
| Описатель | Описание |
| d | Представляет день месяца от 1 до 31. Одноразрядные числа используются без нуля в начале |
| dd | Представляет день месяца от 1 до 31. К одноразрядным числам в начале добавляется ноль |
| ddd | Сокращенное название дня недели |
| dddd | Полное название дня недели |
| f / fffffff | Представляет миллисекунды. Количество символов f указывает на число разрядов в миллисекундах |
| g | Представляет период или эру (например, "н. э.") |
| h | Часы в виде от 1 до 12. Часы с одной цифрой не дополняются нулем |
| hh | Часы в виде от 01 до 12. Часы с одной цифрой дополняются нулем |
| H | Часы в виде от 0 до 23. Часы с одной цифрой не дополняются нулем |
| HH | Часы в виде от 0 до 23. Часы с одной цифрой дополняются нулем |
| K | Часовой пояс |
| m | Минуты от 0 до 59. Минуты с одной цифрой не дополняются начальным нулем |
| mm | Минуты от 0 до 59. Минуты с одной цифрой дополняются начальным нулем |
| M | Месяц в виде от 1 до 12 |
| MM | Месяц в виде от 1 до 12. Месяц с одной цифрой дополняется начальным нулем |
| MMM | Сокращенное название месяца |
| MMMM | Полное название месяца |
| s | Секунды в виде числа от 0 до 59. Секунды с одной цифрой не дополняются начальным нулем |
| ss | Секунды в виде числа от 0 до 59. Секунды с одной цифрой дополняются начальным нулем |
| t | Первые символы в обозначениях AM и PM |
| tt | AM или PM |
| y | Представляет год как число из одной или двух цифр. Если год имеет более двух цифр, то в результате отображаются только две младшие цифры |
| yy | Представляет год как число из одной или двух цифр. Если год имеет более двух цифр, то в результате отображаются только две младшие цифры. Если год имеет одну цифру, то он дополняется начальным нулем |
| yyy | Год из трех цифр |
| yyyy | Год из четырех цифр |
| yyyyy | Год из пяти цифр. Если в году меньше пяти цифр, то он дополняется начальными нулями |
| z | Представляет смещение в часах относительно времени UTC |
| zz | Представляет смещение в часах относительно времени UTC. Если смещение представляет одну цифру, то она дополняется начальным нулем. |

Создадим пару своих форматов:

|  |  |
| --- | --- |
| 1  2  3 | DateTime now = DateTime.Now;  Console.WriteLine(now.ToString("hh:mm:ss"));  Console.WriteLine(now.ToString("dd.MM.yyyy")); |

Консольный вывод:

02:46:38

06.01.2022

## **DateOnly и TimeOnly**

Для упрощения работы с датами и временем в .NET 6 были добавлены две дополнительные структуры - **DateOnly** и **TimeOnly**.

### **DateOnly**

Структура **DateOnly** представляет дату. Для создания структуры можно использовать ряд ее конструкторов.

|  |  |
| --- | --- |
| 1  2  3 | DateOnly()  DateOnly(int year, int month, int day)  DateOnly(int year, int month, int day, System.Globalization.Calendar calendar) |

При использовании конструктора без параметров структура инициализируется датой 01.01.0001:

|  |  |
| --- | --- |
| 1  2 | DateOnly someDate = new DateOnly();  Console.WriteLine(someDate); // 01.01.0001 |

Вторая версия конструктора принимает год, месяц и число, которыми инициализируется структура:

|  |  |
| --- | --- |
| 1  2 | DateOnly someDate = new DateOnly(2022,1,6); // 6 января 2022 года  Console.WriteLine(someDate); // 06.01.2022 |

Третья версия конструктора в дополнение к году, месяцу и числу, также принимает объект календаря - объект **System.Globalization.Calendar**, относительно которого будет расcчитываться дата. Класс Calendar является абстрактным, но .NET предоставляет ряд встроенных типов календарей. Например, расcчитаем дату относительно юлианского календаря:

|  |  |
| --- | --- |
| 1  2  3  4 | using System.Globalization;    DateOnly julianDate = new DateOnly(2022,1,6, new JulianCalendar());  Console.WriteLine(julianDate); // 19.01.2022 |

В данном случае для .NET переданная в конструктор дата - 06.01.2022 расценивается как дата юлианского календаря. При выводе на консоль мы видим тот же день только относительно григорианского календаря.

#### **Свойства DateOnly**

С помощью свойств структуры можно получить отдельные составляющие даты:

* **Day**: возвращает день даты
* **DayNumber**: возвращает количество прошедших дней с 1 января 0001 года относительно григорианского календаря
* **DayOfWeek**: возвращает день недели
* **DayOfYear**: возвращает день года
* **MaxValue**: возвращает максимально возможную дату (статическое свойство)
* **MinValue**: возвращает самую раннюю возможную дату (статическое свойство)
* **Month**: возвращает месяц
* **Year**: возвращает год

Применение свойств:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | DateOnly now = new DateOnly(2022,1,6);  Console.WriteLine(now.Day); // 6  Console.WriteLine(now.DayNumber); // 738160  Console.WriteLine(now.DayOfWeek); // Thursday  Console.WriteLine(now.DayOfYear); // 6  Console.WriteLine(now.Month); // 1  Console.WriteLine(now.Year); // 2022 |

#### **Методы DateOnly**

С помощью методов DateOnly можно производить некоторые операции с датами. Некоторые из них:

* **AddDays(int days)**: добавляет к дате некоторое количество дней
* **AddMonths(int months)**: добавляет к дате некоторое количество месяцев
* **AddYears(int years)**: добавляет к дате некоторое количество лет
* **ToDateTime(TimeOnly)**: возвращает объект DateTime, который в качестве даты исппользует текущий объект DateOnly, а в качестве времени - значение параметра в виде TimeOnly
* **ToLongDateString()**: выводит текущий объект DateOnly в виде подробной даты
* **ToShortDateString()**: выводит текущий объект DateOnly в виде сжатой даты

Также в классе есть ряд статических методов. Некоторые из них:

* **FromDateTime(DateTime dateTime)**: на основе значения DateTime, переданного через параметр, создает и возвращает объект DateOnly
* **FromDayNumber(int days)**: на основе количества дней создает и возвращает объект DateOnly
* **Parse(string date)**: конвертирует строковое представление даты в объект DateOnly
* **ParseExact(string date, string format)**: конвертирует строковое представление даты в объект DateOnly, применяя определенный формат
* **TryParse(String, DateOnly)**: конвертирует строковое представление даты в объект DateOnly. При успешной конвертации возвращает true, а параметр типа DateOnly содержит созданную дату
* **TryParseExact(String, String, DateOnly)**: конвертирует строковое представление даты в объект DateOnly, применяя определенный формат. При успешной конвертации возвращает true, а параметр типа DateOnly содержит созданную дату

Пример некоторых операций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | DateOnly now = DateOnly.Parse("06.01.2022"); // на русскоязычной локализованной ОС  Console.WriteLine(now); // 06.01.2022  now = now.AddDays(1); // 07.01.2022  now = now.AddMonths(4); // 07.05.2022  now = now.AddYears(-1); // 07.05.2021  Console.WriteLine(now.ToShortDateString()); // 07.05.2021  Console.WriteLine(now.ToLongDateString()); // 7 мая 2021 г. |

### **TimeOnly**

Структура **TimeOnly** представляет время в диапазоне от 00:00:00 до 23:59:59.9999999. Для создания структуры можно использовать ряд ее конструкторов.

|  |  |
| --- | --- |
| 1  2  3  4  5 | TimeOnly()  TimeOnly(long ticks)  TimeOnly(int hour, int minute)  TimeOnly(int hour, int minute, int second)  TimeOnly(int hour, int minute, int second, int millisecond) |

При использовании конструктора без параметров структура инициализируется временем 0.00:

|  |  |
| --- | --- |
| 1  2 | TimeOnly time = new TimeOnly();  Console.WriteLine(time); // 0:00 |

Дополнительно с помощью других версий конструктора можно установить количество часов, минут, секунд и миллисекунд:

|  |  |
| --- | --- |
| 1  2  3  4  5 | TimeOnly time1 = new TimeOnly(4, 30);  Console.WriteLine(time1); // 4: 30    TimeOnly time2 = new TimeOnly(14, 23, 30);  Console.WriteLine(time2); // 14: 23 |

#### **Свойства TimeOnly**

С помощью свойств структуры можно получить отдельные составляющие времени:

* **Hour**: возвращает количество часов
* **Minute**: возвращает количество минут
* **Second**: возвращает количество секунд
* **Millisecond**: возвращает количество миллисекунд
* **Ticks**: возвращает количество тиков
* **MaxValue**: возвращает максимально возможное время (статическое свойство)
* **MinValue**: возвращает минимально возможное время (статическое свойство)

Применение свойств:

|  |  |
| --- | --- |
| 1  2  3  4 | TimeOnly time = new TimeOnly(14, 23, 30);  Console.WriteLine(time.Hour); // 14  Console.WriteLine(time.Minute); // 23  Console.WriteLine(time.Second); // 30 |

#### **Методы TimeOnly**

С помощью методов TimeOnly можно производить некоторые операции с временем. Некоторые из них:

* **AddHours(double hours)**: добавляет к времени некоторое количество часов
* **AddMinutes(double minutes)**: добавляет к времени некоторое количество минут
* **Add(TimeSpan value)**: добавляет время из объекта TimeSpan
* **ToLongTimeString()**: выводит текущий объект TimeOnly в виде подробного времени
* **ToShortTimeString()**: выводит текущий объект TimeOnly в виде сжатого времени

Также в классе есть ряд статических методов. Некоторые из них:

* **FromDateTime(DateTime dateTime)**: на основе значения DateTime, переданного через параметр, создает и возвращает объект TimeOnly
* **FromTimeSpan(TimeSpan value)**: на основе объекта TimeSpan создает и возвращает объект TimeOnly
* **Parse(string time)**: конвертирует строковое представление времени в объект TimeOnly
* **ParseExact(string timee, string format)**: конвертирует строковое представление времени в объект TimeOnly, применяя определенный формат
* **TryParse(string time, TimeOnly result)**: конвертирует строковое представление времени в объект TimeOnly. При успешной конвертации возвращает true, а параметр типа TimeOnly содержит сконвертированное время
* **TryParseExact(string time, string format, TimeOnly result)**: конвертирует строковое представление времени в объект TimeOnly, применяя определенный формат. При успешной конвертации возвращает true, а параметр типа TimeOnly содержит сконвертированное время

Пример некоторых операций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | TimeOnly time = TimeOnly.Parse("06:33:22");  Console.WriteLine(time); // 6:33  time = time.AddHours(1); // 7:33  time = time.AddMinutes(-23); // 7:10    Console.WriteLine(time.ToShortTimeString()); // 7:10  Console.WriteLine(time.ToLongTimeString()); // 7:10:22 |

# **Дополнительные классы и структуры .NET**

## **Отложенная инициализация и тип Lazy**

Приложение может использовать множество классов и объектов. Однако есть вероятность, что не все создаваемые объекты будут использованы. Особенно это касается больших приложений. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class Reader  {  Library library = new Library();  public void ReadBook()  {  library.GetBook();  Console.WriteLine("Читаем бумажную книгу");  }    public void ReadEbook()  {  Console.WriteLine("Читаем книгу на компьютере");  }  }    class Library  {  private string[] books = new string[99];    public void GetBook()  {  Console.WriteLine("Выдаем книгу читателю");  }  } |

Есть класс Library, представляющий библиотеку и хранящий некоторый набор книг в виде массива. Есть класс читателя Reader, который хранит ссылку на объект библиотеки, в которой он записан. У читателя определено два метода: для чтения электронной книги и для чтения обычной книги. Для чтения обычной книги необходимо обратиться к методу класса Library, чтобы получить эту книгу.

Но что если читателю вообще не придется читать обычную книгу, а только электронные, например, в следующем случае:

|  |  |
| --- | --- |
| 1  2 | Reader reader = new Reader();  reader.ReadEbook(); |

В этом случае объект library в классе читателя никак не будет использоваться и будет только занимать место памяти. Хотя надобности в нем не будет.

Для подобных случаев в .NET определен специальный класс **Lazy<T>**. Изменим класс читателя следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Reader  {  Lazy<Library> library = new Lazy<Library>();  public void ReadBook()  {  library.Value.GetBook();  Console.WriteLine("Читаем бумажную книгу");  }    public void ReadEbook()  {  Console.WriteLine("Читаем книгу на компьютере");  }  } |

Класс Library остается прежнем. Но теперь класс читателя содержит ссылку на библиотеку в виде объекта Lazy<Library>. А чтобы обратиться к самой библиотеке и ее методам, надо использовать выражение library.Value - это и есть объект Library.

Что меняет в поведении класса Reader эта замена? Рассмотрим его применение:

|  |  |
| --- | --- |
| 1  2  3 | Reader reader = new Reader();  reader.ReadEbook();  reader.ReadBook(); |

Непосредственно объект Library задействуется здесь только на третьей строке в методе reader.ReadBook(), который вызывает в свою очередь метод library.Value.GetBook(). Поэтому вплоть до третьей строки объект Library, используемый читателем, не будет создан. Если мы не будем применять в программе метод reader.ReadBook(), то объект библиотеки тогда вообще не будет создан, и мы избежим лишних затрат памяти. Таким образом, Lazy<T> гарантирует нам, что объект будет создан только тогда, когда в нем есть необходимость.

## **Математические вычисления и класс Math**

Для выполнения различных математических операций в библиотеке классов .NET предназначен класс **Math**. Он является статическим, поэтому все его методы также являются статическими.

Рассмотрим основные методы класса Math:

* Abs(double value): возвращает абсолютное значение для аргумента value

|  |  |
| --- | --- |
| 1 | double result = Math.Abs(-12.4); // 12.4 |

* Acos(double value): возвращает арккосинус value. Параметр value должен иметь значение от -1 до 1

|  |  |
| --- | --- |
| 1 | double result = Math.Acos(1); // 0 |

* Asin(double value): возвращает арксинус value. Параметр value должен иметь значение от -1 до 1
* Atan(double value): возвращает арктангенс value
* BigMul(int x, int y): возвращает произведение x \* y в виде объекта long

|  |  |
| --- | --- |
| 1 | double result = Math.BigMul(100, 9340); // 934000 |

* Ceiling(double value): возвращает наименьшее целое число с плавающей точкой, которое не меньше value

|  |  |
| --- | --- |
| 1 | double result = Math.Ceiling(2.34); // 3 |

* Cos(double d): возвращает косинус угла d
* Cosh(double d): возвращает гиперболический косинус угла d
* DivRem(int a, int b, out int result): возвращает результат от деления a/b, а остаток помещается в параметр result

|  |  |
| --- | --- |
| 1  2  3  4 | int result;  int div = Math.DivRem(14, 5, out result);  //result = 4  // div = 2 |

* Exp(double d): возвращает основание натурального логарифма, возведенное в степень d
* Floor(decimal d): возвращает наибольшее целое число, которое не больше d

|  |  |
| --- | --- |
| 1 | double result = Math.Floor(2.56); // 2 |

* IEEERemainder(double a, double b): возвращает остаток от деления a на b

|  |  |
| --- | --- |
| 1 | double result = Math.IEEERemainder(26, 4); // 2 = 26-24 |

* Log(double d): возвращает натуральный логарифм числа d
* Log(double a, double newBase): возвращает логарифм числа a по основанию newBase
* Log10(double d): возвращает десятичный логарифм числа d
* Max(double a, double b): возвращает максимальное число из a и b
* Min(double a, double b): возвращает минимальное число из a и b
* Pow(double a, double b): возвращает число a, возведенное в степень b
* Round(double d): возвращает число d, округленное до ближайшего целого числа

|  |  |
| --- | --- |
| 1  2 | double result1 = Math.Round(20.56); // 21  double result2 = Math.Round(20.46); //20 |

* Round(double a, int b): возвращает число a, округленное до определенного количества знаков после запятой, представленного параметром b

|  |  |
| --- | --- |
| 1  2 | double result1 = Math.Round(20.567, 2); // 20,57  double result2 = Math.Round(20.463, 1); //20,5 |

* Sign(double value): возвращает число 1, если число value положительное, и -1, если значение value отрицательное. Если value равно 0, то возвращает 0

|  |  |
| --- | --- |
| 1  2 | int result1 = Math.Sign(15); // 1  int result2 = Math.Sign(-5); //-1 |

* Sin(double value): возвращает синус угла value
* Sinh(double value): возвращает гиперболический синус угла value
* Sqrt(double value): возвращает квадратный корень числа value

|  |  |
| --- | --- |
| 1 | double result1 = Math.Sqrt(16); // 4 |

* Tan(double value): возвращает тангенс угла value
* Tanh(double value): возвращает гиперболический тангенс угла value
* Truncate(double value): отбрасывает дробную часть числа value, возвращаяя лишь целое значение

|  |  |
| --- | --- |
| 1 | double result = Math.Truncate(16.89); // 16 |

Также класс Math определяет две константы: Math.E и Math.PI. Например, вычислим площадь круга:

|  |  |
| --- | --- |
| 1  2  3  4 | double radius = 50;  double area = Math.PI \* Math.Pow(radius, 2);  Console.WriteLine($"Площадь круга с радиусом {radius} равна {Math.Round(area,2)}");  Площадь круга с радиусом 50 равна 7853,98 |

## **Преобразование типов и класс Convert**

### **Методы Parse и TryParse**

Все примитивные типы имеют два метода, которые позволяют преобразовать строку к данному типу. Это методы **Parse()** и **TryParse()**.

Метод **Parse()** в качестве параметра принимает строку и возвращает объект текущего типа. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int a = int.Parse("10");  double b = double.Parse("23,56");  decimal c = decimal.Parse("12,45");  byte d = byte.Parse("4");  Console.WriteLine($"a={a} b={b} c={c} d={d}"); |

Стоит отметить, что парсинг дробных чисел зависит от настроек текущей культуры. В частности, для получения числа double я передаю строку "23,56" с запятой в качестве разделителя. Если бы я передал точку вместо запятой, то приложение выдало ошибку выполнения. На компьютерах с другой локалью, наоборот, использование запятой вместо точки выдало бы ошибку.

Чтобы не зависеть от культурных различий мы можем установить четкий формат с помощью класса **NumberFormatInfo** и его свойства **NumberDecimalSeparator**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | using System.Globalization;    IFormatProvider formatter = new NumberFormatInfo { NumberDecimalSeparator = "." };  double number = double.Parse("23.56", formatter);  Console.WriteLine(number); // 23,56 |

В данном случае в качестве разделителя устанавливается точка. Однако тем не менее потенциально при использовании метода Parse мы можем столкнуться с ошибкой, например, при передачи алфавитных символов вместо числовых. И в этом случае более удачным выбором будет применение метода **TryParse()**. Он пытается преобразовать строку к типу и, если преобразование прошло успешно, то возвращает **true**. Иначе возвращается false:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Console.WriteLine("Введите строку:");  string? input = Console.ReadLine();    bool result = int.TryParse(input, out var number);  if (result == true)  Console.WriteLine($"Преобразование прошло успешно. Число: {number}");  else  Console.WriteLine("Преобразование завершилось неудачно"); |

Если преобразование пройдет неудачно, то исключения никакого не будет выброшено, просто метод TryParse возвратит false, а переменная number будет содержать значение по умолчанию.

### **Convert**

Класс **Convert** представляет еще один способ для преобразования значений. Для этого в нем определены следующие статические методы:

* **ToBoolean(value)**
* **ToByte(value)**
* **ToChar(value)**
* **ToDateTime(value)**
* **ToDecimal(value)**
* **ToDouble(value)**
* **ToInt16(value)**
* **ToInt32(value)**
* **ToInt64(value)**
* **ToSByte(value)**
* **ToSingle(value)**
* **ToUInt16(value)**
* **ToUInt32(value)**
* **ToUInt64(value)**

В качестве параметра в эти методы может передаваться значение различных примитивных типов, необязательно строки:

|  |  |
| --- | --- |
| 1  2  3  4 | int n = Convert.ToInt32("23");  bool b = true;  double d = Convert.ToDouble(b);  Console.WriteLine($"n={n} d={d}"); |

Однако опять же, как и в случае с методом Parse, если методу не удастся преобразовать значение к нужному типу, то он выбрасывает исключение FormatException.

# **Многопоточность**

## **Введение в многопоточность. Класс Thread**

Одним из ключевых аспектов в современном программировании является **многопоточность**. Ключевым понятием при работе с многоопоточностью является поток. Поток предствляет некоторую часть кода программы. При выполнении программы каждому потоку выделяется определенный квант времени. И при помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому, к примеру, клиент-серверные приложения (и не только они) практически не мыслимы без многопоточности.

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен **System.Threading**. В нем определен класс, представляющий отдельный поток - класс **Thread**.

Класс Thread определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства класса:

* **ExecutionContext**: позволяет получить контекст, в котором выполняется поток
* **IsAlive**: указывает, работает ли поток в текущий момент
* **IsBackground**: указывает, является ли поток фоновым
* **Name**: содержит имя потока
* **ManagedThreadId**: возвращает числовой идентификатор текущего потока
* **Priority**: хранит приоритет потока - значение перечисления **ThreadPriority**:
  + **Lowest**
  + **BelowNormal**
  + **Normal**
  + **AboveNormal**
  + **Highest**
* По умолчанию потоку задается значение Normal. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет Highest. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.
* **ThreadState** возвращает состояние потока - одно из значений перечисления **ThreadState**:
  + **Aborted**: поток остановлен, но пока еще окончательно не завершен
  + **AbortRequested**: для потока вызван метод Abort, но остановка потока еще не произошла
  + **Background**: поток выполняется в фоновом режиме
  + **Running**: поток запущен и работает (не приостановлен)
  + **Stopped**: поток завершен
  + **StopRequested**: поток получил запрос на остановку
  + **Suspended**: поток приостановлен
  + **SuspendRequested**: поток получил запрос на приостановку
  + **Unstarted**: поток еще не был запущен
  + **WaitSleepJoin**: поток заблокирован в результате действия методов Sleep или Join
* В процессе работы потока его статус многократно может измениться под действием методов. Так, в самом начале еще до применения метода Start его статус имеет значение Unstarted. Запустив поток, мы изменим его статус на Running. Вызвав метод Sleep, статус изменится на WaitSleepJoin.

Кроме того статическое свойство **CurrentThread** класса Thread позволяет получить текущий поток

В программе на C# есть как минимум один поток - главный поток, в котором выполняется метод Main.

Например, используем вышеописанные свойства для получения информации о потоке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using System.Threading;    // получаем текущий поток  Thread currentThread = Thread.CurrentThread;    //получаем имя потока  Console.WriteLine($"Имя потока: {currentThread.Name}");  currentThread.Name = "Метод Main";  Console.WriteLine($"Имя потока: {currentThread.Name}");    Console.WriteLine($"Запущен ли поток: {currentThread.IsAlive}");  Console.WriteLine($"Id потока: {currentThread.ManagedThreadId}");  Console.WriteLine($"Приоритет потока: {currentThread.Priority}");  Console.WriteLine($"Статус потока: {currentThread.ThreadState}"); |

В этом случае мы получим примерно следующий вывод:

Имя потока:

Имя потока: Метод Main

Запущен ли поток: True

Id потока: 1

Приоритет потока: Normal

Статус потока: Running

Так как по умолчанию свойство Name у объектов Thread не установлено, то в первом случае мы получаем в качестве значения этого свойства пустую строку.

Также класс Thread определяет ряд методов для управления потоком. Основные из них:

* Статический метод **GetDomain** возвращает ссылку на домен приложения
* Статический метод **GetDomainID** возвращает id домена приложения, в котором выполняется текущий поток
* Статический метод **Sleep** останавливает поток на определенное количество миллисекунд
* Метод **Interrupt** прерывает поток, который находится в состоянии WaitSleepJoin
* Метод **Join** блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
* Метод **Start** запускает поток

Например, применим метод Sleep для задания задержки выполнения приложения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | using System.Threading;    for(int i = 0; i < 10; i++)  {  Thread.Sleep(500); // задержка выполнения на 500 миллисекунд  Console.WriteLine(i);  } |

## **Создание потоков. Делегат ThreadStart**

Язык C# позволяет запускать и выполнять в рамках приложения несколько потоков, которые будут выполняться одновременно.

Для создания потока применяется один из конструкторов класса **Thread**:

* **Thread(ThreadStart)**: в качестве параметра принимает объект делегата ThreadStart, который представляет выполняемое в потоке действие
* **Thread(ThreadStart, Int32)**: в дополнение к делегату ThreadStart принимает числовое значение, которое устанавливает размер стека, выделяемого под данный поток
* **Thread(ParameterizedThreadStart)**: в качестве параметра принимает объект делегата ParameterizedThreadStart, который представляет выполняемое в потоке действие
* **Thread(ParameterizedThreadStart, Int32)**: вместе с делегатом ParameterizedThreadStart принимает числовое значение, которое устанавливает размер стека для данного потока

Вне зависимости от того, какой конструктор будет применяться для создания, нам надо определить выполняемое в потоке действие. В этой статье рассмотрим использование делегата ThreadStart. Этот делегат представляет действие, которое не принимает никаких параметров и не возвращает никакого значения:

|  |  |
| --- | --- |
| 1 | public delegate void ThreadStart(); |

То есть под этот делегат нам надо определить метод, который имеет тип void и не принимает никаких параметров. Примеры определения потоков:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Thread myThread1 = new Thread(Print);  Thread myThread2 = new Thread(new ThreadStart(Print));  Thread myThread3 = new Thread(()=>Console.WriteLine("Hello Threads"));    void Print() => Console.WriteLine("Hello Threads"); |

Для запуска нового потока применяется метод **Start** класса Thread:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using System.Threading;    // создаем новый поток  Thread myThread1 = new Thread(Print);  Thread myThread2 = new Thread(new ThreadStart(Print));  Thread myThread3 = new Thread(()=>Console.WriteLine("Hello Threads"));    myThread1.Start(); // запускаем поток myThread1  myThread2.Start(); // запускаем поток myThread2  myThread3.Start(); // запускаем поток myThread3    void Print() => Console.WriteLine("Hello Threads"); |

Преимуществом потоком является то, что они могут выполняться одновременно. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | using System.Threading;    // создаем новый поток  Thread myThread = new Thread(Print);  // запускаем поток myThread  myThread.Start();    // действия, выполняемые в главном потоке  for (int i = 0; i < 5; i++)  {  Console.WriteLine($"Главный поток: {i}");  Thread.Sleep(300);  }    // действия, выполняемые во втором потокке  void Print()  {  for (int i = 0; i < 5; i++)  {  Console.WriteLine($"Второй поток: {i}");  Thread.Sleep(400);  }  } |

Здесь новый поток будет производить действия, определенные в методе Print, то есть выводить числа от 0 до 4 на консоль. Причем после каждого вывода производится задержка на 400 миллисекунд.

В главном потоке - в методе Main создаем и запускаем новый поток, в котором выполняется метод Print:

|  |  |
| --- | --- |
| 1  2 | Thread myThread = new Thread(Print);  myThread.Start(); |

Кроме того, в главном потоке производим аналогичные действия - выводим на консоль числа от 0 до 4 с задержкой в 300 миллисекунд.

Таким образом, в нашей программе будут работать одновременно главный поток, представленный методом Main, и второй поток, в котором выполняется метод Print. Как только все потоки отработают, программа завершит свое выполнение. В итоге мы получим следующий консольный вывод:

Главный поток: 0

Второй поток: 0

Главный поток: 1

Второй поток: 1

Главный поток: 2

Второй поток: 2

Главный поток: 3

Второй поток: 3

Главный поток: 4

Второй поток: 4

Подобным образом мы можем создать и запускать и три, и четыре, и целый набор новых потоков, которые смогут решать те или иные задачи.

## **Потоки с параметрами и ParameterizedThreadStart**

В предыдущей статье было рассмотрено, как запускать в отдельных потоках методы без параметров. А что, если нам надо передать какие-нибудь параметры в поток?

Для этой цели используется делегат **ParameterizedThreadStart**, который передается в конструктор класса Thread:

|  |  |
| --- | --- |
| 1 | public delegate void ParameterizedThreadStart(object? obj); |

Применение делегата ParameterizedThreadStart во многом похоже на работу с ThreadStart. Рассмотрим на примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using System.Threading;    // создаем новые потоки  Thread myThread1 = new Thread(new ParameterizedThreadStart(Print));  Thread myThread2 = new Thread(Print);  Thread myThread3 = new Thread(message => Console.WriteLine(message));    // запускаем потоки  myThread1.Start("Hello");  myThread2.Start("Привет");  myThread3.Start("Salut");      void Print(object? message) => Console.WriteLine(message); |

При создании потока в конструктор класса Thread передается объект делегата ParameterizedThreadStart new Thread(new ParameterizedThreadStart(Print)), либо непосредственно метод, который соответствует этому делегату (new Thread(Print)), в том числе в виде лямбда-выражения (new Thread(message => Console.WriteLine(message)))

Затем при запуске потока в метод Start() передается значение, которое передается параметру метода Print. И в данном случае мы получим следующий консольный вывод:

Salut

Hello

Привет

При использовании ParameterizedThreadStart мы сталкиваемся с ограничением: мы можем запускать во втором потоке только такой метод, который в качестве единственного параметра принимает объект типа object?. Поэтому если мы хотим использовать данные других типов, в самом методе необходимо выполнить приведение типов. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | using System.Threading;    int number = 4;  // создаем новый поток  Thread myThread = new Thread(Print);  myThread.Start(number); // n \* n = 16      // действия, выполняемые во втором потокке  void Print(object? obj)  {  // здесь мы ожидаем получить число  if (obj is int n)  {  Console.WriteLine($"n \* n = {n \* n}");  }  } |

в данном случае нам надо дополнительно привести переданное значение к типу int, чтобы его использовать в вычислениях.

Но что делать, если нам надо передать не один, а несколько параметров различного типа? В этом случае можно определить свои типы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | using System.Threading;    Person tom = new Person("Tom", 37);  // создаем новый поток  Thread myThread = new Thread(Print);  myThread.Start(tom);    void Print(object? obj)  {  // здесь мы ожидаем получить объект Person  if (obj is Person person)  {  Console.WriteLine($"Name = {person.Name}");  Console.WriteLine($"Age = {person.Age}");  }  }    record class Person(string Name, int Age); |

Сначала определяем специальный класс Person, объект которого будет передаваться во второй поток, а в методе Main передаем его во второй поток.

Но тут опять же есть одно ограничение: метод Thread.Start не является типобезопасным, то есть мы можем передать в него любой тип, и потом нам придется приводить переданный объект к нужному нам типу. Для решения данной проблемы рекомендуется объявлять все используемые методы и переменные в специальном классе, а в основной программе запускать поток через ThreadStart. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using System.Threading;    Person tom = new Person("Tom", 37);  // создаем новый поток  Thread myThread = new Thread(tom.Print);  myThread.Start();    record class Person(string Name, int Age)  {  public void Print()  {  Console.WriteLine($"Name = {Name}");  Console.WriteLine($"Age = {Age}");  }  } |

## **Синхронизация потоков**

Нередко в потоках используются некоторые разделяемые ресурсы, общие для всей программы. Это могут быть общие переменные, файлы, другие ресурсы. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | int x = 0;    // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Thread myThread = new(Print);  myThread.Name = $"Поток {i}"; // устанавливаем имя для каждого потока  myThread.Start();  }    void Print()  {  x = 1;  for (int i = 1; i < 6; i++)  {  Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  x++;  Thread.Sleep(100);  }  } |

Здесь у нас запускаются пять потоков, которые вызывают метод Print и которые работают с общей переменной x. И мы предполагаем, что метод выведет все значения x от 1 до 5. И так для каждого потока. Однако в реальности в процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым. Например, в моем случае я получил следующий консольный вывод (он может в каждом конкретном случае различаться):

Поток 1: 1

Поток 5: 1

Поток 4: 1

Поток 2: 1

Поток 3: 1

Поток 1: 6

Поток 5: 7

Поток 3: 7

Поток 2: 7

Поток 4: 9

Поток 1: 11

Поток 4: 11

Поток 2: 11

Поток 3: 14

Поток 5: 11

Поток 1: 16

Поток 2: 16

Поток 3: 16

Поток 5: 18

Поток 4: 16

Поток 1: 21

Поток 5: 21

Поток 3: 21

Поток 2: 21

Поток 4: 21

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком. Для этого используется ключевое слово **lock**. Оператор **lock** определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока. Остальный потоки помещаются в очередь ожидания и ждут, пока текущий поток не освободит данный блок кода. В итоге с помощью lock мы можем переделать предыдущий пример следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | int x = 0;  object locker = new(); // объект-заглушка  // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Thread myThread = new(Print);  myThread.Name = $"Поток {i}";  myThread.Start();  }      void Print()  {  lock (locker)  {  x = 1;  for (int i = 1; i < 6; i++)  {  Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  x++;  Thread.Sleep(100);  }  }  } |

Для блокировки с ключевым словом lock используется объект-заглушка, в данном случае это переменная locker. Обычно это переменная типа object. И когда выполнение доходит до оператора lock, объект locker блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток. После окончания работы блока кода, объект locker освобождается и становится доступным для других потоков.

В этом случае консольный вывод будет более упорядоченным:

Поток 1: 1

Поток 1: 2

Поток 1: 3

Поток 1: 4

Поток 1: 5

Поток 5: 1

Поток 5: 2

Поток 5: 3

Поток 5: 4

Поток 5: 5

Поток 3: 1

Поток 3: 2

Поток 3: 3

Поток 3: 4

Поток 3: 5

Поток 2: 1

Поток 2: 2

Поток 2: 3

Поток 2: 4

Поток 2: 5

Поток 4: 1

Поток 4: 2

Поток 4: 3

Поток 4: 4

Поток 4: 5

## **Мониторы**

Наряду с оператором lock для синхронизации потоков мы можем использовать мониторы, представленные классом **System.Threading.Monitor**. Для управления синхронизацией этот класс предоставляет следующие методы:

* **void Enter(object obj)**: получает в эксклюзивное владение объект, передаваемый в качестве параметра.  
  **void Enter(object obj, bool acquiredLock)**: дополнительно принимает второй параметра - логическое значение, которое указывает, получено ли владение над объектом из первого параметра
* **void Exit(object obj)**: освобождает ранее захваченный объект
* **bool IsEntered(object obj)**: возвращает true, если монитор захватил объект obj
* **void Pulse (object obj)**: уведомляет поток из очереди ожидания, что текущий поток освободил объект obj
* **void PulseAll(object obj)**: уведомляет все потоки из очереди ожидания, что текущий поток освободил объект obj. После чего один из потоков из очереди ожидания захватывает объект obj.
* **bool TryEnter (object obj)**: пытается захватить объект obj. Если владение над объектом успешно получено, то возвращается значение true
* **bool Wait (object obj)**: освобождает блокировку объекта и переводит поток в очередь ожидания объекта. Следующий поток в очереди готовности объекта блокирует данный объект. А все потоки, которые вызвали метод Wait, остаются в очереди ожидания, пока не получат сигнала от метода Monitor.Pulse или Monitor.PulseAll, посланного владельцем блокировки.

Стоит отметить, что фактически конструкция оператора lock инкапсулирует в себе синтаксис использования мониторов. Например, в прошлой теме для синхронизации потоков применялся оператор **lock**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | int x = 0;  object locker = new(); // объект-заглушка  // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Thread myThread = new(Print);  myThread.Name = $"Поток {i}";  myThread.Start();  }      void Print()  {  lock (locker)  {  x = 1;  for (int i = 1; i < 6; i++)  {  Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  x++;  Thread.Sleep(100);  }  }  } |

Фактически данный пример будет эквивалентен следующему коду:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | int x = 0;  object locker = new(); // объект-заглушка  // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Thread myThread = new(Print);  myThread.Name = $"Поток {i}";  myThread.Start();  }    void Print()  {  bool acquiredLock = false;  try  {  Monitor.Enter(locker, ref acquiredLock);  x = 1;  for (int i = 1; i < 6; i++)  {  Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  x++;  Thread.Sleep(100);  }  }  finally  {  if (acquiredLock) Monitor.Exit(locker);  }  } |

Метод **Monitor.Enter** принимает два параметра - объект блокировки и значение типа bool, которое указывает на результат блокировки (если он равен true, то блокировка успешно выполнена). Фактически этот метод блокирует объект locker так же, как это делает оператор lock. А в блоке try...finally с помощью метода Monitor.Exit происходит освобождение объекта locker, если блокировка осуществлена успешно, и он становится доступным для других потоков.

## **Класс Lock и синхронизация**

Начиная с версии .NET 9 во фреймворк был добавлен новый API для синхронизации в виде класса **System.Threading.Lock**. Класс Lock служит для синхронизации доступа потоков к некоторой критической секции для предотвращения гонки за ресурсы между потоками. Есть разные варианты применения Lock. Рассмотрим их.

### **С оператором lock**

Самый простой способ использования Lock- с оператором **lock**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | int x = 0; // некоторый общий ресурс  Lock \_lockObj = new(); // объект-заглушка для синхронизации доступа  // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Thread myThread = new(Print);  myThread.Name = $"Поток {i}";  myThread.Start();  }    void Print()  {  lock(\_lockObj) // начало критической секции  {  x = 1;  for (int i = 1; i < 5; i++)  {  Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  x++;  Thread.Sleep(100);  }  } // завершение критической секции  } |

Данный пример похож на обычную синхронизацию с помощью оператора lock, когда в качестве заглушки используется произвольный объект. Здесь мы сначала определяем некоторый общий ресурс, к которому надо синхронизировать доступ - переменную x, и объект **Lock** синхронизации. Для примера здесь запускаем 5 потоков, каждый из которых обращается к некоторой общей секции в методе Print, где происходит доступ и изменение переменной x. Когда один поток захватывает объект \_lockObj и входит критическую секцию, остальные потоки ждут освобождения \_lockObj. После выхода из критической секции поток освобождает объект \_lockObj для других потоков. В итоге мы получим синхронизированный доступ потоков следующего рода:

Поток 1: 1

Поток 1: 2

Поток 1: 3

Поток 1: 4

Поток 1: 5

Поток 2: 1

Поток 2: 2

Поток 2: 3

Поток 2: 4

Поток 2: 5

Поток 4: 1

Поток 4: 2

Поток 4: 3

Поток 4: 4

Поток 4: 5

Поток 5: 1

Поток 5: 2

Поток 5: 3

Поток 5: 4

Поток 5: 5

Поток 3: 1

Поток 3: 2

Поток 3: 3

Поток 3: 4

Поток 3: 5

ПО консольному выводу мы видим, что у нас получился синхронизированный доступ к переменной x.

### **Метод Enter()**

С помощью вызова метода **Enter()** у объекта Lock можно обозначить начало критической секции, в которую опять же может войти только один поток. Остальные потоки опять же ожидают выхода этого потока из критической секции. Для выхода из этой секции (ее завершения) у объекта Lock вызывается метод **Exit()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | int x = 0; // некоторый общий ресурс  Lock \_lockObj = new(); // объект-заглушка для синхронизации доступа  // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Thread myThread = new(Print);  myThread.Name = $"Поток {i}";  myThread.Start();  }    void Print()  {  \_lockObj.Enter();; // начало критической секции  try  {  x = 1;  for (int i = 1; i < 6; i++)  {  Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  x++;  Thread.Sleep(100);  }  }  finally { \_lockObj.Exit(); } // завершение критической секции  } |

Стоит отметить, что выход из критической секции здесь осуществляется в блоке finally. Таким образом, даже в случае исключений мы гарантируем выход потока.

### **TryEnter()**

Метод **TryEnter()** также осуществляет вход в критическую секцию. Этот метод возвращает булевое значение - true или false. Если метод возвращает true, текущий поток является единственным потоком, который удерживает блокировку. Если метод возвращает false, то поток не ждет освобождения блокировки. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | int x = 0; // некоторый общий ресурс  Lock \_lockObj = new(); // объект-заглушка для синхронизации доступа  // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Thread myThread = new(Print);  myThread.Name = $"Поток {i}";  myThread.Start();  }      void Print()  {  if (\_lockObj.TryEnter()) // начало критической секции  {  try  {  x = 1;  for (int i = 1; i < 6; i++)  {  Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  x++;  Thread.Sleep(100);  }  }  finally { \_lockObj.Exit(); } // завершение критической секции  }  } |

Консольный вывод:

Поток 1: 1

Поток 1: 2

Поток 1: 3

Поток 1: 4

Поток 1: 5

Мы видим, что действия критической секции были выполнены только одним потоком. Потому что у нас потоки запускаются одновременно. Первый поток получил блокировку (метод \_lockObj.TryEnter() возвратил true) и этот поток стал выполнять действия из критической секции. Поскольку блокировка стала занята, для других потоков метод \_lockObj.TryEnter() возвратил false, и они не стали выполнять действия из критической секции и не стали ждать освобождения блокировки.

### **EnterScope()**

Еще один способ демонстрирует метод **EnterScope()**, который является рекомендуемым способом (с точки зрения Microsoft). Он также осуществляет вход в критическую секцию. Если один поток уже захватил блокировку и вошел в эту секцию, другие потоки ждут освобождения блокировки. Данный метод прежде всего предназначен для использования в конструкции **using**, которая автоматически освобождает все связанные ресурсы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | int x = 0; // некоторый общий ресурс  Lock \_lockObj = new(); // объект-заглушка для синхронизации доступа  // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Thread myThread = new(Print);  myThread.Name = $"Поток {i}";  myThread.Start();  }      void Print()  {  using (\_lockObj.EnterScope()) // начало критической секции  {  x = 1;  for (int i = 1; i < 6; i++)  {  Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  x++;  Thread.Sleep(100);  }  } // завершение критической секции  } |

Причем в данном случае не надо вызывать метод Exit(), выход из критической секции и освобождение блокировки происходит автоматически. Консольный вывод:

Поток 1: 1

Поток 1: 2

Поток 1: 3

Поток 1: 4

Поток 1: 5

Поток 5: 1

Поток 5: 2

Поток 5: 3

Поток 5: 4

Поток 5: 5

Поток 4: 1

Поток 4: 2

Поток 4: 3

Поток 4: 4

Поток 4: 5

Поток 3: 1

Поток 3: 2

Поток 3: 3

Поток 3: 4

Поток 3: 5

Поток 2: 1

Поток 2: 2

Поток 2: 3

Поток 2: 4

Поток 2: 5

## **Класс AutoResetEvent**

Класс **AutoResetEvent** также служит целям синхронизации потоков. Этот класс представляет событие синхронизации потоков, который позволяет при получении сигнала переключить данный объект-событие из сигнального в несигнальное состояние.

Для управления синхронизацией класс AutoResetEvent предоставляет ряд методов:

* **Reset()**: задает несигнальное состояние объекта, блокируя потоки.
* **Set();**: задает сигнальное состояние объекта, позволяя одному или нескольким ожидающим потокам продолжить работу.
* **WaitOne()**: задает несигнальное состояние и блокирует текущий поток, пока текущий объект AutoResetEvent не получит сигнал.

Событие синхронизации может находиться в сигнальном и несигнальном состоянии. Если состояние события несигнальное, поток, который вызывает метод WaitOne, будет заблокирован, пока состояние события не станет сигнальным. Метод Set, наоборот, задает сигнальное состояние события.

Так, в одной из предыдущих тем для синхронизации потоков применялся оператор lock:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | int x = 0;  object locker = new(); // объект-заглушка  // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Thread myThread = new(Print);  myThread.Name = $"Поток {i}";  myThread.Start();  }      void Print()  {  lock (locker)  {  x = 1;  for (int i = 1; i < 6; i++)  {  Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  x++;  Thread.Sleep(100);  }  }  } |

Перепишем этот пример с использованием **AutoResetEvent**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | int x = 0; // общий ресурс    AutoResetEvent waitHandler = new AutoResetEvent(true); // объект-событие    // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Thread myThread = new(Print);  myThread.Name = $"Поток {i}";  myThread.Start();  }      void Print()  {  waitHandler.WaitOne(); // ожидаем сигнала  x = 1;  for (int i = 1; i < 6; i++)  {  Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  x++;  Thread.Sleep(100);  }  waitHandler.Set(); // сигнализируем, что waitHandler в сигнальном состоянии  } |

Во-первых, создаем переменную типа AutoResetEvent. Передавая в конструктор значение true, мы тем самым указываем, что создаваемый объект изначально будет в сигнальном состоянии.

Когда начинает работать поток, то первым делом срабатывает определенный в методе Print вызов waitHandler.WaitOne(). Метод WaitOne указывает, что текущий поток переводится в состояние ожидания, пока объект waitHandler не будет переведен в сигнальное состояние. И так все потоки у нас переводятся в состояние ожидания.

После завершения работы вызывается метод waitHandler.Set, который уведомляет все ожидающие потоки, что объект waitHandler снова находится в сигнальном состоянии, и один из потоков "захватывает" данный объект, переводит в несигнальное состояние и выполняет свой код. А остальные потоки снова ожидают.

Так как в конструкторе AutoResetEvent мы указываем, что объект изначально находится в сигнальном состоянии, то первый из очереди потоков захватывает данный объект и начинает выполнять свой код.

Но если бы мы написали AutoResetEvent waitHandler = new AutoResetEvent(false), тогда объект изначально был бы в несигнальном состоянии, а поскольку все потоки блокируются методом waitHandler.WaitOne() до ожидания сигнала, то у нас попросту случилась бы блокировка программы, и программа не выполняла бы никаких действий.

Если у нас в программе используются несколько объектов AutoResetEvent, то мы можем использовать для отслеживания состояния этих объектов статические методы **WaitAll** и **WaitAny**, которые в качестве параметра принимают массив объектов класса WaitHandle - базового класса для AutoResetEvent.

Так, мы тоже можем использовать WaitAll в вышеприведенном примере. Для этого надо строку

|  |  |
| --- | --- |
| 1 | waitHandler.WaitOne(); |

заменить на следующую:

|  |  |
| --- | --- |
| 1 | AutoResetEvent.WaitAll(new WaitHandle[] {waitHandler}); |

## **Мьютексы**

Еще один инструмент управления синхронизацией потоков представляет класс **Mutex** или мьютекс, который также располагается в пространстве имен System.Threading.

Так, возьмем пример с оператором lock из одной из предыдущих тем, в котором применялась синхронизация потоков:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | int x = 0;  object locker = new(); // объект-заглушка  // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Thread myThread = new(Print);  myThread.Name = $"Поток {i}";  myThread.Start();  }      void Print()  {  lock (locker)  {  x = 1;  for (int i = 1; i < 6; i++)  {  Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  x++;  Thread.Sleep(100);  }  }  } |

И перепишем данный пример, используя мьютексы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | int x = 0;  Mutex mutexObj = new();    // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Thread myThread = new(Print);  myThread.Name = $"Поток {i}";  myThread.Start();  }    void Print()  {  mutexObj.WaitOne(); // приостанавливаем поток до получения мьютекса  x = 1;  for (int i = 1; i < 6; i++)  {  Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  x++;  Thread.Sleep(100);  }  mutexObj.ReleaseMutex(); // освобождаем мьютекс  } |

Сначала создаем объект мьютекса:

|  |  |
| --- | --- |
| 1 | Mutex mutexObj = new Mutex() |

Основную работу по синхронизации выполняют методы **WaitOne()** и **ReleaseMutex()**. Метод mutexObj.WaitOne() приостанавливает выполнение потока до тех пор, пока не будет получен мьютекс mutexObj.

Изначально мьютекс свободен, поэтому его получает один из потоков.

После выполнения всех действий, когда мьютекс больше не нужен, поток освобождает его с помощью метода mutexObj.ReleaseMutex(). А мьютекс получает один из ожидающих потоков.

Таким образом, когда выполнение дойдет до вызова mutexObj.WaitOne(), поток будет ожидать, пока не освободится мьютекс. И после его получения продолжит выполнять свою работу.

## **Семафоры**

**Семафоры** являются еще одним инструментом, который предлагает нам платформа .NET для управления синхронизацией. Семафоры позволяют ограничить количество потоков, которые имеют доступ к определенным ресурсам. В .NET семафоры представлены классом **Semaphore**.

Для создания семафора применяется один из конструкторов класса Semaphore:

* **Semaphore (int initialCount, int maximumCount)**: параметр initialCount задает начальное количество потоков, а maximumCount - максимальное количество потоков, которые имеют доступ к общим ресурсам
* **Semaphore (int initialCount, int maximumCount, string? name)**: в дополнение задает имя семафора
* **Semaphore (int initialCount, int maximumCount, string? name, out bool createdNew)**: последний параметр - createdNew при значении true указывает, что новый семафор был успешно создан. Если этот параметр равен false, то семафор с указанным именем уже существует

Для работы с потоками класс Semaphore имеет два основных метода:

* **WaitOne()**: ожидает получения свободного места в семафоре
* **Release()**: освобождает место в семафоре

Например, у нас такая задача: есть некоторое число читателей, которые приходят в библиотеку три раза в день и что-то там читают. И пусть у нас будет ограничение, что единовременно в библиотеке не может находиться больше трех читателей. Данную задачу очень легко решить с помощью семафоров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | // запускаем пять потоков  for (int i = 1; i < 6; i++)  {  Reader reader = new Reader(i);  }  class Reader  {  // создаем семафор  static Semaphore sem = new Semaphore(3, 3);  Thread myThread;  int count = 3;// счетчик чтения    public Reader(int i)  {  myThread = new Thread(Read);  myThread.Name = $"Читатель {i}";  myThread.Start();  }    public void Read()  {  while (count > 0)  {  sem.WaitOne(); // ожидаем, когда освободиться место    Console.WriteLine($"{Thread.CurrentThread.Name} входит в библиотеку");    Console.WriteLine($"{Thread.CurrentThread.Name} читает");  Thread.Sleep(1000);    Console.WriteLine($"{Thread.CurrentThread.Name} покидает библиотеку");    sem.Release(); // освобождаем место    count--;  Thread.Sleep(1000);  }  }  } |

В данной программе читатель представлен классом Reader. Он инкапсулирует всю функциональность, связанную с потоками, через переменную Thread myThread.

Сам семафор определяется в виде статической переменной sem:

static Semaphore sem = new Semaphore(3, 3);.

Его конструктор принимает два параметра: первый указывает, какому числу объектов изначально будет доступен семафор, а второй параметр указывает, какой максимальное число объектов будет использовать данный семафор. В данном случае у нас только три читателя могут одновременно находиться в библиотеке, поэтому максимальное число равно 3.

Основной функционал сосредоточен в методе Read, который и выполняется в потоке. В начале для ожидания получения семафора используется метод sem.WaitOne():

|  |  |
| --- | --- |
| 1 | sem.WaitOne(); // ожидаем, когда освободиться место |

После того, как в семафоре освободится место, данный поток заполняет свободное место и начинает выполнять все дальнейшие действия.

После окончания чтения мы высвобождаем семафор с помощью метода sem.Release():

|  |  |
| --- | --- |
| 1 | sem.Release(); // освобождаем место |

После этого в семафоре освобождается одно место, которое заполняет другой поток.

Пример консольного вывода:

Читатель 5 входит в библиотеку

Читатель 5 читает

Читатель 4 входит в библиотеку

Читатель 4 читает

Читатель 1 входит в библиотеку

Читатель 1 читает

Читатель 5 покидает библиотеку

Читатель 1 покидает библиотеку

Читатель 4 покидает библиотеку

Читатель 3 входит в библиотеку

Читатель 3 читает

Читатель 2 входит в библиотеку

Читатель 2 читает

Читатель 4 входит в библиотеку

Читатель 3 покидает библиотеку

Читатель 2 покидает библиотеку

Читатель 5 входит в библиотеку

Читатель 5 читает

Читатель 4 читает

Читатель 1 входит в библиотеку

Читатель 1 читает

Читатель 5 покидает библиотеку

Читатель 3 входит в библиотеку

Читатель 3 читает

Читатель 4 покидает библиотеку

Читатель 1 покидает библиотеку

Читатель 2 входит в библиотеку

Читатель 2 читает

Читатель 3 покидает библиотеку

Читатель 5 входит в библиотеку

Читатель 5 читает

Читатель 2 покидает библиотеку

Читатель 1 входит в библиотеку

Читатель 4 входит в библиотеку

Читатель 1 читает

Читатель 4 читает

Читатель 5 покидает библиотеку

Читатель 1 покидает библиотеку

Читатель 4 покидает библиотеку

Читатель 2 входит в библиотеку

Читатель 3 входит в библиотеку

Читатель 2 читает

Читатель 3 читает

Читатель 3 покидает библиотеку

Читатель 2 покидает библиотеку

# **Параллельное программирование и библиотека TPL**

## **Задачи и класс Task**

В эпоху многоядерных машин, которые позволяют параллельно выполнять сразу несколько процессов, стандартных средств работы с потоками в .NET уже оказалось недостаточно. Поэтому во фреймворк .NET была добавлена библиотека параллельных задач TPL (Task Parallel Library), основной функционал которой располагается в пространстве имен **System.Threading.Tasks**. Данная библиотека упрощает работу с многопроцессорными, многоядерными системами. Кроме того, она упрощает работу по созданию новых потоков. Поэтому обычно рекомендуется использовать именно TPL и ее классы для создания многопоточных приложений, хотя стандартные средства и класс Thread по-прежнему находят широкое применение.

В основе библиотеки TPL лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию. В библиотеке классов .NET задача представлена специальным классом - классом **Task**, который находится в пространстве имен **System.Threading.Tasks**. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке. Однако в любом случае следует отметить, что **задача - это не поток**.

Для определения и запуска задачи можно использовать различные способы.

* Первый способ создание объекта Task и вызов у него метода Start:

|  |  |
| --- | --- |
| 1  2 | Task task = new Task(() => Console.WriteLine("Hello Task!"));  task.Start(); |

* В качестве параметра объект Task принимает делегат Action, то есть мы можем передать любое действие, которое соответствует данному делегату, например, лямбда-выражение, как в данном случае, или ссылку на какой-либо метод. То есть в данном случае при выполнении задачи на консоль будет выводиться строка "Hello Task!".  
  А метод Start() собственно запускает задачу.
* Второй способ заключается в использовании статического метода **Task.Factory.StartNew()**. Этот метод также в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться. При этом этот метод сразу же запускает задачу:

|  |  |
| --- | --- |
| 1 | Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello Task!")); |

* В качестве результата метод возвращает запущенную задачу.
* Третий способ определения и запуска задач представляет использование статического метода **Task.Run()**:

|  |  |
| --- | --- |
| 1 | Task task = Task.Run(() => Console.WriteLine("Hello Task!")); |

* Метод Task.Run() также в качестве параметра может принимать делегат Action - выполняемое действие и возвращает объект Task.

Определим небольшую программу, где используем все эти способы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Task task1 = new Task(() => Console.WriteLine("Task1 is executed"));  task1.Start();    Task task2 = Task.Factory.StartNew(() => Console.WriteLine("Task2 is executed"));    Task task3 = Task.Run(() => Console.WriteLine("Task3 is executed")); |

Итак, в данном коде задачи создаются и запускаются, но при выполнении приложения на консоли мы можем не увидеть ничего. Почему? Потому что когда поток задачи запускается из основного потока программы - потока метода Main, приложение может завершить выполнение до того, как все три или даже хотя бы одна из трех задач начнет выполнение. Чтобы этого не произошло, мы можем программным образом ожидать завершения задачи.

### **Ожидание завершения задачи**

Чтобы приложение ожидало завершения задачи, можно использовать метод **Wait()** объекта Task:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Task task1 = new Task(() => Console.WriteLine("Task1 is executed"));  task1.Start();    Task task2 = Task.Factory.StartNew(() => Console.WriteLine("Task2 is executed"));    Task task3 = Task.Run(() => Console.WriteLine("Task3 is executed"));    task1.Wait(); // ожидаем завершения задачи task1  task2.Wait(); // ожидаем завершения задачи task2  task3.Wait(); // ожидаем завершения задачи task3 |

Возможный консольный вывод программы:

Task3 is executed

Task2 is executed

Task1 is executed

Консольный вывод не детерминирован, поскольку задачи не выполняются последовательно. Первая запущенная задача может завершить свое выполнение после последней задачи.

Стоит отметить, что метод **Wait()** блокирует вызывающий поток, в котором запущена задача, пока эта задача не завершит свое выполнение. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Console.WriteLine("Main Starts");  // создаем задачу  Task task1 = new Task(() =>  {  Console.WriteLine("Task Starts");  Thread.Sleep(1000); // задержка на 1 секунду - имитация долгой работы  Console.WriteLine("Task Ends");  });  task1.Start(); // запускаем задачу  task1.Wait(); // ожидаем выполнения задачи  Console.WriteLine("Main Ends"); |

Для эмуляции долговременной работы здесь в задаче task1 устанавливается задержка на 1 секунду. В итоге, когда выполнение дойдет до вызова task1.Wait() основной поток остановит свое выполнение и будет ждать завершения задачи. И мы получим следующий консольный вывод:

Main Starts

Task Starts

Task Ends

Main Ends

Если подобное поведение не принципиально, то ожидание завершения задачи можно поместить в конец метода Main:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Console.WriteLine("Main Starts");  // создаем задачу  Task task1 = new Task(() =>  {  Console.WriteLine("Task Starts");  Thread.Sleep(1000); // задержка на 1 секунду - имитация долгой работы  Console.WriteLine("Task Ends");  });  task1.Start(); // запускаем задачу  Console.WriteLine("Main Ends");  task1.Wait(); // ожидаем выполнения задачи |

В этом случае приложение все равно будет ждать завершения задачи, однако другие синхронные действия в основном потоке не будут блокироваться и ожидать завершения задачи.

### **Синхронный запуск задачи**

По умолчанию задачи запускаются асинхронно. Однако с помощью метода **RunSynchronously()** можно запускать синхронно:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Console.WriteLine("Main Starts");  // создаем задачу  Task task1 = new Task(() =>  {  Console.WriteLine("Task Starts");  Thread.Sleep(1000);  Console.WriteLine("Task Ends");  });  task1.RunSynchronously(); // запускаем задачу синхронно  Console.WriteLine("Main Ends"); // этот вызов ждет завершения задачи task1 |

### **Свойства класса Task**

Класс Task имеет ряд свойств, с помощью которых мы можем получить информацию об объекте. Некоторые из них:

* **AsyncState**: возвращает объект состояния задачи
* **CurrentId**: возвращает идентификатор текущей задачи (статическое свойство)
* **Id**: возвращает идентификатор текущей задачи
* **Exception**: возвращает объект исключения, возникшего при выполнении задачи
* **Status**: возвращает статус задачи. Представляет перечисление System.Threading.Tasks.TaskStatus, которое имеет следующие значения:
  + Canceled: задача отменена
  + Created: задача создана, но еще не запущена
  + Faulted: в процессе работы задачи произошло исключение
  + RanToCompletion: задача успешно завершена
  + Running: задача запущена, но еще не завершена
  + WaitingForActivation: задача ожидает активации и постановки в график выполнения
  + WaitingForChildrenToComplete: задача завершена и теперь ожидает завершения прикрепленных к ней дочерних задач
  + WaitingToRun: задача поставлена в график выполнения, но еще не начала свое выполнение
* **IsCompleted**: возвращает true, если задача завершена
* **IsCanceled**: возвращает true, если задача была отменена
* **IsFaulted**: возвращает true, если задача завершилась при возникновении исключения
* **IsCompletedSuccessfully**: возвращает true, если задача завершилась успешно

Используем некоторые из этих свойств:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Task task1 = new Task(() =>  {  Console.WriteLine($"Task{Task.CurrentId} Starts");  Thread.Sleep(1000);  Console.WriteLine($"Task{Task.CurrentId} Ends");  });  task1.Start(); //запускаем задачу    // получаем информацию о задаче  Console.WriteLine($"task1 Id: {task1.Id}");  Console.WriteLine($"task1 is Completed: {task1.IsCompleted}");  Console.WriteLine($"task1 Status: {task1.Status}");    task1.Wait(); // ожидаем завершения задачи |

Пример консольного вывода:

task1 Id: 1

Task1 Starts

task1 is Completed: False

task1 Status: Running

Task1 Ends

## **Работа с классом Task**

### **Вложенные задачи**

Одна задача может запускать другую - вложенную задачу. При этом эти задачи выполняются независимо друг от друга. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var outer = Task.Factory.StartNew(() => // внешняя задача  {  Console.WriteLine("Outer task starting...");  var inner = Task.Factory.StartNew(() => // вложенная задача  {  Console.WriteLine("Inner task starting...");  Thread.Sleep(2000);  Console.WriteLine("Inner task finished.");  });  });  outer.Wait(); // ожидаем выполнения внешней задачи  Console.WriteLine("End of Main"); |

Несмотря на то, что здесь мы ожидаем выполнения внешней задачи, но вложенная задача может завершить выполнение даже после завершения метода Main:

Outer task starting...

End of Main

При этом внутренняя задача может даже не начать свое выполнение к завершению работы основного потока программы. То есть в данном случае внешняя и вложенная задачи выполняются независимо друг от друга.

Если необходимо, чтобы вложенная задача выполнялась как часть внешней, необходимо использовать значение **TaskCreationOptions.AttachedToParent**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var outer = Task.Factory.StartNew(() => // внешняя задача  {  Console.WriteLine("Outer task starting...");  var inner = Task.Factory.StartNew(() => // вложенная задача  {  Console.WriteLine("Inner task starting...");  Thread.Sleep(2000);  Console.WriteLine("Inner task finished.");  }, TaskCreationOptions.AttachedToParent);  });  outer.Wait(); // ожидаем выполнения внешней задачи  Console.WriteLine("End of Main"); |

Консольный вывод:

Outer task starting...

Inner task starting...

Inner task finished.

End of Main

В данном случае вложенная задача прикреплена к внешней и выполняется как часть внешней задачи. И внешняя задача завершится только когда завершатся все прикрепленные к ней вложенные задачи.

### **Массив задач**

Также как и с потоками, мы можем создать и запустить массив задач. Можно определить все задачи в массиве непосредственно через объект Task:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Task[] tasks1 = new Task[3]  {  new Task(() => Console.WriteLine("First Task")),  new Task(() => Console.WriteLine("Second Task")),  new Task(() => Console.WriteLine("Third Task"))  };  // запуск задач в массиве  foreach (var t in tasks1)  t.Start(); |

Либо также можно использовать методы Task.Factory.StartNew или Task.Run и сразу запускать все задачи:

|  |  |
| --- | --- |
| 1  2  3  4 | Task[] tasks2 = new Task[3];  int j = 1;  for (int i = 0; i < tasks2.Length; i++)  tasks2[i] = Task.Factory.StartNew(() => Console.WriteLine($"Task {j++}")); |

Но в любом случае мы опять же можем столкнуться с тем, что все задачи из массива могут завершиться после того, как отработает метод Main, в котором запускаются эти задачи:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Task[] tasks = new Task[3];  for(var i = 0; i < tasks.Length; i++)  {  tasks[i] = new Task(() =>  {  Thread.Sleep(1000); // эмуляция долгой работы  Console.WriteLine($"Task{i} finished");  });  tasks[i].Start(); // запускаем задачу  }  Console.WriteLine("Завершение метода Main"); |

Один из возможных консольных выводов программы:

Завершение метода Main

Если необходимо завершить выполнение программы или вообще выполнять некоторый код лишь после того, как все задачи из массива завершатся, то применяется метод **Task.WaitAll(tasks)**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Task[] tasks = new Task[3];  for(var i = 0; i < tasks.Length; i++)  {  tasks[i] = new Task(() =>  {  Thread.Sleep(1000); // эмуляция долгой работы  Console.WriteLine($"Task{i} finished");  });  tasks[i].Start(); // запускаем задачу  }  Console.WriteLine("Завершение метода Main");    Task.WaitAll(tasks); // ожидаем завершения всех задач |

В этом случае сначала завершатся все задачи, и лишь только потом будет выполняться последующий код из метода Main:

Завершение метода Main

Task3 finished

Task3 finished

Task3 finished

В то же время порядок выполнения самих задач в массиве также недетерминирован.

Также мы можем применять метод **Task.WaitAny(tasks)**. Он ждет, пока завершится хотя бы одна из массива задач.

### **Возвращение результатов из задач**

Задачи могут не только выполняться как процедуры, но и возвращать определенные результаты:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int n1 =4, n2 = 5;  Task<int> sumTask = new Task<int>(() => Sum(n1, n2));  sumTask.Start();    int result = sumTask.Result;  Console.WriteLine($"{n1} + {n2} = {result}"); // 4 + 5 = 9    int Sum(int a, int b) => a + b; |

Во-первых, чтобы получать из задачи не который результат, необходимо типизировать объект Task тем типом, объект которого мы хотим получить из задачи. Например, в примере выше мы ожидаем из задачи sumTask получить число типа int, соответственно типизируем объект Task данным типом - Task<int>.

И, во-вторых, в качестве задачи должен выполняться метод, который возвращает данный тип объекта. Так, в данном случае у нас в качестве задачи выполняется метод Sum, которая принимаетдва числа и на выходе возвращает их сумму - значение типа int.

Возвращаемое число будет храниться в свойстве Result: sumTask.Result. Нам не надо его приводить к типу int, оно уже само по себе будет представлять число.

|  |  |
| --- | --- |
| 1 | int result = sumTask.Result; |

При этом при обращении к свойству Result текущий поток останавливает выполнение и ждет, когда будет получен результат из выполняемой задачи.

Другой пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Task<Person> defaultPersonTask = new Task<Person>(() => new Person("Tom", 37));  defaultPersonTask.Start();    Person defaultPerson = defaultPersonTask.Result;  Console.WriteLine($"{defaultPerson.Name} - {defaultPerson.Age}"); // Tom - 37    record class Person(string Name, int Age); |

В данном случае задача defaultPersonTask возвращает объект типа Person, который мы можем получить из свойства Result.

## **Задачи продолжения**

Задачи продолжения или **continuation task** позволяют определить задачи, которые выполняются после завершения других задач. Благодаря этому мы можем вызвать после выполнения одной задачи несколько других, определить условия их вызова, передать из предыдущей задачи в следующую некоторые данные.

Задачи продолжения похожи на методы обратного вызова, но фактически являются обычными задачами Task. Посмотрим на примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | Task task1 = new Task(() =>  {  Console.WriteLine($"Id задачи: {Task.CurrentId}");  });    // задача продолжения - task2 выполняется после task1  Task task2 = task1.ContinueWith(PrintTask);    task1.Start();    // ждем окончания второй задачи  task2.Wait();  Console.WriteLine("Конец метода Main");      void PrintTask(Task t)  {  Console.WriteLine($"Id задачи: {Task.CurrentId}");  Console.WriteLine($"Id предыдущей задачи: {t.Id}");  Thread.Sleep(3000);  } |

Первая задача задается с помощью лямбда-выражения, которое просто выводит id этой задачи. Вторая задача - задача продолжения задается с помощью метода **ContinueWith**, который в качестве параметра принимает делегат Action<Task>. То есть метод PrintTask, который передается в вызов ContinueWith, должен принимать параметр типа Task.

Благодаря передачи в метод параметра Task, мы можем получить различные свойства предыдущей задачи, как например, в данном случае получает ее Id.

И после завершения задачи task1 сразу будет вызываться задача task2. Консольный вывод программы:

Id задачи: 1

Id задачи: 2

Id предыдущей задачи: 1

Конец метода Main

Также мы можем передавать конкретный результат работы предыдущей задачи:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Task<int> sumTask = new Task<int>(() => Sum(4, 5));    // задача продолжения  Task printTask = sumTask.ContinueWith(task => PrintResult(task.Result));    sumTask.Start();    // ждем окончания второй задачи  printTask.Wait();  Console.WriteLine("Конец метода Main");    int Sum(int a, int b) => a + b;  void PrintResult(int sum) => Console.WriteLine($"Sum: {sum}"); |

В данном случае задача sumTask выполняет метод Sum и возвращает его результат. Задача printTask является задачей продолжения, выполняется сразу после sumTask и получает ее результат. Так, в вызове

|  |  |
| --- | --- |
| 1 | Task printTask = sumTask.ContinueWith(task => PrintResult(task.Result)); |

Параметр task в лямбда-выражении фактически представляет задачу sumTask, из которой извлекается результат.

Подобным образом можно построить целую цепочку последовательно выполняющихся задач:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Task task1 = new Task(() => Console.WriteLine($"Current Task: {Task.CurrentId}"));    // задача продолжения  Task task2 = task1.ContinueWith(t =>  Console.WriteLine($"Current Task: {Task.CurrentId} Previous Task: {t.Id}"));    Task task3 = task2.ContinueWith(t =>  Console.WriteLine($"Current Task: {Task.CurrentId} Previous Task: {t.Id}"));      Task task4 = task3.ContinueWith(t =>  Console.WriteLine($"Current Task: {Task.CurrentId} Previous Task: {t.Id}"));    task1.Start();    task4.Wait(); // ждем завершения последней задачи  Console.WriteLine("Конец метода Main"); |

здесь друг за другом выполняются задачи task1, task2, task3, task4. Консольный вывод программы:

Current Task: 1

Current Task: 2 Previous Task: 1

Current Task: 3 Previous Task: 2

Current Task: 4 Previous Task: 3

Конец метода Main

## **Класс Parallel**

Класс **Parallel** также является частью TPL и предназначен для упрощения параллельного выполнения кода. **Parallel** имеет ряд методов, которые позволяют распараллелить задачу.

Одним из методов, позволяющих параллельное выполнение задач, является метод Invoke:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | // метод Parallel.Invoke выполняет три метода  Parallel.Invoke(  Print,  () =>  {  Console.WriteLine($"Выполняется задача {Task.CurrentId}");  Thread.Sleep(3000);  },  () => Square(5)  );    void Print()  {  Console.WriteLine($"Выполняется задача {Task.CurrentId}");  Thread.Sleep(3000);  }  // вычисляем квадрат числа  void Square(int n)  {  Console.WriteLine($"Выполняется задача {Task.CurrentId}");  Thread.Sleep(3000);  Console.WriteLine($"Результат {n \* n}");  } |

Метод Parallel.Invoke в качестве параметра принимает массив объектов Action, то есть мы можем передать в данный метод набор методов, которые будут вызываться при его выполнении. Количество методов может быть различным, но в данном случае мы определяем выполнение трех методов. Опять же как и в случае с классом Task мы можем передать либо название метода, либо лямбда-выражение.

И таким образом, при наличии нескольких ядер на целевой машине данные методы будут выполняться параллельно на различных ядрах. Пример консольного вывода программы:

Выполняется задача 1

Выполняется задача 3

Выполняется задача 2

Результат 25

### **Parallel.For**

Метод Parallel.For позволяет выполнять итерации цикла параллельно. Он имеет следующее определение:

|  |  |
| --- | --- |
| 1 | For(int, int, Action<int>) |

Первый параметр метода задает начальный индекс элемента в цикле, а второй параметр - конечный индекс. Третий параметр - делегат Action - указывает на метод, который будет выполняться один раз за итерацию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Parallel.For(1, 5, Square);    // вычисляем квадрат числа  void Square(int n)  {  Console.WriteLine($"Выполняется задача {Task.CurrentId}");  Console.WriteLine($"Квадрат числа {n} равен {n \* n}");  Thread.Sleep(2000);  } |

В данном случае в качестве первого параметра в метод Parallel.For передается число 1, а в качестве второго - число 5. Таким образом, метод будет вести итерацию с 1 до 4 включительно. Третий параметр представляет метод, который вычисляет квадрат числа. Так как этот параметр представляет тип Action<int>, то этот метод в качестве параметра должен принимать объект int.

А в качестве значения параметра в этот метод передается счетчик, который проходит в цикле от 1 до 4 включительно. И метод Square, таким образом, вызывается 4 раза. Пример консольного вывода:

Выполняется задача 1

Выполняется задача 2

Квадрат числа 4 равен 16

Выполняется задача 4

Квадрат числа 1 равен 1

Выполняется задача 3

Квадрат числа 3 равен 9

Квадрат числа 2 равен 4

### **Parallel.ForEach**

Метод **Parallel.ForEach** осуществляет итерацию по коллекции, реализующей интерфейс **IEnumerable**, подобно циклу foreach, только осуществляет параллельное выполнение перебора. Он имеет следующее определение:

|  |  |
| --- | --- |
| 1 | ParallelLoopResult ForEach<TSource>(IEnumerable<TSource> source,Action<TSource> body) |

где первый параметр представляет перебираемую коллекцию, а второй параметр - делегат, выполняющийся один раз за итерацию для каждого перебираемого элемента коллекции.

На выходе метод возвращает структуру ParallelLoopResult, которая содержит информацию о выполнении цикла.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | ParallelLoopResult result = Parallel.ForEach<int>(  new List<int>() { 1, 3, 5, 8 },  Square  );    // вычисляем квадрат числа  void Square(int n)  {  Console.WriteLine($"Выполняется задача {Task.CurrentId}");  Console.WriteLine($"Квадрат числа {n} равен {n \* n}");  Thread.Sleep(2000);  } |

В данном случае поскольку мы используем коллекцию объектов int, то и метод, который мы передаем в качестве второго параметра, должен в качестве параметра принимать значение int. Пример консольного вывода:

Выполняется задача 1

Выполняется задача 3

Квадрат числа 8 равен 64

Выполняется задача 4

Квадрат числа 3 равен 9

Выполняется задача 2

Квадрат числа 5 равен 25

Квадрат числа 1 равен 1

### **Выход из цикла**

В стандартных циклах for и foreach предусмотрен преждевременный выход из цикла с помощью оператора break. В методах Parallel.ForEach и Parallel.For мы также можем, не дожидаясь окончания цикла, выйти из него:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | ParallelLoopResult result = Parallel.For(1, 10, Square);  if (!result.IsCompleted)  Console.WriteLine($"Выполнение цикла завершено на итерации {result.LowestBreakIteration}");    // вычисляем квадрат числа  void Square(int n, ParallelLoopState pls)  {  if (n == 5) pls.Break(); // если передано 5, выходим из цикла    Console.WriteLine($"Квадрат числа {n} равен {n \* n}");  Thread.Sleep(2000);  } |

Здесь метод Square, который обрабатывает каждую итерацию, принимает дополнительный параметр - объект **ParallelLoopState**. И если счетчик в цикле достигнет значения 5, вызывается метод Break. Благодаря чему система осуществит выход и прекратит выполнение метода Parallel.For при первом удобном случае.

Методы Parallel.ForEach и Parallel.For возвращают объект **ParallelLoopResult**, наиболее значимыми свойствами которого являются два следующих:

* **IsCompleted**: определяет, завершилось ли полное выполнение параллельного цикла
* **LowestBreakIteration**: возвращает индекс, на котором произошло прерывание работы цикла

Так как у нас на индексе равном 5 происходит прерывание, то свойство IsCompleted будет иметь значение false, а LowestBreakIteration будет равно 5.

## **Отмена задач и параллельных операций. CancellationToken**

Параллельное выполнение задач может занимать много времени. И иногда может возникнуть необходимость прервать выполняемую задачу. Для этого платформа .NET предоставляет структуру **CancellationToken** из пространства имен System.Threading.

Общий алгоритм отмены задачи обычно предусматривает следующий порядок действий:

1. Создание объекта **CancellationTokenSource**, который управляет и посылает уведомление об отмене токену.
2. С помощью свойства **CancellationTokenSource.Token** получаем собственно токен - объект структуры **CancellationToken** и передаем его в задачу, которая может быть отменена.

|  |  |
| --- | --- |
| 1  2 | CancellationTokenSource cancelTokenSource = new CancellationTokenSource();  CancellationToken token = cancelTokenSource.Token; |

1. Для передачи токена в задачу можно применять один из конструкторов класса Task:

|  |  |
| --- | --- |
| 1  2  3 | CancellationTokenSource cancelTokenSource = new CancellationTokenSource();  CancellationToken token = cancelTokenSource.Token;  Task task = new Task(() => { выполняемые\_действия}, token); // |

1. Определяем в задаче действия на случай ее отмены.
2. Вызываем метод **CancellationTokenSource.Cancel()**, который устанавливает для свойства **CancellationToken.IsCancellationRequested** значение true. Стоит понимать, что сам по себе метод **CancellationTokenSource.Cancel()** не отменяет задачу, он лишь посылает уведомление об отмене через установку свойства CancellationToken.IsCancellationRequested. Каким образом будет происходить выход из задачи, это решает сам разработчик.
3. Класс CancellationTokenSource реализует интерфейс IDisposable. И когда работа с объектом CancellationTokenSource завершена, у него следует вызвать метод Dispose для освобождения всех связанных с ним используемых ресурсов. (Вместо явного вызова метода Dispose можно использовать конструкцию using).

Теперь касательно третьего пункта - определения действий отмены задачи. Как именно завершить задачу? Конкретные действия на лежат целиком на разработчике, тем не менее есть два общих варианта выхода:

* При получении сигнала отмены выйти из метода задачи, например, с помощью оператора return или построив логику метода соответствующим образом. Но следует учитывать, что в этом случае задача перейдет в состояние TaskStatus.RanToCompletion, а не в состояние TaskStatus.Canceled.
* При получении сигнала отмены сгенерировать исключение **OperationCanceledException**, вызвав у токена метод **ThrowIfCancellationRequested()**. После этого задача перейдет в состояние TaskStatus.Canceled.

### **Мягкий выход из задачи без исключения OperationCanceledException**

Сначала рассмотрим первый - "мягкий" вариант завершения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | CancellationTokenSource cancelTokenSource = new CancellationTokenSource();  CancellationToken token = cancelTokenSource.Token;    // задача вычисляет квадраты чисел  Task task = new Task(() =>  {  for (int i = 1; i < 10; i++)  {  if (token.IsCancellationRequested) // проверяем наличие сигнала отмены задачи  {  Console.WriteLine("Операция прервана");  return; // выходим из метода и тем самым завершаем задачу  }  Console.WriteLine($"Квадрат числа {i} равен {i \* i}");  Thread.Sleep(200);  }  }, token);  task.Start();    Thread.Sleep(1000);  // после задержки по времени отменяем выполнение задачи  cancelTokenSource.Cancel();  // ожидаем завершения задачи  Thread.Sleep(1000);  // проверяем статус задачи  Console.WriteLine($"Task Status: {task.Status}");  cancelTokenSource.Dispose(); // освобождаем ресурсы |

В данном случае задача task вычисляет и выводит на консоль квадраты чисел от 1 до 9. Для отмены задачи нам надо создать и использовать токен. Вначале создается объект CancellationTokenSource:

|  |  |
| --- | --- |
| 1 | CancellationTokenSource cancelTokenSource = new CancellationTokenSource(); |

Затем из него получаем сам токен:

|  |  |
| --- | --- |
| 1 | CancellationToken token = cancelTokenSource.Token; |

Чтобы отменить операцию, необходимо вызвать метод Cancel() у объекта CancellationTokenSource:

|  |  |
| --- | --- |
| 1 | cancelTokenSource.Cancel(); |

В данном случае отмена задачи вызывается через секунду, чтобы задача произвела некоторые действия.

В самом методе задачи в цикле мы можем отловить сигнал отмены с помощью проверки свойства token.IsCancellationRequested:

|  |  |
| --- | --- |
| 1  2  3  4  5 | if (token.IsCancellationRequested)  {  Console.WriteLine("Операция прервана");  return;  } |

Если был вызван метод cancelTokenSource.Cancel(), то выражение token.IsCancellationRequested возвращает true.

После завершения задачи проверяем ее статус:

|  |  |
| --- | --- |
| 1 | Console.WriteLine($"Task Status: {task.Status}"); |

Поскольку задача успешно завершена, у задачи должен быть статус RanToCompletion

И в конце у объекта CancellationTokenSource вызываем метод Dispose:

|  |  |
| --- | --- |
| 1 | cancelTokenSource.Dispose(); |

Консольный вывод программы:

Квадрат числа 1 равен 1

Квадрат числа 2 равен 4

Квадрат числа 3 равен 9

Квадрат числа 4 равен 16

Квадрат числа 5 равен 25

Операция прервана

Task Status: RanToCompletion

### **Отмена задачи с помощью генерации исключения**

Второй способ завершения задачи представляет генерация исключения **OperationCanceledException**. Для этого применяется метод **ThrowIfCancellationRequested()** объекта CancellationToken:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | CancellationTokenSource cancelTokenSource = new CancellationTokenSource();  CancellationToken token = cancelTokenSource.Token;    Task task = new Task(() =>  {  for (int i = 1; i < 10; i++)  {  if (token.IsCancellationRequested)  token.ThrowIfCancellationRequested(); // генерируем исключение    Console.WriteLine($"Квадрат числа {i} равен {i \* i}");  Thread.Sleep(200);  }  }, token);  try  {  task.Start();  Thread.Sleep(1000);  // после задержки по времени отменяем выполнение задачи  cancelTokenSource.Cancel();    task.Wait(); // ожидаем завершения задачи  }  catch (AggregateException ae)  {  foreach (Exception e in ae.InnerExceptions)  {  if (e is TaskCanceledException)  Console.WriteLine("Операция прервана");  else  Console.WriteLine(e.Message);  }  }  finally  {  cancelTokenSource.Dispose();  }    // проверяем статус задачи  Console.WriteLine($"Task Status: {task.Status}"); |

Здесь опять же проверяем значение свойства IsCancellationRequested, и если оно равно true, генерируем исключение:

|  |  |
| --- | --- |
| 1  2 | if (token.IsCancellationRequested)  token.ThrowIfCancellationRequested(); // генерируем исключение |

Чтобы обработать исключение, помещаем весь код работы с задачей в конструкцию try..catch и также с помощью вызова cancelTokenSource.Cancel() посылаем сообщение об отмене задачи.

Стоит отметить, что генерируемое исключение будет спрятано в объекте AggregateException, который по сути представляет набор исключений. Если причина исключения состояла в отмене задачи, то мы можем найти в этом наборе исключений исключение типа **TaskCanceledException**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | catch (AggregateException ae)  {  foreach (Exception e in ae.InnerExceptions)  {  if (e is TaskCanceledException)  Console.WriteLine("Операция прервана");  else  Console.WriteLine(e.Message);  }  } |

Класс TaskCanceledException является производным от OperationCanceledException. Исключение типа TaskCanceledException возникает, если для задачи устанавливается статус Canceled.

Консольный вывод программы:

Квадрат числа 1 равен 1

Квадрат числа 2 равен 4

Квадрат числа 3 равен 9

Квадрат числа 4 равен 16

Квадрат числа 5 равен 25

Операция прервана

Task Status: Canceled

Стоит отметить, что исключение возникает только тогда, когда мы останавливаем текущий поток и ожидаем завершения задачи с помощью методов Wait или WaitAll. Если эти методы не используются для ожидания задачи, то для нее просто устанавливается состояние Canceled. Например, в следующем случае исключение не возникнет:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | CancellationTokenSource cancelTokenSource = new CancellationTokenSource();  CancellationToken token = cancelTokenSource.Token;    Task task = new Task(() =>  {  for (int i = 1; i < 10; i++)  {  if (token.IsCancellationRequested)  token.ThrowIfCancellationRequested(); // генерируем исключение    Console.WriteLine($"Квадрат числа {i} равен {i \* i}");  Thread.Sleep(200);  }  }, token);  try  {  task.Start();  Thread.Sleep(1000);  // после задержки по времени отменяем выполнение задачи  cancelTokenSource.Cancel();    // ожидаем завершения задачи  Thread.Sleep(1000);  }  catch (AggregateException ae)  {  foreach (Exception e in ae.InnerExceptions)  {  if (e is TaskCanceledException)  Console.WriteLine("Операция прервана");  else  Console.WriteLine(e.Message);  }  }  finally  {  cancelTokenSource.Dispose();  }    // проверяем статус задачи  Console.WriteLine($"Task Status: {task.Status}"); |

Консольный вывод программы:

Квадрат числа 1 равен 1

Квадрат числа 2 равен 4

Квадрат числа 3 равен 9

Квадрат числа 4 равен 16

Квадрат числа 5 равен 25

Task Status: Canceled

### **Регистрация обработчика отмены задачи**

Выше для проверки сигнала отмены применялось свойство IsCancellationRequested. Но есть и другой способ узнать о том, что был послан сигнал отмены задачи. Метод **Register()** позволяет зарегистрировать обработчик отмены задачи в виде делегата Action:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | CancellationTokenSource cancelTokenSource = new CancellationTokenSource();  CancellationToken token = cancelTokenSource.Token;    // задача вычисляет квадраты чисел  Task task = new Task(() =>  {  int i = 1;  token.Register(() =>  {  Console.WriteLine("Операция прервана");  i = 10;  });  for (; i < 10; i++)  {  Console.WriteLine($"Квадрат числа {i} равен {i \* i}");  Thread.Sleep(400);  }  }, token);  task.Start();    Thread.Sleep(1000);  // после задержки по времени отменяем выполнение задачи  cancelTokenSource.Cancel();  // ожидаем завершения задачи  Thread.Sleep(1000);  // проверяем статус задачи  Console.WriteLine($"Task Status: {task.Status}");  cancelTokenSource.Dispose(); // освобождаем ресурсы |

Здесь обработчик отмены представлен лямбда-выражением:

|  |  |
| --- | --- |
| 1  2  3  4  5 | token.Register(() =>  {  Console.WriteLine("Операция прервана");  i = 10;  }); |

Поскольку действие задачи представляет цикл, который выполняется при значении i меньше 10, то установка этой переменной в обработчике отмены приведет к выходу из цикла и соответственно завершению задачи.

### **Передача токена во внешний метод**

Если операция, которая выполняется в задаче, представляет внешний метод, то ему можно передавать в качестве одного из параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | CancellationTokenSource cancelTokenSource = new CancellationTokenSource();  CancellationToken token = cancelTokenSource.Token;    Task task = new Task(() =>PrintSquares(token), token);  try  {  task.Start();  Thread.Sleep(1000);  // после задержки по времени отменяем выполнение задачи  cancelTokenSource.Cancel();    // ожидаем завершения задачи  task.Wait();  }  catch (AggregateException ae)  {  foreach (Exception e in ae.InnerExceptions)  {  if (e is TaskCanceledException)  Console.WriteLine("Операция прервана");  else  Console.WriteLine(e.Message);  }  }  finally  {  cancelTokenSource.Dispose();  }    // проверяем статус задачи  Console.WriteLine($"Task Status: {task.Status}");      void PrintSquares(CancellationToken token)  {  for (int i = 1; i < 10; i++)  {  if (token.IsCancellationRequested)  token.ThrowIfCancellationRequested(); // генерируем исключение    Console.WriteLine($"Квадрат числа {i} равен {i \* i}");  Thread.Sleep(200);  }  } |

### **Отмена параллельных операций Parallel**

Для отмены выполнения параллельных операций, запущенных с помощью методов **Parallel.For()** и **Parallel.ForEach()**, можно использовать перегруженные версии данных методов, которые принимают в качестве параметра объект **ParallelOptions**. Данный объект позволяет установить токен:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | CancellationTokenSource cancelTokenSource = new CancellationTokenSource();  CancellationToken token = cancelTokenSource.Token;    // в другой задаче посылаем сигнал отмены  new Task(() =>  {  Thread.Sleep(400);  cancelTokenSource.Cancel();  }).Start();    try  {  Parallel.ForEach<int>(new List<int>() { 1, 2, 3, 4, 5},  new ParallelOptions { CancellationToken = token }, Square);  // или так  //Parallel.For(1, 5, new ParallelOptions { CancellationToken = token }, Square);  }  catch (OperationCanceledException)  {  Console.WriteLine("Операция прервана");  }  finally  {  cancelTokenSource.Dispose();  }    void Square(int n)  {  Thread.Sleep(3000);  Console.WriteLine($"Квадрат числа {n} равен {n \* n}");  } |

В параллельной запущенной задаче через 400 миллисекунд происходит вызов cancelTokenSource.Cancel(), в результате программа выбрасывает исключение OperationCanceledException, и выполнение параллельных операций прекращается.

# **Aсинхронное программирование**

## **Асинхронные методы, async и await**

Нередко программа выполняет такие операции, которые могут занять продолжительное время, например, обращение к сетевым ресурсам, чтение-запись файлов, обращение к базе данных и т.д. Такие операции могут серьезно нагрузить приложение. Особенно это актуально в графических (десктопных или мобильных) приложениях, где продолжительные операции могут блокировать интерфейс пользователя и негативно повлиять на желание пользователя работать с программой, или в веб-приложениях, которые должны быть готовы обслуживать тысячи запросов в секунду. В синхронном приложении при выполнении продолжительных операций в основном потоке этот поток просто бы блокировался на время выполнения операции. И чтобы продолжительные операции не блокировали общую работу приложения, в C# можно задействовать асинхронность.

**Асинхронность** позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы и при этом более экономно использовать потоки. Асинхронные методы выполняются в отдельных потоках. Однако при выполнении продолжительной операции поток асинхронного метода возвратится в пул потоков и будет использоваться для других задач. А когда продолжительная операция завершит свое выполнение, для асинхронного метода опять выделяется поток из пула потоков, и асинхронный метод продолжает свою работу.

Ключевыми для работы с асинхронными вызовами в C# являются два оператора: **async** и **await**, цель которых - упростить написание асинхронного кода. Они используются вместе для создания асинхронного метода.

**Асинхронный метод** обладает следующими признаками:

* В заголовке метода используется модификатор **async**
* Метод содержит одно или несколько выражений **await**
* В качестве возвращаемого типа используется один из следующих:
  + void
  + Task
  + Task<T>
  + ValueTask<T>

Асинхронный метод, как и обычный, может использовать любое количество параметров или не использовать их вообще. Однако асинхронный метод не может определять параметры с модификаторами **out**, **ref** и **in**.

Также стоит отметить, что слово **async**, которое указывается в определении метода, НЕ делает автоматически метод асинхронным. Оно лишь указывает, что данный метод может содержать одно или несколько выражений **await**.

Рассмотрим простейший пример определения и вызова асинхронного метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | await PrintAsync(); // вызов асинхронного метода  Console.WriteLine("Некоторые действия в методе Main");    void Print()  {  Thread.Sleep(3000); // имитация продолжительной работы  Console.WriteLine("Hello METANIT.COM");  }    // определение асинхронного метода  async Task PrintAsync()  {  Console.WriteLine("Начало метода PrintAsync"); // выполняется синхронно  await Task.Run(() => Print()); // выполняется асинхронно  Console.WriteLine("Конец метода PrintAsync");  } |

Здесь прежде всего определен обычный метод Print, который просто выводит некоторую строку на консоль. Для имитации долгой работы в нем используется задержка на 3 секунд с помощью метода Thread.Sleep(). То есть условно Print - это некоторый метод, который выполняет некоторую продолжительную операцию. В реальном приложении это могло бы быть обращение к базе данных или чтение-запись файлов, но для упрощения понимания он просто выводит строку на консоль.

Также здесь определен асинхронный метод PrintAsync(). Асинхронным он является потому, что имеет в определении перед возвращаемым типом модификатор **async**, его возвращаемым типом является Task, и в теле метода определено выражение **await**.

Стоит отметить, что явным образом метод PrintAsync не возвращает никакого объекта Task, однако поскольку в теле метода применяется выражение **await**, то в качестве возвращаемого типа можно использовать тип Task.

Оператор **await** предваряет выполнение задачи, которая будет выполняться асинхронно. В данном случае подобная операция представляет выполнение метода Print:

|  |  |
| --- | --- |
| 1 | await Task.Run(()=>Print()); |

По негласным правилам в названии асинхроннных методов принято использовать суффикс **Async** - Print**Async**(), хотя в принципе это необязательно делать.

И затем в программе (в данном случае в методе Main) вызывается этот асинхронный метод.

|  |  |
| --- | --- |
| 1 | await PrintAsync(); // вызов асинхронного метода |

Посмотрим, какой у программы будет консольный вывод:

Начало метода PrintAsync

Hello METANIT.COM

Конец метода PrintAsync

Некоторые действия в методе Main

Разберем поэтапно, что здесь происходит:

1. Запускается программа, а точнее метод Main, в котором вызывается асинхронный метод PrintAsync.
2. Метод PrintAsync начинает выполняться **синхронно** вплоть до выражения await.

|  |  |
| --- | --- |
| 1 | Console.WriteLine("Начало метода PrintAsync"); // выполняется синхронно |

1. Выражение await запускает асинхронную задачу Task.Run(()=>Print())
2. Пока выполняется асинхронная задача Task.Run(()=>Print()) (а она может выполняться довольно продожительное время), выполнение кода возвращается в вызывающий метод - то есть в метод Main.
3. Когда асинхронная задача завершила свое выполнение (в случае выше - вывела строку через три секунды), продолжает работу асинхронный метод PrintAsync, который вызвал асинхронную задачу.
4. После завершения метода PrintAsync продолжает работу метод Main.

### **Асинхронный метод Main**

Стоит учитывать, что оператор **await** можно применять только в методе, который имеет модификатор **async**. И если мы в методе Main используем оператор **await**, то метод Main тоже должен быть определен как асинхронный. То есть предыдущий пример фактически будет аналогичен следующему:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class Program  {  async static Task Main(string[] args)  {  await PrintAsync(); // вызов асинхронного метода  Console.WriteLine("Некоторые действия в методе Main");      void Print()  {  Thread.Sleep(3000); // имитация продолжительной работы  Console.WriteLine("Hello METANIT.COM");  }    // определение асинхронного метода  async Task PrintAsync()  {  Console.WriteLine("Начало метода PrintAsync"); // выполняется синхронно  await Task.Run(() => Print()); // выполняется асинхронно  Console.WriteLine("Конец метода PrintAsync");  }  }  } |

### **Задержка асинхронной операции и Task.Delay**

В асинхронных методах для остановки метода на некоторое время можно применять метод **Task.Delay()**. В качестве параметра он принимает количество миллисекунд в виде значения int, либо объект TimeSpan, который задает время задержки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | await PrintAsync(); // вызов асинхронного метода  Console.WriteLine("Некоторые действия в методе Main");    // определение асинхронного метода  async Task PrintAsync()  {  await Task.Delay(3000); // имитация продолжительной работы  // или так  //await Task.Delay(TimeSpan.FromMilliseconds(3000));  Console.WriteLine("Hello METANIT.COM");  } |

Причем метод Task.Delay сам по себе представляет асинхронную операцию, поэтому к нему применяется оператор await.

### **Преимущества асинхронности**

Выше приведенные примеры являются упрощением, и вряд ли их можно считать показательным. Рассмотрим другой пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | PrintName("Tom");  PrintName("Bob");  PrintName("Sam");    void PrintName(string name)  {  Thread.Sleep(3000); // имитация продолжительной работы  Console.WriteLine(name);  } |

Данный код является синхронным и выполняет последовательно три вызова метода PrintName. Поскольку для имитации продолжительной работы в методе установлена задержка на три секунды, то общее выполнение программы займет не менее 9 секунд. Так как каждый последующий вызов PrintName будет ждать пока завершится предыдущий.

Изменим в программе синхронный метод PrintName на асинхронный:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | await PrintNameAsync("Tom");  await PrintNameAsync("Bob");  await PrintNameAsync("Sam");    // определение асинхронного метода  async Task PrintNameAsync(string name)  {  await Task.Delay(3000); // имитация продолжительной работы  Console.WriteLine(name);  } |

Вместо метода PrintName теперь вызывается три раза PrintNameAsync. Для имитации продолжительной работы в методе установлена задержка на 3 секунды с помощью вызова Task.Delay(3000). И поскольку при вызовае каждого метода применяется оператор await, который останавливает выполнение до завершения асинхронного метода, то общее выполнение программы опять же займет не менее 9 секунд. Тем не менее теперь выполнение асинхронных операций не блокирует основной поток.

Теперь оптимизируем программу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | var tomTask = PrintNameAsync("Tom");  var bobTask = PrintNameAsync("Bob");  var samTask = PrintNameAsync("Sam");    await tomTask;  await bobTask;  await samTask;  // определение асинхронного метода  async Task PrintNameAsync(string name)  {  await Task.Delay(3000); // имитация продолжительной работы  Console.WriteLine(name);  } |

В данном случае задачи фактически запускаются при определении. А оператор await применяется лишь тогда, когда нам нужно дождаться завершения асинхронных операций - то есть в конце программы. И в этом случае общее выполнение программы займет не менее 3 секунд, но гораздо меньше 9 секунд.

### **Определение асинхронного лямбда-выражения**

Асинхронную операцию можно определить не только с помощью отдельного метода, но и с помощью лямбда-выражения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // асинхронное лямбда-выражение  Func<string, Task> printer = async (message) =>  {  await Task.Delay(1000);  Console.WriteLine(message);  };    await printer("Hello World");  await printer("Hello METANIT.COM"); |

## **Возвращение результата из асинхронного метода**

В качестве возвращаемого типа в асинхронном методе должны использоваться типы **void**, **Task**, **Task<T>** или **ValueTask<T>**

### **void**

При использовании ключевого слова void асинхронный метод ничего не возвращает:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | PrintAsync("Hello World");  PrintAsync("Hello METANIT.COM");    Console.WriteLine("Main End");  await Task.Delay(3000); // ждем завершения задач    // определение асинхронного метода  async void PrintAsync(string message)  {  await Task.Delay(1000); // имитация продолжительной работы  Console.WriteLine(message);  } |

Однако асинхронных void-методов следует избегать и следует использовать только там, где эти подобные методы представляют единственный возможный способ определения асинхронного метода. Прежде всего, мы не можем применить к подобным методам оператор await. Также потому что исключения в таких методах сложно обрабатывать, так как они не могут быть перехвачены вне метода. Кроме того, подобные void-методы сложно тестировать.

Тем не менее есть ситуации, где без подобных методов не обойтись - например, при обработке событий:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | Account account = new Account();  account.Added += PrintAsync;    account.Put(500);    await Task.Delay(2000); // ждем завершения    // определение асинхронного метода  async void PrintAsync(object? obj, string message)  {  await Task.Delay(1000); // имитация продолжительной работы  Console.WriteLine(message);  }    class Account  {  int sum = 0;  public event EventHandler<string>? Added;  public void Put(int sum)  {  this.sum += sum;  Added?.Invoke(this, $"На счет поступило {sum} $");  }  } |

В данном случае событие Added в классе Account представляет делегат EventHandler, который имеет тип void. Соответственно под это событие можно определить только метод-обработчик с типом void.

### **Task**

Возвращение объекта типа Task:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | await PrintAsync("Hello Metanit.com");    // определение асинхронного метода  async Task PrintAsync(string message)  {  await Task.Delay(1000); // имитация продолжительной работы  Console.WriteLine(message);  } |

Здесь формально метод PrintAsync не использует оператор return для возвращения результата. Однако если в асинхронном методе выполняется в выражении await асинхронная операция, то мы можем возвращать из метода объект Task.

Для ожидания завершения асинхронной задачи можно применить оператор **await**. Причем его необязательно использовать непосредственно при вызове задачи. Его можно применить лишь там, где нам нужно гарантировано получить результат задачи или удостовериться, что задача завершена.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var task = PrintAsync("Hello Metanit.com"); // задача начинает выполняться  Console.WriteLine("Main Works");    await task; // ожидаем завершения задачи    // определение асинхронного метода  async Task PrintAsync(string message)  {  await Task.Delay(0);  Console.WriteLine(message);  } |

### **Task<T>**

Метод может возвращать некоторое значение. Тогда возвращаемое значение оборачивается в объект Task, а возвращаемым типом является **Task<T>**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | int n1 = await SquareAsync(5);  int n2 = await SquareAsync(6);  Console.WriteLine($"n1={n1} n2={n2}"); // n1=25 n2=36    async Task<int> SquareAsync(int n)  {  await Task.Delay(0);  return n \* n;  } |

В данном случае метод Square возвращает значение типа int - квадрат числа. Поэтому возвращаемым типом в данном случае является типа Task<int>.

Чтобы получить результат асинхронного метода применяем оператор await при вызове SquareAsync:

|  |  |
| --- | --- |
| 1 | int n1 = await SquareAsync(5); |

Подобным образом можно получать данные других типов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Person person = await GetPersonAsync("Tom");  Console.WriteLine(person.Name); // Tom  // определение асинхронного метода  async Task<Person> GetPersonAsync(string name)  {  await Task.Delay(0);  return new Person(name);  }  record class Person(string Name); |

Опять же получение непосредственных результатов асинхронной задачи можно отложить до того момента, когда они непосредственно нужны:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | var square5 = SquareAsync(5);  var square6 = SquareAsync(6);    Console.WriteLine("Остальные действия в методе Main");    int n1 = await square5;  int n2 = await square6;  Console.WriteLine($"n1={n1} n2={n2}"); // n1=25 n2=36    async Task<int> SquareAsync(int n)  {  await Task.Delay(0);  var result = n \* n;  Console.WriteLine($"Квадрат числа {n} равен {result}");  return result;  } |

Пример работы программы (ввывод не детерминирован):

Квадрат числа 5 равен 25

Квадрат числа 6 равен 36

Остальные действия в методе Main

n1=25 n2=36

### **ValueTask<T>**

Использование типа ValueTask<T> во многом аналогично применению Task<T> за исключением некоторых различий в работе с памятью, поскольку ValueTask - структура, которая содержит большее количество полей. Поэтому применение ValueTask вместо Task приводит к копированию большего количества данных и соответственно создает некоторые дополнительные издержки.

Преимуществом ValueTask перед Task является то, что данный тип позволяет избежать дополнительных выделений памяти в хипе. Например, иногда требуется синхронно возвратить некоторое значение. Так, возьмем следующий пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var result = await AddAsync(4, 5);  Console.WriteLine(result);    Task<int> AddAsync(int a, int b)  {  return Task.FromResult(a + b);  } |

Здесь метод AddAsync синхронно возвращает некоторое значение - в данном случае сумму двух чисел. С помощью статического метода **Task.FromResult** можно синхронно возвратить некоторое значение. Однако использование типа Task приведет к выделению дополнительной задачи с сопутствующими выделениями памяти в хипе. ValueTask решает эту задачу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var result = await AddAsync(4, 5);  Console.WriteLine(result);    ValueTask<int> AddAsync(int a, int b)  {  return new ValueTask<int>(a + b);  } |

В данном случае дополнительный объект Task не будет создаваться и соответственно дополнительная память не будет выделяться. Поэтому ValueTask обычно применяется, когда результат асинхронной операции уже имеется.

При необходимости также можно преобразовать ValueTask в объект Task с помощью метода **AsTask()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var getMessage = GetMessageAsync();  string message = await getMessage.AsTask();  Console.WriteLine(message); // Hello    async ValueTask<string> GetMessageAsync()  {  await Task.Delay(0);  return "Hello";  } |

## **Последовательное и параллельное выполнение. Task.WhenAll и Task.WhenAny**

Асинхронный метод может содержать множество выражений await. Когда система встречает в блоке кода оператор await, то выполнение в асинхронном методе останавливается, пока не завершится асинхронная задача. После завершения задачи управление переходит к следующему оператору await и так далее. Это позволяет вызывать асинхронные задачи последовательно в определенном порядке. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | await PrintAsync("Hello C#");  await PrintAsync("Hello World");  await PrintAsync("Hello METANIT.COM");    async Task PrintAsync(string message)  {  await Task.Delay(2000); // имитация продолжительной операции  Console.WriteLine(message);  } |

Консольный вывод данной программы:

Hello C#

Hello World

Hello METANIT.COM

То есть мы видим, что вызовы PrintAsync выполняются последовательно в том порядке, в котором они определены в коде. Каждая задача выполняется как минимум 2 секунды, соответственно общее время выполнения трех задач будет как минимум 6 секунд. И в данном случае вывод строго детерминирован.

Нередко такая последовательность бывает необходима, если одна задача зависит от результатов другой.

Однако это не всегда необходимо. В подобном случае мы можем сразу запустить все задачи параллельно и применить оператор await там, где необходимо гарантировать завершение выполнения задачи, например, в самом конце программы.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // определяем и запускаем задачи  var task1 = PrintAsync("Hello C#");  var task2 = PrintAsync("Hello World");  var task3 = PrintAsync("Hello METANIT.COM");    // ожидаем завершения задач  await task1;  await task2;  await task3;    async Task PrintAsync(string message)  {  await Task.Delay(2000); // имитация продолжительной операции  Console.WriteLine(message);  } |

В этом случае все задачи запускаются и выполняются параллельно, соответственно общее время выполнения будет меньше 6 секунд, а консольный вывод программы недетерминирован. Например, он может быть следующим:

Hello METANIT.COM

Hello C#

Hello World

Однако .NET позволяет упростить отслеживание выполнения набора задач с помощью метода **Task.WhenAll**. Этот метод принимает набор асинхронных задач и ожидает завершения всех этих задач. Этот метод является аналогом статического метода Task.WaitAll(), однако предназначен непосредственно для асинхронных методов и позволяет применять оператор await:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // определяем и запускаем задачи  var task1 = PrintAsync("Hello C#");  var task2 = PrintAsync("Hello World");  var task3 = PrintAsync("Hello METANIT.COM");    // ожидаем завершения всех задач  await Task.WhenAll(task1, task2, task3);    async Task PrintAsync(string message)  {  await Task.Delay(2000); // имитация продолжительной операции  Console.WriteLine(message);  } |

Вначале запускаются три задачи. Затем Task.WhenAll создает новую задачу, которая будет автоматически выполнена после выполнения всех предоставленных задач, то есть задач task1, task2, task3. А с помощью оператора await ожидаем ее завершения.

Если нам надо дождаться, когда будет выполнена хотя бы одна задача из некоторого набора задач, то можно применять метод **Task.WhenAny()**. Это аналог метода Task.WaitAny() - он завершает выполнение, когда завершается хотя бы одна задача. Но для ожидания выполнения к Task.WhenAny() применяется оператор await:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // определяем и запускаем задачи  var task1 = PrintAsync("Hello C#");  var task2 = PrintAsync("Hello World");  var task3 = PrintAsync("Hello METANIT.COM");    // ожидаем завершения хотя бы одной задачи  await Task.WhenAny(task1, task2, task3);    async Task PrintAsync(string message)  {  await Task.Delay(new Random().Next(1000, 2000)); // имитация продолжительной операции  Console.WriteLine(message);  } |

### **Получение результата**

Задачи, передаваемые в Task.WhenAll и Task.WhenAny, могут возвращать некоторое значение. В этом случае из методов Task.WhenAll и Task.WhenAny можно получить массив, который будет содержать результаты задач:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // определяем и запускаем задачи  var task1 = SquareAsync(4);  var task2 = SquareAsync(5);  var task3 = SquareAsync(6);    // ожидаем завершения всех задач  int[] results = await Task.WhenAll(task1, task2, task3);  // получаем результаты:  foreach (int result in results)  Console.WriteLine(result);    async Task<int> SquareAsync(int n)  {  await Task.Delay(1000);  return n \* n;  } |

В данном случае метод Square возвращает число int - квадрат передаваемого в метод числа. И переменная results будет содержать результат вызова Task.WhenAll - по сути результаты всех трех запущенных задач. Поскольку все передаваемые в Task.WhenAll задачи возвращают int, то соответственно результат Task.WhenAll будет представлять массив значений int.

Также после завершения задачи ее результат можно получить стандартным образом через свойство Result:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | // определяем и запускаем задачи  var task1 = SquareAsync(4);  var task2 = SquareAsync(5);  var task3 = SquareAsync(6);    await Task.WhenAll(task1, task2, task3);  // получаем результат задачи task2  Console.WriteLine($"task2 result: {task2.Result}"); // task2 result: 25    async Task<int> SquareAsync(int n)  {  await Task.Delay(1000);  return n \* n;  } |

## **Обработка ошибок в асинхронных методах**

Обработка ошибок в асинхронных методах, использующих ключевые слова async и await, имеет свои особенности.

Для обработки ошибок выражение **await** помещается в блок try:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | try  {  await PrintAsync("Hello METANIT.COM");  await PrintAsync("Hi");  }  catch (Exception ex)  {  Console.WriteLine(ex.Message);  }    async Task PrintAsync(string message)  {  // если длина строки меньше 3 символов, генерируем исключение  if (message.Length < 3)  throw new ArgumentException($"Invalid string length: {message.Length}");  await Task.Delay(100); // имитация продолжительной операции  Console.WriteLine(message);  } |

В данном случае асинхронный метод PrintAsync генерирует исключение ArgumentException, если методу передается строка с длиной меньше 3 символов.

Для обработки исключения в методе Main выражение await помещено в блок try. В итоге при выполнении вызова await PrintAsync("Hi") будет сгенерировано исключение, что привет к генерации исключения. Однако программа не остановит аварийно свою работу, а обработает исключение и продолжит дальнейшие вычисления.

Консольный вывод программы:

Hello METANIT.COM

Invalid string length: 2

Следует учитывать, что если асинхронный метод имеет тип void, то в этом случае исключение во вне не передается, соответственно мы не сможем обработать исключение при вызове метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | try  {  PrintAsync("Hello METANIT.COM");  PrintAsync("Hi"); // здесь программа сгенерирует исключение и аварийно остановится  await Task.Delay(1000); // ждем завершения задач  }  catch (Exception ex) // исключение НЕ будет обработано  {  Console.WriteLine(ex.Message);  }    async void PrintAsync(string message)  {  // если длина строки меньше 3 символов, генерируем исключение  if (message.Length < 3)  throw new ArgumentException($"Invalid string length: {message.Length}");  await Task.Delay(100); // имитация продолжительной операции  Console.WriteLine(message);  } |

В данном случае, не смотря на то, что асинхронные методы вызываются в блоке try, исключение не будет перехвачено и обработано. В этом один из минусов применения асинхронных void-методов. Правда, в этом случае мы можем определить обработку исключения в самом асинхронном методе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | PrintAsync("Hello METANIT.COM");  PrintAsync("Hi");  await Task.Delay(1000); // ждем завершения задач    async void PrintAsync(string message)  {  try  {  // если длина строки меньше 3 символов, генерируем исключение  if (message.Length < 3)  throw new ArgumentException($"Invalid string length: {message.Length}");  await Task.Delay(100); // имитация продолжительной операции  Console.WriteLine(message);  }  catch (Exception ex)  {  Console.WriteLine(ex.Message);  }    } |

### **Исследование исключения**

При возникновении ошибки у объекта Task, представляющего асинхронную задачу, в которой произошла ошибка, свойство **IsFaulted** имеет значение true. Кроме того, свойство Exception объекта Task содержит всю информацию об ошибке. Проинспектируем данное свойство:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | var task = PrintAsync("Hi");  try  {  await task;  }  catch  {  Console.WriteLine(task.Exception?.InnerException?.Message); // Invalid string length: 2  Console.WriteLine($"IsFaulted: {task.IsFaulted}"); // IsFaulted: True  Console.WriteLine($"Status: {task.Status}"); // Status: Faulted  }    async Task PrintAsync(string message)  {  // если длина строки меньше 3 символов, генерируем исключение  if (message.Length < 3)  throw new ArgumentException($"Invalid string length: {message.Length}");  await Task.Delay(1000); // имитация продолжительной операции  Console.WriteLine(message);  } |

И если мы передадим в метод строку с длиной меньше 3 символов, то task.IsFaulted будет равно true.

### **Обработка нескольких исключений. WhenAll**

Если мы ожидаем выполнения сразу нескольких задач, например, с помощью Task.WhenAll, то мы можем получить сразу несколько исключений одномоментно для каждой выполняемой задачи. В этом случае мы можем получить все исключения из свойства Exception.InnerExceptions:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | // определяем и запускаем задачи  var task1 = PrintAsync("H");  var task2 = PrintAsync("Hi");  var allTasks = Task.WhenAll(task1, task2);  try  {  await allTasks;  }  catch (Exception ex)  {  Console.WriteLine($"Exception: {ex.Message}");  Console.WriteLine($"IsFaulted: {allTasks.IsFaulted}");  if(allTasks.Exception is not null)  {  foreach (var exception in allTasks.Exception.InnerExceptions)  {  Console.WriteLine($"InnerException: {exception.Message}");  }  }  }    async Task PrintAsync(string message)  {  // если длина строки меньше 3 символов, генерируем исключение  if (message.Length < 3)  throw new ArgumentException($"Invalid string: {message}");  await Task.Delay(1000); // имитация продолжительной операции  Console.WriteLine(message);  } |

Здесь в два вызова метода PrintAsync передаются заведомо некорректные значения. Таким образом, при обоих вызовах будет сгенерирована ошибка.

Хотя блок catch через переменную Exception ex будет получать одно перехваченное исключение, но с помощью коллекции Exception.InnerExceptions мы сможем получить информацию обо всех возникших исключениях.

В итоге при выполнении этого метода мы получим следующий консольный вывод:

Exception: Invalid string: H

IsFaulted: True

InnerException: Invalid string: H

InnerException: Invalid string: Hi

## **Асинхронные стримы**

Начиная с версии C# 8.0 в C# были добавлены **асинхронные стримы**, которые упрощают работу с потоками данных в асинхронном режиме. Хотя асинхронность в C# существует уже довольно давно, тем не менее асинхронные методы до сих пор позволяли получать один объект, когда асинхронная операция была готова предоставить результат. Для возвращения нескольких значений в C# могут применяться итераторы, но они имеют синхронную природу, блокируют вызывающий поток и не могут использоваться в асинхронном контексте. Асинхронные стримы обходят эту проблему, позволяя получать множество значений и возвращать их по мере готовности в асинхронном режиме.

По сути асинхронный стрим представляет метод, который обладает тремя характеристиками:

* метод имеет модификатор async
* метод возвращает объект **IAsyncEnumerable<T>**. Интерфейс IAsyncEnumerable определяет метод GetAsyncEnumerator, который возвращает IAsyncEnumerator:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public interface IAsyncEnumerable<out T>  {  IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken cancellationToken = default);  }    public interface IAsyncEnumerator<out T> : IAsyncDisposable  {  T Current { get; }  ValueTask<bool> MoveNextAsync();  }  public interface IAsyncDisposable  {  ValueTask DisposeAsync();  } |

* метод содержит выражения yield return для последовательного получения элементов из асинхронного стрима

Фактически асинхронный стрим объединяет асинхронность и итераторы. Рассмотрим простейший пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | await foreach (var number in GetNumbersAsync())  {  Console.WriteLine(number);  }    async IAsyncEnumerable<int> GetNumbersAsync()  {  for (int i = 0; i < 10; i++)  {  await Task.Delay(100);  yield return i;  }  } |

Итак, метод GetNumbersAsync() фактически и представляет асинхронный стрим. Этот метод является асинхронным. Его возвращаемый тип - IAsyncEnumerable<int>. А его суть сводится к тому, что он возвращает с помощью yield return каждые 100 миллисекунд некоторое число. То есть фактически метод должен вернуть 10 чисел от 0 до 9 с промежутком в 100 миллисекунд.

Для получения данных из стрима в методе Main используется цикл foreach:

|  |  |
| --- | --- |
| 1 | await foreach (var number in GetNumbersAsync()) |

Важно отметить, что он предваряется оператором **await**. В итоге, каждый раз когда асинхронный стрим будет возвращать очередное число, цикл будет его получать и выводить на консоль.

Где можно применять асинхронные стримы? Асинхронные стримы могут применяться для получения данных из какого-нибудь внешнего хранилища. Например, пусть имеется следующий класс некоторого хранилища:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Repository  {  string[] data = { "Tom", "Sam", "Kate", "Alice", "Bob" };  public async IAsyncEnumerable<string> GetDataAsync()  {  for (int i = 0; i < data.Length; i++)  {  Console.WriteLine($"Получаем {i + 1} элемент");  await Task.Delay(500);  yield return data[i];  }  }  } |

Для упрощения примера данные здесь представлены в виде простого внутреннего массива строк. Для имитации задержки в получении применяется метод Task.Delay.

Получим эти данные в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Repository repo = new Repository();  IAsyncEnumerable<string> data = repo.GetDataAsync();  await foreach (var name in data)  {  Console.WriteLine(name);  }    class Repository  {  string[] data = { "Tom", "Sam", "Kate", "Alice", "Bob" };  public async IAsyncEnumerable<string> GetDataAsync()  {  for (int i = 0; i < data.Length; i++)  {  Console.WriteLine($"Получаем {i + 1} элемент");  await Task.Delay(500);  yield return data[i];  }  }  } |