Оглавление

[Оглавление 1](#_Toc172731287)

[Основы 11](#_Toc172731288)

[Литералы 11](#_Toc172731289)

[Типы данных 11](#_Toc172731290)

[Консольный ввод-вывод 11](#_Toc172731291)

[Логические операции 12](#_Toc172731292)

[Операции присваивания 12](#_Toc172731293)

[Массивы 12](#_Toc172731294)

[Одномерные массивы 12](#_Toc172731295)

[Многомерные массивы 13](#_Toc172731296)

[Массив массивов 13](#_Toc172731297)

[Методы 13](#_Toc172731298)

[Передача параметров по ссылке. Выходные параметры 14](#_Toc172731299)

[Модификатор ref 14](#_Toc172731300)

[Модификатор out 14](#_Toc172731301)

[Модификатор in 14](#_Toc172731302)

[Модификатор ref readonly (c C#12) 14](#_Toc172731303)

[Массив параметров и ключевое слово params 15](#_Toc172731304)

[Локальные функции 15](#_Toc172731305)

[Конструкция switch 15](#_Toc172731306)

[Перечисления enum 16](#_Toc172731307)

[Классы 17](#_Toc172731308)

[Создание конструкторов 17](#_Toc172731309)

[Цепочка вызовов конструкторов 17](#_Toc172731310)

[Первичные конструкторы (начиная с версии C#12) 17](#_Toc172731311)

[Инициализаторы объектов 17](#_Toc172731312)

[Деконструкторы 17](#_Toc172731313)

[Структуры 18](#_Toc172731314)

[Инициализация с помощью конструктора 18](#_Toc172731315)

[Цепочка конструкторов 18](#_Toc172731316)

[Первичный конструктор 19](#_Toc172731317)

[Инициализатор структуры 19](#_Toc172731318)

[Копирование структуры с помощью with 19](#_Toc172731319)

[Модификаторы доступа 19](#_Toc172731320)

[Свойства 20](#_Toc172731321)

[Определение свойств 20](#_Toc172731322)

[Свойства только для чтения и записи 20](#_Toc172731323)

[Модификаторы доступа 20](#_Toc172731324)

[Автоматические свойства 20](#_Toc172731325)

[Блок init (начиная с версии C# 9.0) 21](#_Toc172731326)

[Сокращенная запись свойств 21](#_Toc172731327)

[Модификатор required (добавлен в C# 11) 21](#_Toc172731328)

[Статические члены и модификатор static 22](#_Toc172731329)

[Статические поля 22](#_Toc172731330)

[Статические свойства 22](#_Toc172731331)

[Статические методы 22](#_Toc172731332)

[Статический конструктор 22](#_Toc172731333)

[Статические классы 22](#_Toc172731334)

[Константы, поля и структуры для чтения 22](#_Toc172731335)

[Константы класса 22](#_Toc172731336)

[Поля для чтения и модификатор readonly 23](#_Toc172731337)

[Сравнение констант 23](#_Toc172731338)

[Структуры для чтения 23](#_Toc172731339)

[Null и ссылочные типы 23](#_Toc172731340)

[Оператор ! (null-forgiving operator) 23](#_Toc172731341)

[Null и значимые типы 23](#_Toc172731342)

[Проверка на null, операторы ?. и ?? 24](#_Toc172731343)

[Оператор ?? 24](#_Toc172731344)

[Оператор условного null ( ?. ) 24](#_Toc172731345)

[Псевдонимы типов и статический импорт 24](#_Toc172731346)

[Псевдонимы 24](#_Toc172731347)

[Статический импорт 24](#_Toc172731348)

[ООП 25](#_Toc172731349)

[Наследование 25](#_Toc172731350)

[Ключевое слово base 25](#_Toc172731351)

[Преобразование типов 26](#_Toc172731352)

[Восоходящее преобразование. Upcasting 26](#_Toc172731353)

[Нисходящее преобразование. Downcasting 26](#_Toc172731354)

[Способы преобразований 26](#_Toc172731355)

[Виртуальные методы и свойства 26](#_Toc172731356)

[Ключевое слово base 27](#_Toc172731357)

[Переопределение свойств 27](#_Toc172731358)

[Запрет переопределения методов 27](#_Toc172731359)

[Скрытие методов и свойств 27](#_Toc172731360)

[Различие переопределения и скрытия методов 28](#_Toc172731361)

[Переопределение 28](#_Toc172731362)

[Скрытие 28](#_Toc172731363)

[Абстрактные классы и члены классов 28](#_Toc172731364)

[Класс System.Object и его методы 29](#_Toc172731365)

[Обобщения (generics) 29](#_Toc172731366)

[Статические поля обобщенных классов 29](#_Toc172731367)

[Обобщенные методы 29](#_Toc172731368)

[Ограничения обобщений 29](#_Toc172731369)

[Ограничения методов 29](#_Toc172731370)

[Ограничения обобщений в типах 30](#_Toc172731371)

[Типы ограничений и стандартные ограничения 30](#_Toc172731372)

[Использование нескольких универсальных параметров 30](#_Toc172731373)

[Наследование обобщенных типов 30](#_Toc172731374)

[Исключения 32](#_Toc172731375)

[Конструкция try..catch..finally 32](#_Toc172731376)

[Фильтры исключений 32](#_Toc172731377)

[Типы исключений. Класс Exception 32](#_Toc172731378)

[Генерация исключения и оператор throw 33](#_Toc172731379)

[Делегаты, лямбды и события 34](#_Toc172731380)

[Делегаты 34](#_Toc172731381)

[Добавление методов в делегат 34](#_Toc172731382)

[Объединение делегатов 34](#_Toc172731383)

[Вызов делегата с помощью метода Invoke() 35](#_Toc172731384)

[Обобщенные делегаты (Generic) 35](#_Toc172731385)

[Делегаты как параметры методов 35](#_Toc172731386)

[Возвращение делегатов из метода 35](#_Toc172731387)

[Анонимные методы 35](#_Toc172731388)

[Лямбды 36](#_Toc172731389)

[Параметры лямбды 37](#_Toc172731390)

[Возвращение результата 37](#_Toc172731391)

[Добавление и удаление действий в лямбда-выражении, метод Invoke 37](#_Toc172731392)

[Лямбда-выражение как аргумент метода 38](#_Toc172731393)

[Лямбда-выражение как результ метода 38](#_Toc172731394)

[События 38](#_Toc172731395)

[Определение и вызов событий 38](#_Toc172731396)

[Добавление обработчика события 38](#_Toc172731397)

[Управление обработчиками (add, remove) 39](#_Toc172731398)

[Ковариантность и контравариантность делегатов 39](#_Toc172731399)

[Ковариантность 40](#_Toc172731400)

[Контрвариантность 40](#_Toc172731401)

[Ковариантность и контравариантность в обобщенных делегатах 40](#_Toc172731402)

[Делегаты Action, Predicate и Func 40](#_Toc172731403)

[Action 40](#_Toc172731404)

[Predicate 41](#_Toc172731405)

[Func 41](#_Toc172731406)

[Замыкания 41](#_Toc172731407)

[Реализация с помощью лямбда-выражений 42](#_Toc172731408)

[Применение параметров 42](#_Toc172731409)

[Интерфейсы 43](#_Toc172731410)

[Определение интерфейсов 43](#_Toc172731411)

[Применение интерфейсов 43](#_Toc172731412)

[Множественная реализация интерфейсов 45](#_Toc172731413)

[Явная реализация интерфейсов 45](#_Toc172731414)

[Модификаторы доступа 45](#_Toc172731415)

[Реализация интерфейсов в базовых и производных классах 46](#_Toc172731416)

[Наследование интерфейсов 48](#_Toc172731417)

[Интерфейсы как ограничения обобщений 48](#_Toc172731418)

[Обобщенные интерфейсы 49](#_Toc172731419)

[Копирование объектов. Интерфейс ICloneable 49](#_Toc172731420)

[Поверхностное копирование 49](#_Toc172731421)

[Глубокое копирование 49](#_Toc172731422)

[Сортировка объектов. Интерфейс IComparable 49](#_Toc172731423)

[Применение компаратора 50](#_Toc172731424)

[Ковариантность и контравариантность обобщенных интерфейсов 51](#_Toc172731425)

[Ковариантные интерфейсы 51](#_Toc172731426)

[Контравариантные интерфейсы 51](#_Toc172731427)

[Дополнительные возможности ООП 52](#_Toc172731428)

[Определение операторов 52](#_Toc172731429)

[Определение инкремента и декремента 53](#_Toc172731430)

[Определение операций true и false 53](#_Toc172731431)

[Перегрузка операций преобразования типов 53](#_Toc172731432)

[Индексаторы 54](#_Toc172731433)

[Переменные-ссылки и возвращение ссылки 54](#_Toc172731434)

[Переменная-ссылка 54](#_Toc172731435)

[Ссылка как результат функции 55](#_Toc172731436)

[Методы расширения 55](#_Toc172731437)

[Частичные классы и методы 55](#_Toc172731438)

[Частичные методы 56](#_Toc172731439)

[Анонимные типы 56](#_Toc172731440)

[Инициализаторы с проекцией 56](#_Toc172731441)

[Кортежи 57](#_Toc172731442)

[Кортеж как результат метода 57](#_Toc172731443)

[Кортеж как параметр метода 57](#_Toc172731444)

[Records (C#9) 57](#_Toc172731445)

[Сравнение records 58](#_Toc172731446)

[Оператор with 58](#_Toc172731447)

[Позиционные классы records 58](#_Toc172731448)

[Позиционные структуры для чтения 59](#_Toc172731449)

[ToString 59](#_Toc172731450)

[Наследование 59](#_Toc172731451)

[Pattern matching 60](#_Toc172731452)

[Паттерн типов 60](#_Toc172731453)

[type pattern 60](#_Toc172731454)

[constant pattern 60](#_Toc172731455)

[в конструкции switch: 60](#_Toc172731456)

[выражение when в конструкции switch: 60](#_Toc172731457)

[Паттерн свойств 60](#_Toc172731458)

[Паттерны кортежей 61](#_Toc172731459)

[Позиционный паттерн 61](#_Toc172731460)

[Реляционный и логический паттерны (C#9.0) 62](#_Toc172731461)

[relational pattern 62](#_Toc172731462)

[logical pattern 62](#_Toc172731463)

[Паттерны списков (C#11) 62](#_Toc172731464)

[Свойства коллекций 63](#_Toc172731465)

[Коллекции 64](#_Toc172731466)

[Список List<T> 64](#_Toc172731467)

[Создание пустого списка 64](#_Toc172731468)

[Создание списка, содежащего начальные значения 64](#_Toc172731469)

[Создание списка с элементами из другой коллекции 64](#_Toc172731470)

[С C#12 64](#_Toc172731471)

[Установка начальной емкости списка 64](#_Toc172731472)

[Обращение к элементам списка 64](#_Toc172731473)

[Длина списка 64](#_Toc172731474)

[Методы списка 64](#_Toc172731475)

[Двухсвязный список LinkedList<T> 66](#_Toc172731476)

[Создание связанного списка 66](#_Toc172731477)

[Свойства LinkedList 66](#_Toc172731478)

[Свойства LinkedListNode 66](#_Toc172731479)

[Циклы: 66](#_Toc172731480)

[Методы LinkedList 66](#_Toc172731481)

[Очередь Queue 67](#_Toc172731482)

[Создание очереди 67](#_Toc172731483)

[Методы Queue 67](#_Toc172731484)

[Коллекция Stack<T> 67](#_Toc172731485)

[Создание стека 67](#_Toc172731486)

[Методы Stack 68](#_Toc172731487)

[Коллекция Dictionary<K, V> 68](#_Toc172731488)

[Создание и инициализация словаря 68](#_Toc172731489)

[KeyValuePair 68](#_Toc172731490)

[Перебор словаря 69](#_Toc172731491)

[Получение элементов 69](#_Toc172731492)

[Методы и свойства Dictionary 69](#_Toc172731493)

[Класс ObservableCollection 70](#_Toc172731494)

[Создание и инициализация ObservableCollection 70](#_Toc172731495)

[Обращение к элементам коллекции 70](#_Toc172731496)

[Перебор коллекции 70](#_Toc172731497)

[Методы и свойства ObservableCollection 70](#_Toc172731498)

[Уведомление об измении коллекции 71](#_Toc172731499)

[Пример 71](#_Toc172731500)

[Интерфейсы IEnumerable и IEnumerator 72](#_Toc172731501)

[Итераторы и оператор yield 72](#_Toc172731502)

[Пример с расширением Int32 72](#_Toc172731503)

[Именованный итератор 72](#_Toc172731504)

[Класс Array и массивы 73](#_Toc172731505)

[Свойства 73](#_Toc172731506)

[Методы 73](#_Toc172731507)

[Span 74](#_Toc172731508)

[Создание Span 74](#_Toc172731509)

[Например 74](#_Toc172731510)

[Методы Span 74](#_Toc172731511)

[ReadOnlySpan 75](#_Toc172731512)

[Ограничения Span 75](#_Toc172731513)

[Индексы и диапазоны 75](#_Toc172731514)

[Индексы 75](#_Toc172731515)

[Диапазоны 75](#_Toc172731516)

[Работа со строками 76](#_Toc172731517)

[Строки и класс String 76](#_Toc172731518)

[Создание строк 76](#_Toc172731519)

[Строка как набор символов 76](#_Toc172731520)

[Перебор строк 76](#_Toc172731521)

[Сравнение строк 76](#_Toc172731522)

[Многострочные строки 76](#_Toc172731523)

[C C# 11 76](#_Toc172731524)

[Основные методы строк 76](#_Toc172731525)

[Операции со строками 77](#_Toc172731526)

[Объединение строк 77](#_Toc172731527)

[Сравнение строк 77](#_Toc172731528)

[Поиск в строке 77](#_Toc172731529)

[Разделение строк 78](#_Toc172731530)

[Обрезка строки 78](#_Toc172731531)

[Вставка 78](#_Toc172731532)

[Удаление строк 78](#_Toc172731533)

[Замена 78](#_Toc172731534)

[Смена регистра 78](#_Toc172731535)

[Форматирование и интерполяция строк 78](#_Toc172731536)

[Форматирование строк 78](#_Toc172731537)

[Спецификаторы форматирования 78](#_Toc172731538)

[Метод ToString 79](#_Toc172731539)

[Интерполяция строк 79](#_Toc172731540)

[Класс StringBuilder 79](#_Toc172731541)

[Создание StringBuilder 79](#_Toc172731542)

[Операции со строками в StringBuilder 80](#_Toc172731543)

[Когда надо использовать класс String, а когда StringBuilder? 80](#_Toc172731544)

[Регулярные выражения 80](#_Toc172731545)

[Параметр RegexOptions 81](#_Toc172731546)

[Синтаксис регулярных выражений 81](#_Toc172731547)

[Проверка на соответствие строки формату (IsMatch) 82](#_Toc172731548)

[Замена и метод Replace 82](#_Toc172731549)

[Работа с датами и временем 83](#_Toc172731550)

[Структура DateTime 83](#_Toc172731551)

[Операции с DateTime 83](#_Toc172731552)

[Форматирование дат и времени 84](#_Toc172731553)

[Настройка формата времени и даты 85](#_Toc172731554)

[DateOnly и TimeOnly 86](#_Toc172731555)

[DateOnly 86](#_Toc172731556)

[TimeOnly 87](#_Toc172731557)

[Дополнительные классы и структуры .NET 88](#_Toc172731558)

[Отложенная инициализация и тип Lazy 88](#_Toc172731559)

[Математические вычисления и класс Math 89](#_Toc172731560)

[Основные методы класса Math 89](#_Toc172731561)

[Константы 90](#_Toc172731562)

[Преобразование типов и класс Convert 90](#_Toc172731563)

[Метод Parse 90](#_Toc172731564)

[Метод TryParse 91](#_Toc172731565)

[Convert 91](#_Toc172731566)

[Многопоточность 92](#_Toc172731567)

[Введение в многопоточность. 92](#_Toc172731568)

[Класс Thread 92](#_Toc172731569)

[Свойства класса Thread 92](#_Toc172731570)

[Методы класса Thread 93](#_Toc172731571)

[Создание потоков. Делегат ThreadStart 93](#_Toc172731572)

[Потоки с параметрами и ParameterizedThreadStart 94](#_Toc172731573)

[Синхронизация потоков 94](#_Toc172731574)

[Мониторы 95](#_Toc172731575)

[Методы 95](#_Toc172731576)

[Пример 95](#_Toc172731577)

[Класс AutoResetEvent 96](#_Toc172731578)

[Методы 96](#_Toc172731579)

[Пример 96](#_Toc172731580)

[Мьютексы 96](#_Toc172731581)

[Семафоры 97](#_Toc172731582)

[Конструкторы 97](#_Toc172731583)

[Методы 97](#_Toc172731584)

[Пример 97](#_Toc172731585)

[Параллельное программирование и библиотека TPL (Task Parallel Library) 98](#_Toc172731586)

[Задачи и класс Task 98](#_Toc172731587)

[Ожидание завершения задачи 98](#_Toc172731588)

[Синхронный запуск задачи 98](#_Toc172731589)

[Свойства класса Task 98](#_Toc172731590)

[Работа с классом Task 99](#_Toc172731591)

[Вложенные задачи 99](#_Toc172731592)

[Массив задач 99](#_Toc172731593)

[Возвращение результатов из задач 100](#_Toc172731594)

[Задачи продолжения (continuation task) 100](#_Toc172731595)

[Класс Parallel 100](#_Toc172731596)

[Методы 100](#_Toc172731597)

[Выход из цикла 101](#_Toc172731598)

[Свойства ParallelLoopResult 101](#_Toc172731599)

[Отмена задач и параллельных операций. CancellationToken 101](#_Toc172731600)

[Общий алгоритм отмены задачи 102](#_Toc172731601)

[Мягкий выход из задачи без исключения OperationCanceledException 102](#_Toc172731602)

[Отмена задачи с помощью генерации исключения 103](#_Toc172731603)

[Регистрация обработчика отмены задачи 103](#_Toc172731604)

[Передача токена во внешний метод 103](#_Toc172731605)

[Отмена параллельных операций Parallel 104](#_Toc172731606)

[Aсинхронное программирование 105](#_Toc172731607)

[Асинхронные методы, async и await 105](#_Toc172731608)

[Пример 105](#_Toc172731609)

[Асинхронный метод Main 106](#_Toc172731610)

[Задержка асинхронной операции и Task.Delay 106](#_Toc172731611)

[Преимущества асинхронности 106](#_Toc172731612)

[Определение асинхронного лямбда-выражения 107](#_Toc172731613)

[Возвращение результата из асинхронного метода 107](#_Toc172731614)

[void 107](#_Toc172731615)

[Task 107](#_Toc172731616)

[Task<T> 107](#_Toc172731617)

[ValueTask<T> 107](#_Toc172731618)

[Последовательное и параллельное выполнение. Task.WhenAll и Task.WhenAny 108](#_Toc172731619)

[Последовательное выполнение: 108](#_Toc172731620)

[Параллельное выполнение: 108](#_Toc172731621)

[Task.WhenAll 108](#_Toc172731622)

[Task.WhenAny 109](#_Toc172731623)

[Получение результата 109](#_Toc172731624)

[Обработка ошибок в асинхронных методах 109](#_Toc172731625)

[Исследование исключения 109](#_Toc172731626)

[Обработка нескольких исключений. WhenAll 110](#_Toc172731627)

[Асинхронные стримы с C# 8.0 110](#_Toc172731628)

[Пример 111](#_Toc172731629)

[LINQ 112](#_Toc172731630)

[Основы LINQ 112](#_Toc172731631)

[Операторы запросов LINQ 112](#_Toc172731632)

[Методы расширения LINQ 112](#_Toc172731633)

[Проекция данных (Select, let, SelectMany) 114](#_Toc172731634)

[Переменые в запросах и оператор let 114](#_Toc172731635)

[Выборка из нескольких источников 114](#_Toc172731636)

[SelectMany и сведение объектов 115](#_Toc172731637)

[Фильтрация коллекции 115](#_Toc172731638)

[Сложные фильтры 115](#_Toc172731639)

[Фильтрация по типу данных 116](#_Toc172731640)

[Сортировка 116](#_Toc172731641)

[Оператор orderby и метод OrderBy 116](#_Toc172731642)

[Сортировка по убыванию 116](#_Toc172731643)

[Множественные критерии сортировки 117](#_Toc172731644)

[Переопределение критерия сортировки 117](#_Toc172731645)

[Объединение, пересечение и разность коллекций 117](#_Toc172731646)

[Разность последовательностей (Except) 117](#_Toc172731647)

[Пересечение последовательностей (Intersect) 117](#_Toc172731648)

[Удаление дубликатов 118](#_Toc172731649)

[Объединение последовательностей (Union, Concat) 118](#_Toc172731650)

[Работа со сложными объектами 118](#_Toc172731651)

[Агрегатные операции 118](#_Toc172731652)

[Метод Aggregate 118](#_Toc172731653)

[Получение размера выборки (Count) 118](#_Toc172731654)

[Получение суммы (Sum) 118](#_Toc172731655)

[Максимальное, минимальное и среднее значения (Max, Min, Average) 119](#_Toc172731656)

[Методы Skip и Take 119](#_Toc172731657)

[Skip и SkipLast 119](#_Toc172731658)

[SkipWhile 119](#_Toc172731659)

[Take и TakeLast 119](#_Toc172731660)

[TakeWhile 119](#_Toc172731661)

[Группировка 120](#_Toc172731662)

[Оператор group by 120](#_Toc172731663)

[GroupBy 120](#_Toc172731664)

[Создание нового объекта при группировке 120](#_Toc172731665)

[Вложенные запросы 120](#_Toc172731666)

[Соединение коллекций 121](#_Toc172731667)

[Оператор join 121](#_Toc172731668)

[Метод Join 121](#_Toc172731669)

[GroupJoin 122](#_Toc172731670)

[Метод Zip 123](#_Toc172731671)

[Проверка наличия и получение элементов 123](#_Toc172731672)

[All 123](#_Toc172731673)

[Any 123](#_Toc172731674)

[Contains 123](#_Toc172731675)

[First 124](#_Toc172731676)

[FirstOrDefault 124](#_Toc172731677)

[Last 124](#_Toc172731678)

[LastOrDefault 124](#_Toc172731679)

[Отложенное и немедленное выполнение LINQ 125](#_Toc172731680)

[Отложенное (deferred) выполнение 125](#_Toc172731681)

[Немедленное (immediate) выполнение 126](#_Toc172731682)

[Делегаты в запросах LINQ 126](#_Toc172731683)

[Parallel LINQ 127](#_Toc172731684)

[Введение в Parallel LINQ. Метод AsParallel 127](#_Toc172731685)

[Метод AsParallel 127](#_Toc172731686)

[Метод ForAll 127](#_Toc172731687)

[Метод AsOrdered 127](#_Toc172731688)

[Обработка ошибок и отмена операции 128](#_Toc172731689)

[Прерывание параллельной операции 128](#_Toc172731690)

[Рефлексия 130](#_Toc172731691)

[Введение в рефлексию. Класс System.Type 130](#_Toc172731692)

[Получение типа 131](#_Toc172731693)

[Поиск реализованных интерфейсов 131](#_Toc172731694)

[Применение рефлексии и исследование типов 132](#_Toc172731695)

[Получение всех компонентов типа 132](#_Toc172731696)

[BindingFlags 133](#_Toc172731697)

[Получение одного компонента по имени 133](#_Toc172731698)

[Исследование методов и конструкторов с помощью рефлексии 134](#_Toc172731699)

[Получение информации о методах 134](#_Toc172731700)

[BindingFlags 135](#_Toc172731701)

[Исследование параметров 136](#_Toc172731702)

[Вызов методов 136](#_Toc172731703)

[Получение конструкторов 137](#_Toc172731704)

[Исследование полей и свойств с помощью рефлексии 138](#_Toc172731705)

[Получение информации о полях 138](#_Toc172731706)

[Получение и изменение значения поля 138](#_Toc172731707)

[Свойства 139](#_Toc172731708)

[Динамическая загрузка сборок и позднее связывание 139](#_Toc172731709)

[Позднее связывание 140](#_Toc172731710)

[Атрибуты в .NET 141](#_Toc172731711)

[Ограничение применения атрибута 141](#_Toc172731712)

[Dynamic Language Runtime 143](#_Toc172731713)

[DLR в C#. Ключевое слово dynamic 143](#_Toc172731714)

[DynamicObject и ExpandoObject 143](#_Toc172731715)

[ExpandoObject 143](#_Toc172731716)

[DynamicObject 143](#_Toc172731717)

[Использование IronPython в .NET 145](#_Toc172731718)

[ScriptScope 146](#_Toc172731719)

[Вызов функций из IronPython 146](#_Toc172731720)

[Сборка мусора, управление памятью и указатели 147](#_Toc172731721)

[Сборщик мусора в C# 147](#_Toc172731722)

[Класс System.GC 147](#_Toc172731723)

[Финализируемые объекты 148](#_Toc172731724)

[Создание деструкторов 148](#_Toc172731725)

[Интерфейс IDisposable 149](#_Toc172731726)

[Комбинирование подходов 149](#_Toc172731727)

[Общие рекомендации по использованию Finalize и Dispose 150](#_Toc172731728)

[Конструкция using 150](#_Toc172731729)

[Освобождение множества ресурсов 151](#_Toc172731730)

Основы

## Литералы

0b11 **-** 3 число в двоичной форме

0x0A **-** 10 число в шестнадцатеричной форме

3.2e3 **(**3.2E3**)** **-** 3.2 **\*** 10**^**3

'\x78' **-** x символ в ASCII

'\u0420' **-** P символ в Unicode

## Типы данных

**bool: хранит значение true или false (логические литералы).**

**byte: хранит целое число от 0 до 255 и занимает 1 байт.**

**sbyte: хранит целое число от -128 до 127 и занимает 1 байт.**

**short: хранит целое число от -32768 до 32767 и занимает 2 байта.**

**ushort: хранит целое число от 0 до 65535 и занимает 2 байта.**

**int: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта.**

**uint: хранит целое число от 0 до 4294967295 и занимает 4 байта. Суффикс u (U) - 10U**

**long: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Суффикс l (L) - 10L**

**ulong: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Суффикс ul (UL) - 10L**

**float: хранит число с плавающей точкой от -3.4e38 до 3.4e38 и занимает 4 байта. Суффикс f (F) - 3.14F**

**double: хранит число с плавающей точкой от ±5.0e-324 до ±1.7e308 и занимает 8 байта.**

**decimal: хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от ±1.0\*10-28 до ±7.9228\*1028, может хранить 28 знаков после запятой и занимает 16 байт. Суффикс m (M) - 3.14M**

**char: хранит одиночный символ в кодировке Unicode и занимает 2 байта.**

**string: хранит набор символов Unicode.**

**object: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе.**

## Консольный ввод-вывод

**Convert.ToInt32() (преобразует к типу int)**

**Convert.ToDouble() (преобразует к типу double)**

**Convert.ToDecimal() (преобразует к типу decimal)**

**Console.Write("Hello"); // Hello**

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

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

**Console.WriteLine($"Age = {age} years old, height = {height} meters");**

**Console.WriteLine("Age = {0} years old, height = {1} meters", age, height);**

## Логические операции

int x **=** 2**;** // 010

int y **=** 5**;** //101

x**&**y **-** 0 // 000 AND

x**|**y **-** 7 // 111 OR

x**^**y **-** 7 // 111 XOR

**~**x **-** **-**2 // 1...010 старший бит равен 1

x**<<**1 **-** 4 // 100

x**>>**1 **-** 1 // 001

## Операции присваивания

int a**,** b**,** c**;**

a **=** b **=** c **=** 34**;**

**+=**

**-=**

**\*=**

**/=**

**%=**

**&=**

**|=**

**^=**

**>>=**

**<<=**

## Массивы

### Одномерные массивы

int**[]** nums **=** **new** int**[**4**];**

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

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

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

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

c C#12

int**[]** nums **=** **[** 1**,** 2**,** 3**,** 5 **];**

int**[]** nums **=** **[];** // пустой массив

nums**[**0**]** **-** 1

nums**[^**1**]** **-** 5m

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

**{**

Console**.**WriteLine**(**nums**[**i**]);**

**}**

**foreach(**var n **in** nums**)**

Console**.**WriteLine**(**n**);**

### Многомерные массивы

int**[,]** nums**;**

int**[,]** nums **=** **new** int**[**2**,** 3**];**

int**[,]** nums **=** **new** int**[**2**,** 3**]** **{** **{** 0**,** 1**,** 2 **},** **{** 3**,** 4**,** 5 **}** **};**

int**[,]** nums **=** **new** int**[,]** **{** **{** 0**,** 1**,** 2 **},** **{** 3**,** 4**,** 5 **}** **};**

int**[,]** nums **=** **new** **[,]{** **{** 0**,** 1**,** 2 **},** **{** 3**,** 4**,** 5 **}** **};**

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

**foreach** **(**int i **in** numbers**)**

Console**.**Write**(**$"{i} "**);** // 012345

**for** **(**val i **=** 0**;** i **<=** nums**.**GetUpperBound**(**0**);** i**++)**

**for** **(**val j **=** 0**;** j **<=** nums**.**GetUpperBound**(**1**);** j**++)**

Console**.**Write**(**nums**[**i**,**j**]);** // 012345

### Массив массивов

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

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

// выделяем память для первого подмассива

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

// выделяем память для второго подмассива

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

// выделяем память для третьего подмассива

int**[][]** nums **=** **{**

**new** int**[]** **{** 1**,** 2 **},**

**new** int**[]** **{** 1**,** 2**,** 3 **},**

**new** int**[]** **{** 1**,** 2**,** 3**,** 4**,** 5 **}**

**};**

**foreach** **(**int**[]** row **in** nums**)**

**foreach** **(**int num **in** row**)**

Console**.**Write**(**num**);** // 12345

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

**for** **(**val j **=** 0**;** j **<** nums**[**i**].**Length**;** j**++)**

Console**.**Write**(**nums**[**i**][**j**]);** \\ 12345

## Методы

int Sum**(**int x**,** int y **=** 0**)**

**{**

**return** x **+** y**;**

**}**

int Sum**(**int x**,** int y **=** 0**)** **=>** x **+** y**;**

Sum**(**4**)** **-** 4

Sum**(**4**,** 5**)** **-** 9

Sum**(**y**:**10**,** x**:**5**)** **-** 15

## Передача параметров по ссылке. Выходные параметры

### Модификатор ref

**При передаче значений параметрам по ссылке метод получает адрес переменной в памяти.**

**Параметр с модификатором ref обязан быть инициализирован перед передачей в метод.**

void Increment(ref int n)

{

n++;

}

int number = 5;

Increment(ref number) // number = 6

### Модификатор out

**Методы, использующие параметры out, обязательно должны присваивать им определенное значение, даже если оно им уже присвоено где-то в коде перед вызовом метода (в отличие от ref).**

void Sum(int x, int y, out int sum, out int mul)

{

sum = x + y;

mul = x \* y;

}

int numSum;

**Sum(10, 5, out numSum, out int numMul); // numSum = 15, numMul = 50 //(переменную можно определить непосредственно при вызове метода)**

### Модификатор in

**Модификатор in указывает, что данный параметр будет передаваться в метод по ссылке, однако внутри метода его значение параметра нельзя будет изменить.**

int Sum(in int x, in int y = 0) => x + y;

### Модификатор ref readonly (c C#12)

**ref-параметры только для чтения.**

void Increment(ref readonly int n)

{

// n++; // нельзя, иначе будет ошибка компиляции

}

**int number = 5;**

**Increment(ref number);**

## Массив параметров и ключевое слово params

int Sum**(params** int**[]** numbers**)**

**{**

int result **=** 0**;**

**foreach** **(**var n **in** numbers**)**

result **+=** n**;**

**return** result**;**

**}**

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

Sum**(**nums**);**

Sum**(**1**,** 2**,** 3**,** 4**);**

Sum**();**

## Локальные функции

**Локальные функции представляют функции, определенные внутри других методов.**

**Локальная функция, как правило, содержит действия, которые применяются только в рамках ее метода.**

bool AreEqual**(**int**[]** nums1**,** int**[]** nums2**)**

**{**

int sum1 **=** Sum**(**nums1**);**

int sum2 **=** Sum**(**nums2**);**

**return** sum1 **==** sum2**;**

int Sum**(**int**[]** nums**)** **=>** nums**.**Sum**();**

// static int Sum(int[] nums) => nums.Sum();

// static функции не могут обращаться к переменным окружения, то есть //метода,

// в котором статическая функция определена в отличие от нестатических //функций.

**}**

## Конструкция switch

**switch** **(**number**)**

**{**

**case** 1**:**

Console**.**WriteLine**(**"case 1"**);**

**goto** **case** 5**;** // переход к case 5

**case** 3**:**

Console**.**WriteLine**(**"case 3"**);**

**break;**

**case** 5**:**

Console**.**WriteLine**(**"case 5"**);**

**break;**

**default:**

Console**.**WriteLine**(**"default"**);**

**break;**

**}**

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

**{**

**switch** **(**op**)**

**{**

**case** 1**:** **return** a **+** b**;**

**case** 2**:** **return** a **-** b**;**

**case** 3**:** **return** a **\*** b**;**

**default:** **return** 0**;**

**}**

**}**

int result **=** op **switch** **{**

1 **=>** a **+** b**,**

2 **=>** a **-** b**,**

3 **=>** a **\*** b**,**

\_ **=>** 0

**};**

## Перечисления enum

enum DayTime

**{**

Morning**,**

Afternoon**,**

Evening**,**

Night

**}**

DayTime**.**Morning **- обращение к значению Morning**

enum Time **:** byte

**{**

Morning**,** // 0

Afternoon**,** // 1

Evening**,** // 2

Night // 3

**}**

**(**int**)** DayTime**.**Night **-** 3

enum DayTime

**{**

Morning **=** 3**,** // каждый следующий элемент по умолчанию увеличивается на //единицу

Afternoon**,** // 4

Evening**,** // 5

Night **=** Morning // 3

**}**

Классы

## Создание конструкторов

class Person

**{**

**public** string name**;**

**public** Person**()** **{** name **=** "Неизвестно"**}** // 1 конструктор

**public** Person**(**string name**)** **{** **this.**name **=** name **}** // 2 конструктор

**}**

Person tom **=** **new** **();** // аналогично new Person();

Person bob **=** **new** **(**"Bob"**);** // аналогично new Person("Bob");

### Цепочка вызовов конструкторов

class Person

**{**

**public** string name**;**

**public** Person**()** **:** **this(**"Неизвестно"**)** **{** **}** // первый конструктор

**public** Person**(**string name**)** **{** **this.**name **=** name **}** // второй конструктор

**}**

### Первичные конструкторы (начиная с версии C#12)

**public** class Person**(**string name**,** int age**)**

**{**

**public** Person**(**string name**)** **:** **this(**name**,** 18**)** **{** **}**

**}**

Person tom **=** **new** **(**"Tom"**,** 34**);**

Person bob **=** **new** **(**"Bob"**);**

### Инициализаторы объектов

class Person

**{**

**public** string name**;**

**public** Company company**;**

**public** Person**()**

**{**

name **=** "Undefined"**;**

company **=** **new** Company**();**

**}**

**}**

class Company

**{**

**public** string title **=** "Unknown"**;**

**}**

Person tom **=** **new** Person**{** name **=** "Tom"**,** company **=** **{** title **=** "Microsoft"**}** **};**

### Деконструкторы

class Person

**{**

string name**;**

int age**;**

**public** Person**(**string name**,** int age**)**

**{**

**this.**name **=** name**;**

**this.**age **=** age**;**

**}**

**public** void Deconstruct**(out** string personName**,** **out** int personAge**)**

**{**

personName **=** name**;**

personAge **=** age**;**

**}**

**}**

Person person **=** **new** Person**(**"Tom"**,** 33**);**

**(**string name**,** int age**)** **=** person**;**

// То же самое, что и:

// string name; int age;

// person.Deconstruct(out name, out age);

// Если нужна только одна переменная, можно так:

// (\_, int age) = person;

Console**.**WriteLine**(**name**);** // Tom

Console**.**WriteLine**(**age**);** // 33

## Структуры

### Инициализация с помощью конструктора

struct Person

**{**

// элементы структуры:

// поля и методы

int age **=** 10**;** // инициализация полей значениями по умолчанию - доступна с //C#10

**public** Person**()** // начиная с C#10 мы можем определить свой конструктор //без параметров

**{**

age **=** 20**;**

**}**

**}**

Person tom **=** **new** Person**();** // вызов конструктора

Person bob **=** **new();**

### Цепочка конструкторов

struct Person

**{**

**public** int age**;**

**public** Person**()** **:** **this(**10**)**

**{** **}**

**public** Person**(**string name**)** **:** **this(**name**,** 1**)**

**{** **}**

**public** Person**(**int age**)**

**{**

**this.**age **=** age**;**

**}**

**}**

var tom **=** **new** Person**(**20**);**

### Первичный конструктор

**public** struct Person**(**string name**,** int age**)**

**{**

**public** Person**(**string name**)** **:** **this(**name**,** 18**)** **{** **}**

**}**

### Инициализатор структуры

struct Person

**{**

**public** string name**;**

**public** int age**;**

**}**

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

### Копирование структуры с помощью with

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

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

// объект bob получает все значения объекта tom,

// а затем после оператора with в фигурных скобках указывается поля со //значениями,

// которые мы хотим изменить.

## Модификаторы доступа

**private: закрытый или приватный компонент класса или структуры. Приватный компонент доступен только в рамках своего класса или структуры.**

**private protected: компонент класса доступен из любого места в своем классе или в производных классах, которые определены в той же сборке.**

**file: добавлен в версии C# 11 и применяется к типам, например, классам и структурам. Класс или структура с такми модификатором доступны только из текущего файла кода.**

**protected: такой компонент класса доступен из любого места в своем классе или в производных классах. При этом производные классы могут располагаться в других сборках.**

**internal: компоненты класса или структуры доступен из любого места кода в той же сборке, однако он недоступен для других программ и сборок.**

**protected internal: совмещает функционал двух модификаторов protected и internal. Такой компонент класса доступен из любого места в текущей сборке и из производных классов, которые могут располагаться в других сборках.**

**public: публичный, общедоступный компонент класса или структуры. Такой компонент доступен из любого места в коде, а также из других программ и сборок.**

## Свойства

### Определение свойств

**private** string name **=** "Undefined"**;**

**public** string Name

**{**

**get**

**{**

**return** name**;** // возвращаем значение свойства

**}**

**set**

**{**

name **=** **value;** // устанавливаем новое значение свойства

**}**

**}**

person**.**Name **=** "Tom"**;**

### Свойства только для чтения и записи

// свойство только для записи

**public** int Age

**{**

**set** **{** age **=** **value;** **}**

**}**

// свойство только для чтения

**public** string Name

**{**

**get** **{** **return** name**;** **}**

**}**

### Модификаторы доступа

**public** string Name

**{**

**get** **{** **return** name**;** **}**

**private** **set** **{** name **=** **value;**

**}**

***Ограничения:***

**- Модификатор для блока set или get можно установить, если свойство имеет оба блока (и set, и get);**

**- Только один блок set или get может иметь модификатор доступа, но не оба сразу;**

**- Модификатор доступа блока set или get должен быть более ограничивающим, чем модификатор доступа свойства.**

### Автоматические свойства

string Name **{** **get;** **set;** **}**

int Age **{** **get;** **set;** **}** **=** 37**;** // значение по умолчанию

string City **{** **private** **set;** **get;}** // модификатор доступа

### Блок init (начиная с версии C# 9.0)

**После инициализации значений свойств их значения изменить нельзя - они доступны только для чтения.**

***1 способ:***

string Name **{** **get;** init**;** **}** **=** "Undefined"**;** // изменить нельзя

***2 способ:***

Person**(**string name**)** **=>** Name **=** name**;** // инициализация в конструкторе

string Name **{** **get;** init**;** **}**

***3 способ:***

class Person

**{**

string Name **{** **get;** init**;** **}** **=** ""**;**

**}**

Person person **=** **new()** **{** Name **=** "Bob"**};**

***4 способ:***

string name **=** ""**;**

string Name // свойство Name управляет полем для чтения name

**{**

**get** **{** **return** name**;** **}**

init

**{**

name **=** **value;**

**}**

**}**

### Сокращенная запись свойств

string name**;**

string Name

**{**

**get** **=>** name**;**

**set** **=>** name **=** **value;**

**}**

string name**;**

**public** string Name **=>** name**;**

// эквивалентно public string Name { get { return name; } }

### Модификатор required (добавлен в C# 11)

**Модификатор required указывает, что поле или свойство с этим модификатором обязательно должно быть инициализировано.**

Person**(**string name**)**

**{**

Name **=** name**;**

**}**

required string Name **{** **get;** **set;** **}** // инициализировано в конструкторе

required int Age **{** **get;** **set;** **}** **=** 22**;**

## Статические члены и модификатор static

### Статические поля

**Статические поля хранят состояние всего класса / структуры.**

### Статические свойства

static int age **=** 65**;**

static int Age

**{**

**get** **{** **return** age**;** **}**

**set** **{** age **=** **value;** **}**

**}**

### Статические методы

**Статические методы могут обращаться только к статическим членам класса.**

### Статический конструктор

**- Статические конструкторы не должны иметь модификатор доступа и не принимают параметров;**

**- Как и в статических методах, в статических конструкторах нельзя использовать ключевое слово this для ссылки на текущий объект класса и можно обращаться только к статическим членам класса;**

**- Статические конструкторы нельзя вызвать в программе вручную. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам (если таковые имеются).**

### Статические классы

**Статические классы объявляются с модификатором static и могут содержать только статические поля, свойства и методы.**

## Константы, поля и структуры для чтения

### Константы класса

**Значение константы устанавливается один раз непосредственно при её объявлении и впоследствии не может быть изменено.**

**Константы хранят некоторые данные, которые относятся не к одному объекту, а ко всему классу в целом.**

**Для обращения к константам применяется не имя объекта, а имя класса:**

class Person

**{**

const string type **=** "Person"**;**

**}**

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

### Поля для чтения и модификатор readonly

**Таким полям можно присвоить значение либо при непосредственно при их объявлении, либо в конструкторе.**

**Их значение нельзя изменить.**

**readonly** string name **=** "Tom"**;**

### Сравнение констант

**- const должны быть определены во время компиляции, а readonly могут быть определены во время выполнения программы в конструкторе.**

**- const в отличие от readonly не могут использовать модификатор static, так как уже неявно являются статическими.**

### Структуры для чтения

**Все их поля должны быть также полями для чтения:**

**readonly** struct Person

**{**

**readonly** int age **=** 5**;**

**readonly** string Name **{** **get;** **}** **=** "Tom"**;** // указывать readonly //необязательно

**}**

## Null и ссылочные типы

#nullable disable

string text **=** **null;** // здесь nullable-контекст не действует

#nullable restore

string**?** name **=** **null;** // здесь nullable-контекст снова действует

### Оператор ! (null-forgiving operator)

string**?** name **=** "Tom"

Console**.**WriteLine**(**name**!);**

## Null и значимые типы

int**?** val **=** **null;** // nullable int тип

// то же самое, что и:

Nullable**<**int**>** val **=** **null;**

**Структура Nullable<T> имеет два свойства:**

**- Value - значение объекта;**

**- HasValue: возвращает true, если объект хранит некоторое значение, и false, если объект равен null.**

**... и метод:**

**- GetValueOrDefault(): возвращает значение переменной/параметра, если они не равны null. Если они равны null, то возвращается значение по умолчанию. Значение по умолчанию можно передать в метод.**

int**?** number **=** **null;** // если значения нет, метод возвращает значение по //умолчанию

Console**.**WriteLine**(**number**.**GetValueOrDefault**());** // 0 - значение по //умолчанию для числовых типов

Console**.**WriteLine**(**number**.**GetValueOrDefault**(**10**));** // 10

**В арифметических операциях, если один из операндов равен null, то результатом операции также будет null:**

int**?** x **=** **null;**

int**?** w **=** x **+** 7**;** // w = null

**В операциях сравнения >, <, >= и <=, если хотя бы один из операндов равен null, то возвращается false (кроме операции !=).**

## Проверка на null, операторы ?. и ??

**С помощью оператора is мы можем проверить значение объекта:**

string name **=** **null;**

**if** **(**name **is** **null)** **...** // объект is значение

// Также можно проверить на соответствие типу:

// name is string

**С помощью is not можно проверить отсутствие значения:**

**if** **(**name **is** not **null)** **...**

### Оператор ??

**Оператор ?? возвращает левый операнд, если этот операнд не равен null.**

**Иначе возвращается правый операнд.**

**При этом левый операнд должен принимать null.**

int**?** id **=** 200**;**

int personid **=** id **??** 0**;** // равно 200, так как id не равен null

### Оператор условного null ( ?. )

Person**?.**name

## Псевдонимы типов и статический импорт

### Псевдонимы

**using** printer **=** System**.**Console**;**

printer**.**WriteLine**(**"Hello, World"**);**

**using** User **=** Person**;**

User tom **=** **new** User**(**"Tom"**);**

### Статический импорт

**Выражение using static подключает в программу все статические методы и свойства, а также константы.**

**И после этого мы можем не указывать название класса при вызове метода.**

**using** static System**.**Console**;**

WriteLine**(**"Hello from C# 8.0"**);**

ООП

## Наследование

class Person **{** string Name **{** **get;** **set;** **}** **}**

class Employee **:** Person **{** **...** **}**

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

Person employee **=** **new** Employee **{** Name **=** "Sam" **};**

***Ограничения:***

**- Класс может наследоваться только от одного класса.**

**- Тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. Если базовый и производный класс находятся в разных сборках (проектах), то производый класс может наследовать только от класса public.**

**- sealed класс не допускает создание наследников: sealed class Admin {}**

**- Нельзя унаследовать класс от статического класса.**

### Ключевое слово base

**Если в базовом классе не определен конструктор по умолчанию без параметров, а только конструкторы с параметрами, то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово base.**

**При вызове конструктора класса сначала отрабатываются конструкторы базовых классов и только затем конструкторы производных.**

class Person

**{**

**public** string Name **{** **get;** **set;}**

**public** Person**(**string name**)**

**{**

Name **=** name**;**

**}**

**}**

class Employee **:** Person

**{**

**public** string Company **{** **get;** **set;** **}**

**public** Employee**(**string name**,** string company**)** **:** **base(**name**)**

**{**

Company **=** company**;**

**}**

**}**

## Преобразование типов

### Восоходящее преобразование. Upcasting

Employee employee **=** **new** Employee**(**"Tom"**,** "Microsoft"**);**

Person person **=** employee**;** // преобразование от Employee к Person

**employee и person будут указывать на один и тот же объект в памяти, но переменной person будет доступна только та часть, которая представляет функционал типа Person.**

### Нисходящее преобразование. Downcasting

Person person **=** **new** Person**();**

Employee employee2 **=** **(**Employee**)**person**;** // преобразование от Person к //Employee

### Способы преобразований

Person person **=** **new** Person**();**

Employee**?** employee **=** person **as** Employee**;** // В случае неудачного //преобразования выражение будет содержать значение null

**if** **(**person **is** Employee employee**)**

**{**

Console**.**WriteLine**(**employee**.**Company**);**

**}**

**Выражение if (person is Employee employee) проверяет, является ли переменная person объектом типа Employee. Если person является объектом Employee, то автоматически преобразует значение переменной person в тип Employee и преобразованное значение сохраняет в переменную employee.**

**if** **(**person **is** Employee**)** // без преобразования типа Person в Employee

**{**

Console**.**WriteLine**(**"Представляет тип Employee"**);**

**}**

## Виртуальные методы и свойства

**Методы в родительском классе, помеченные модификатором virtual, доступны для переопределения.**

**В классе-наследнике мы можем переопределить его, пометив модификатором** **override.**

class Person

**{**

**virtual** void Method**(){}**

**}**

class Employee **:** Person

**{**

**virtual** void Method**(){}**

**}**

**Ограничения:**

**- Виртуальный и переопределенный методы должны иметь один и тот же модификатор доступа.**

**- Нельзя переопределить или объявить виртуальным статический метод.**

### Ключевое слово base

class Employee **:** Person

**{**

Employee**(** **...** **)** **:base(** **...** **)** // из базового класса можно вызвать и конструктор

**{**

**...**

**}**

**override** void Method**()**

**{**

**base.**Print**();** // и метод с помощью ключевого слова base

**...**

**}**

**}**

### Переопределение свойств

**Можно переопределять свойства:**

class Person

**{**

**private** int age**;**

**public** **virtual** int Age **{** **get;** **set;** **}**

**}**

class Employee **:** Person

**{**

**public** **override** Age

**{**

**get** **=>** **base.**Age**;**

**set** **{** **if** **(value** **>** 18**)** **base.**Age **=** **value;** **}**

**}**

**}**

### Запрет переопределения методов

**override sealed запрещает переопределение методов и свойств. Оно работает только в уже переопределенных методах в классах-наследниках, т.к. в родительском классе достаточно не ставить модификатор virtual:**

class Employee **:** Person

**{**

**override** **sealed** void Method**()** **{** **...** **}**

**}**

## Скрытие методов и свойств

class Person

**{**

**public** int age **=** 1**;**

**public** string Name **{** **get;** **set;** **}**

**public** void Print**()** **{}** // не виртуальный метод - нельзя переопределить

**}**

class Employee **:** Person

**{**

**public** **new** int age **=** 18**;** // скрываем родительскую переменную или //константу своей

**public** **new** string Name **{** **get** **=>** **...** **;** **set** **=>** **base.**Name **=** **value;** **}** // //скрываем родительское свойство своим

**public** **new** void Print**()** **{}** // скрываем родительский метод своим

**}**

## Различие переопределения и скрытия методов

### Переопределение

class Person

**{**

**virtual** void Print**()** **{** Console**.**WriteLine**(**"Person"**);** **}**

**}**

class Employee **:** Person

**{**

**override** void Print**()** **{** Console**.**WriteLine**(**"Employee"**);** **}**

**}**

Person person **=** **new** Employee**();**

tom**.**Print**();** // Employee

**Через Таблицу Виртуальных Методов вызывается метод класса Employee, хотя объект типа Person.**

### Скрытие

class Person

**{**

void Print**()** **{** Console**.**WriteLine**(**"Person"**);** **}**

**}**

class Employee **:** Person

**{**

**new** void Print**()** **{** Console**.**WriteLine**(**"Employee"**);** **}**

**}**

Person person **=** **new** Employee**();**

tom**.**Print**();** // Person

**Вызывается метод класса Person объекта типа Person.**

## Абстрактные классы и члены классов

**abstract** class Transport

**{**

int age **=** 0**;** // может иметь переменные

int Age **{** **get** **=>** age**;** **set** **=>** age **=** **value;** **}** // может иметь свойства

void Move**()** **{** **...** реализация **}** // может иметь методы с реализацией

**abstract** string Name **{** **get;** **set;** **}** // абстрактные свойства

**abstract** int Speed **{** **get;** **set;** **}**

**abstract** void Stop**();** // абстрактный метод

**}**

class Car **:** Transport

**{**

string name**;**

**override** string Name **{** **get** **=>** name**;** **set** **=>** name **=** **value;** **}** // реализации //свойств

**override** int Speed **{** **get;** **set;** **}**

**override** void Stop**()** **{** **...** реализация **}** // реализация метода

**}**

## Класс System.Object и его методы

**У класса Object есть основные методы, которые можно переопределить с помощью ключевого слова override:**

**override** string**?** ToString**()**

**override** int GetHashCode**()**

**override** bool Equals**(object?** obj**)**

**... и метод который не переопределяется:**

Type GetType**()** // проверить тип можно с помощью конструкций:

// if (person.GetType() == typeof(Person)

// if (person is Person)

## Обобщения (generics)

class Person**<**T**>** **{...}**

class Person**<**T**,** K**>** **{...}**

### Статические поля обобщенных классов

class Person**<**T**>**

**{**

static T**?** Id**;**

**}**

Person**<**int**>** tom **=** **new** Person**<**int**>();**

tom**.**Id **=** 1**;**

Person**<**string**>** bob **=** **new** Person**<**string**>();**

bob**.**Id **=** "Bob"**;**

**Для класса Person<int> и для класса Person<string> в куче будет создана своя переменная Id.**

### Обобщенные методы

void Method**<**T**>(**T x**)** **{...}**

## Ограничения обобщений

### Ограничения методов

void SendMessage**<**T**>(**T message**)** **where** T**:** Message **{...}**

// через универсальный параметр T будут передаваться объекты класса Message и //производных классов

### Ограничения обобщений в типах

class Messenger**<**T**>** **where** T **:** Message **{...}**

### Типы ограничений и стандартные ограничения

**В качестве ограничений мы можем использовать следующие типы:**

**- Классы (where T : Message)**

**- Интерфейсы (where T : IMessage)**

**- class - универсальный параметр должен представлять класс (where T : class)**

**- struct - универсальный параметр должен представлять структуру (where T : struct)**

**- new() - универсальный параметр должен представлять тип, который имеет общедоступный (public) конструктор без параметров (where T : new())**

**Порядок следования нескольких ограничений:**

**1. Название класса, class, struct. Причем мы можем одновременно определить только одно из этих ограничений**

**2. Название интерфейса**

3. **new()**

class Messenger**<**T**>** **where** T**:** Message**,** **new()**

### Использование нескольких универсальных параметров

class Messenger**<**T**,** P**>**

**where** T **:** Message

**where** P**:** Person

**{...}**

## Наследование обобщенных типов

**Пример 1:**

class Person**<**T**>** **{...}**

class Employee**<**T**>** **:** Person**<**T**>** **{...}**

**Пример 2:**

class Person**<**T**>** **{...}**

class Employee **:** Person**<**string**>** **{...}**

**Пример 3:**

class Person**<**T**>** **{...}**

class Employee**<**T**>** **:** Person**<**int**>** **{...}**

**Пример 4:**

class Person**<**T**>** **{...}**

class Employee**<**T**,** K**>** **:** Person**<**T**>** **where** K **:** struct **{...}**

**Пример 5:**

class Person**<**T**>** **where** T **:** class **{...}**

class Employee**<**T**>** **:** Person**<**T**>** **where** T **:** class **{...}**

**Если в базовом классе в качестве ограничение указано class, то есть любой класс, то в производном классе также надо указать в качестве ограничения class, либо же какой-то конкретный класс.**

Исключения

## Конструкция try..catch..finally

**try** **{}**

**catch** **{}** // просто catch или catch(Exception), или catch(Exception e)

**finally** **{}** // опционально

### Фильтры исключений

int x **=** 1**,** y **=** 0**;**

**try**

**{**

int result **=** x **/** y**;**

**}**

**catch** **when** **(**y **==** 0**)** **{...}**

// catch(Exception) when (),

// catch(Exception e) when()

## Типы исключений. Класс Exception

***Свойства класса Exception:***

**- InnerException: хранит информацию об исключении, которое послужило причиной текущего исключения**

**- Message: хранит сообщение об исключении, текст ошибки**

**- Source: хранит имя объекта или сборки, которое вызвало исключение**

**- StackTrace: возвращает строковое представление стека вызывов, которые привели к возникновению исключения**

**- TargetSite: возвращает метод, в котором и было вызвано исключение**

***Некоторые типы исключений:***

**- DivideByZeroException: представляет исключение, которое генерируется при делении на ноль**

**- ArgumentOutOfRangeException: генерируется, если значение аргумента находится вне диапазона допустимых значений**

**- ArgumentException: генерируется, если в метод для параметра передается некорректное значение**

**- IndexOutOfRangeException: генерируется, если индекс элемента массива или коллекции находится вне диапазона допустимых значений**

**- InvalidCastException: генерируется при попытке произвести недопустимые преобразования типов**

**- NullReferenceException: генерируется при попытке обращения к объекту, который равен null (то есть по сути неопределен)**

## Генерация исключения и оператор throw

**throw** **new** Exception**(**"..."**);**

**catch**

**{**

**...**

**throw;**

**}** // в блоке catch исключение обрабатывается, а затем передается внешнему блоку кода

Делегаты, лямбды и события

## Делегаты

Operation operation **=** Add**;** // делегат указывает на метод Add

int result **=** operation**(**4**,** 5**);** // фактически Add(4, 5)

// можно создать объект делегата с помощью конструктора

operation **=** **new** Operation**(**Multiply**);**

result **=** operation**(**4**,** 5**);** // фактически Multiply(4, 5)

int Add**(**int x**,** int y**)** **=>** x **+** y**;**

int Multiply**(**int x**,** int y**)** **=>** x **\*** y**;**

**delegate** int Operation**(**int x**,** int y**);**

**Делегат можно определять внутри или вне класса.**

**Методы должны соответствовать делегату по возвращаемому типу и наборы параметров (учитывая ref, out, in)**

### Добавление методов в делегат

**В делегат можно добавлять методы (с помощью +=), которые попадают в список - invokation list, и будут вызываться последовательно, но, ЕСЛИ ДЕЛЕГАТ ВОЗВРАЩАЕТ ЗНАЧЕНИЕ, то ВОЗВРАЩАЕТСЯ ЗНАЧЕНИЕ ТОЛЬКО ПОСЛЕДНЕГО МЕТОДА ИЗ СПИСКА. Можно несколько раз добавлять один и тот же метод. Можно удалять метод (если список содержит несколько одинаковых методов, то с конца списка) с помощью -=.**

Message message **=** Hello**;**

message **+=** HowAreYou**;** // теперь message указывает на два метода

message **+=** Hello**;**

message **+=** HowAreYou**;**

message **-=** Hello**;** // удалили последний Hello()

message**();** // методы вызываются по очереди

void Hello**()** **=>** Console**.**WriteLine**(**"Hello/n"**);**

void HowAreYou**()** **=>** Console**.**WriteLine**(**"How are you?/n"**);**

**delegate** void Message**();**

/\*

Hello

How are you?

How are you?

\*/

### Объединение делегатов

Message mes1 **=** Hello**;**

Message mes2 **=** HowAreYou**;**

Message mes3 **=** mes1 **+** mes2**;** // объединяем делегаты

mes3**();** // вызываются все методы из mes1 и mes2

void Hello**()** **=>** Console**.**WriteLine**(**"Hello/n"**);**

void HowAreYou**()** **=>** Console**.**WriteLine**(**"How are you?/n"**);**

**delegate** void Message**();**

/\*

Hello

How are you?

\*/

### Вызов делегата с помощью метода Invoke()

Operation**?** op **=** Add**;**

// другой способ вызова делегата - с помощью Invoke()

int result **=** op**?.**Invoke**(**4**,** 5**)** **?:** 0**;** // проверяем на null (вдруг список делегата пуст)

int Add**(**int x**,** int y**)** **=>** x **+** y**;**

**delegate** int Operation**(**int x**,** int y**);**

### Обобщенные делегаты (Generic)

Operation**<**double**,** int**>** op **=** Square**;**

int result **=** op**(**4**);** // 16.0

double Square**(**int n**)** **=>** **(**double**)** n **\*** n**;**

**delegate** T Operation**<**T**,** K**>(**K val**);**

### Делегаты как параметры методов

DoOperation**(**5**,** 4**,** Add**);** // 9

DoOperation**(**5**,** 4**,** Multiply**);** // 20

void DoOperation**(**int a**,** int b**,** Operation op**)**

**{**

Console**.**WriteLine**(**op**(**a**,**b**));**

**}**

int Add**(**int x**,** int y**)** **=>** x **+** y**;**

int Multiply**(**int x**,** int y**)** **=>** x **\*** y**;**

**delegate** int Operation**(**int x**,** int y**);**

**При вызове метода DoOperation мы можем передать в него в качестве третьего параметра метод, который соответствует делегату Operation.**

### Возвращение делегатов из метода

Letter op **=** SelectOp**(true);**

string result **=** op**();** // A

Letter SelectOp**(**bool a**)** **=>** a **==** "A" **?** A**()** **:** B**();**

string A**()** **=>** "A"

string B**()** **=>** "B"

**delegate** string Letter**();**

## Анонимные методы

***Анонимные методы используются для создания экземпляров делегатов.***

Operation operation **=** **delegate** **(**int x**,** int y**)**

**{**

**return** x **+** y**;**

**};**

int result **=** operation**(**4**,** 5**);** // 9

**delegate** int Operation**(**int x**,** int y**);**

***Если параметров нет, то скобки опускаются***

Message message **=** **delegate**

**{**

Console**.**WriteLine**(**"Message"**);**

**};**

message**();** // Message

**delegate** void Operation**();**

***Даже если делегат содержит параметры, то в анонимном методе они могут не использоваться, и скобки опускаются***

Operation operation **=** **delegate**

**{**

**return** 0**;**

**};**

int result **=** operation**(**4**,** 5**);** // 9

**delegate** int Operation**(**int x**,** int y**);**

***Можно передавать анонимный метод в качестве параметра метода типа делегата***

ShowMessage**(**"Hello!"**,** **delegate** **(**string mes**)**

**{**

Console**.**WriteLine**(**mes**);**

**});**

static void ShowMessage**(**string message**,** MessageHandler handler**)**

**{**

handler**(**message**);**

**}**

**delegate** void MessageHandler**(**string message**);**

## Лямбды

***(список\_параметров) => выражение***

**Лямбда-выражения представляют упрощенную запись анонимных методов. С точки зрения типа данных лямбда-выражение представляет делегат.**

Message hello **=** **()** **=>** Console**.**WriteLine**(**"Hello"**);**

hello**();** // Hello

**delegate** void Message**();**

**Если лямбда-выражение содержит несколько действий, то они помещаются в фигурные скобки:**

Message hello **=** **()** **=>**

**{**

Console**.**Write**(**"Hello "**);**

Console**.**WriteLine**(**"World"**);**

**};**

hello**();** // Hello World

**delegate** void Message**();**

**Начиная с версии C# 10:**

var hello **=** **()** **=>** Console**.**WriteLine**(**"Hello"**);**

// переменной hello присваивается тип встроенного делегата Action,

// который не принимает параметров и ничего не возвращает

hello**();** // Hello

### Параметры лямбды

**При определении списка параметров мы можем не указывать для них тип данных:**

Operation sum **=** **(**x**,** y**)** **=>** Console**.**WriteLine**(**$"{x + y}"**);**

sum**(**1**,** 2**);** // 3

**delegate** void Operation**(**int x**,** int y**);**

**Но при использовании неявной типизации (var) их нужно указать:**

var sum **=** **(**int x**,** int y**)** **=>** Console**.**WriteLine**(**$"{x + y}"**);**

**Если лямбда имеет один параметр, для которого не требуется указывать тип данных, то скобки можно опустить:**

Print print **=** message **=>** Console**.**WriteLine**(**message**);**

print**(**"Hello"**);** // Hello

**delegate** void Print**(**string message**);**

**Начиная с C# 12 параметры лямбда-выражений могут иметь значения по умолчанию:**

var welcome **=** **(**string message **=** "Hello"**)** **=>** Console**.**WriteLine**(**message**);**

welcome**();** // Hello

welcome**(**"Bye"**);** // Bye

### Возвращение результата

var sum **=** **(**int x**,** int y**)** **=>** x **+** y**;**

int result **=** sum**(**3**,** 2**);** // 5

// Если лямбда-выражение содержит несколько выражений (или одно выражение в фигурных

// скобках), тогда нужно использовать оператор return, как в обычных методах:

Operation multiply **=** **(**x**,** y**)** **=>**

**{**

**...**

**return** x **\*** y**;**

**}**

result **=** **(**3**,** 2**);** // 6

**delegate** int Operation**(**int x**,** int y**);**

### Добавление и удаление действий в лямбда-выражении, метод Invoke

var hello **=** **()** **=>** Console**.**WriteLine**(**"Hello "**);**

var world **=** **()** **=>** Console**.**WriteLine**(**"World"**);**

hello **+=** world**;**

hello **+=** **()** **=>** Console**.**WriteLine**(**"!"**);**

hello**?.**Invoke**();** // Hello World!

hello **-=** world**;**

hello**();** // Hello !

### Лямбда-выражение как аргумент метода

ShowMessage**(**"Hello!"**,** mes **=>** Console**.**WriteLine**(**mes**));**

static void ShowMessage**(**string message**,** MessageHandler handler**)**

**{**

handler**(**message**);**

**}**

**delegate** void MessageHandler**(**string message**);**

### Лямбда-выражение как результ метода

Letter op **=** SelectOp**(true);**

string result **=** op**();** // A

Letter SelectOp**(**bool a**)** **=>**

**{**

**return** a **==** "A" **?** **()** **=>** "A" **:** **()** **=>** "B"**;**

**}**

**delegate** string Letter**();**

## События

### Определение и вызов событий

**В одном классе определяем событие, и в нужных местах (в нужных методах) вызываем это событие (с разными параметрами, если нужно)**

class MyClass

**{**

**delegate** void MyDelegate**(**string message**);** // 1. Определение делегата

**event** MyDelegate MyEvent**;** // 2.Определение события

**...**

MyEvent**(**"Произошло хорошее событие"**);** // 3.Вызов события

**...**

MyEvent**?.**Invoke**(**"Произошло плохое событие"**);** // Вызов события в другом //месте

**}**

### Добавление обработчика события

**В другом классе создаем обработчик события - метод с той же сигнатурой, что и делегат события, который реализует поведение-реакцию на событие (в зависимости от пришедших параметров, если они есть). Добавляем обработчик события с помощью +=**

MyClass**.**MyEvent **+=** EventHandler**;** // 5. Добавление обработчика события

MyClass**.**MyEvent **+=** EventHandlerRed**;** // Добавление другого обработчика события

MyClass**.**MyEvent **-=** EventHandler**;** // Удаление обработчика события

void EventHandler**(**string message**)** **=>** Console**.**WriteLine**(**message**);**

// 4. Реализация обработчика события

void EventHandlerRed**(**string message**)**

// Реализация другого обработчика события

**{**

// Устанавливаем красный цвет символов

Console**.**ForegroundColor **=** ConsoleColor**.**Red**;**

Console**.**WriteLine**(**message**);**

// Сбрасываем настройки цвета

Console**.**ResetColor**();**

**}**

***Установка в качестве обработчика делегата:***

MyClass**.**MyEvent **+=** **new** MyClass**.**MyDelegate**(**EventHandler**);**

// установка делегата через конструктор

***Установка в качестве обработчика анонимного метода:***

MyClass**.**MyEvent **+=** **delegate** **(**string message**)** **=>** Console**.**WriteLine**(**message**);**

***Установка в качестве обработчика лямбда-выражения:***

MyClass**.**MyEvent **+=** message **=>** Console**.**WriteLine**(**message**);**

### Управление обработчиками (add, remove)

class MyClass

**{**

**delegate** void MyDelegate**(**string message**);** // делегат

MyDelegate**?** myEvent**;** // переменная делегата

**event** MyDelegate MyEvent // переменная события

**{**

**add** // будет вызываться при добавлении обработчика

**{**

myEvent **+=** **value;**

// value - ключевое слово - добавляемый обработчик

**}**

**remove** // будет вызываться при удалении обработчика

**{**

myEvent **-=** **value;**

**}**

**}**

**...**

**}**

MyClass**.**MyEvent **+=** EventHandler**;**

void EventHandler**(**string message**)** **=>** Console**.**WriteLine**(**message**);**

## Ковариантность и контравариантность делегатов

class Message **{**

string text**;**

Message**(**string text**)** **=>** **this.**text **=** text**;**

**}**

class Email **:** Message **{** Email**(**string text**)** **:** **base(**text**)** **{}** **}**

**Дочерний класс - это частный случай родительского класса, и он может иметь больший функционал, чем родительский класс.**

**Ковариантность - это возвращение дочерних классов вместо родительских.**

**(могу вернуть Email вместо Message, т.к. Email - это и есть Message, но не наоборот, и то, что есть в Message, есть и в Email).**

**Контрвариантность - это использование родительских классов вместо дочерних.**

**(в качестве реализации делегата могу использовать метод, принимающий Message вместо Email, т.к. то, что есть в Message, есть и в Email).**

### Ковариантность

**Ковариантность позволяет передать делегату метод, возвращаемый тип которого является дочерним от возвращаемого типа делегата.**

**delegate** Message MyDelegate**(**string text**);** // делегат возвращает родителя

Email MyMethod**(**string text**)** **=>** **new** Email**(**text**);** // метод возвращает экземпляр //дочернего класса

MyDelegate myDelegate **=** MyMethod**;** // ковариантность

### Контрвариантность

**delegate** void MyDelegate**(**Email message**);** // делегат принимает экземпляр //дочернего класса

void MyMethod**(**Message message**);** // метод принимает родителя

MyDelegate myDelegate **=** MyMethod**;** // контравариантность

### Ковариантность и контравариантность в обобщенных делегатах

out - можно этот тип или родительский

in - можно этот тип или дочерний

#### Ковариантность

**delegate** T MyDelegate**<out** T**>(**string text**);** // делегат

// реализация делегата возвращает экземпляр дочернего типа

MyDelegate**<**Email**>** myMethod1 **=** text **=>** **new** Email**(**text**);**

// реализация делегата возвращает экземпляр родительского типа

MyDelegate**<**Message**>** myMethod2 **=** myDelegate1**;** // ковариантность

Message message **=** myMethod2**(**"Message"**);** // вызов делегата. Возвращение //родительского //типа

#### Контравариантность

**delegate** void MyDelegate**<in** T**>(**T message**);**

// реализация делегата принимает кземпляр родительского типа

MyDelegate**<**Message**>** myMethod1 **=** message **=>** Console**.**WriteLine**(**message**.**text**);**

// реализация делегата принимает кземпляр дочернего типа

MyDelegate**<**Email**>** myMethod2 **=** myDelegate1**;** // контравариантность

myMethod2**(new** Email**(**"Email"**));** // вызов делегата. Использование дочернего //типа

## Делегаты Action, Predicate и Func

### Action

**public** **delegate** void Action**()**

**public** **delegate** void Action**<in** T**>(**T obj**)**

**Принимает от 0 до 16 объектов, ничего не возвращает.**

void Add**(**int x**,** int y**)** **=>** Console**.**WriteLine**(**$**{**x **+** y**});** // метод соответсвует //делегату Action

void DoOperation**(**int a**,** int b**,** Action**<**int**,** int**>** operation**)** // метод принимает //делегат Action

**=>** operation**(**a**,** b**);**

DoOperation**(**5**,** 6**,** Add**);** // 11

### Predicate

**delegate** bool Predicate**<in** T**>(**T obj**);**

**Принимает один параметр и возвращает значение типа bool.**

Predicate**<**int**>** isPositive **=** **(**int x**)** **=>** x **>** 0**;** // проверяет на положительность

### Func

TResult Func**<out** TResult**>()**

TResult Func**<in** T**,** **out** TResult**>(**T obj**)**

**Принимает от 0 до 16 объектов, возвращает результат.**

int Add**(**int x**,** int y**)** **=>** x **+** y**;** // метод соответсвует делегату Func

int DoOperation**(**int a**,** int b**,** Func**<**int**,** int**>** operation**)** // метод принимает //делегат Func

**=>** operation**(**a**,** b**);**

int result **=** DoOperation**(**5**,** 6**,** Add**);** // 11

## Замыкания

**Замыкание (closure) представляет объект функции, который запоминает свое лексическое окружение и может его менять даже при вызове вне своей области видимости.**

**Технически замыкание включает три компонента:**

**- внешняя функция, которая определяет некоторую область видимости**

**- переменные и параметры (лексическое окружение), которые определены во внешней функции**

**- вложенная функция, которая использует переменные и параметры внешней функции**

Action Outer**()** // метод или внешняя функция

**{**

int x **=** 5**;** // лексическое окружение - локальная переменная

void Inner**()** // локальная функция

**{**

x**++;** // операции с лексическим окружением

Console**.**WriteLine**(**x**);**

**}**

**return** Inner**;** // возвращаем локальную функцию

**}**

Action fn **=** Outer**();**

// fn = Inner, так как метод Outer возвращает функцию Inner

// вызываем внутреннюю функцию Inner

fn**();** // 6

fn**();** // 7

fn**();** // 8

### Реализация с помощью лямбда-выражений

Action outerFun **=** **()** **=>**

**{**

int x **=** 5**;**

var innerFun **=** **()** **=>** Console**.**WriteLine**(++**x**);**

**return** innerFun**;**

**};**

Action fn **=** outerFun**();** // fn = innerFun, так как outerFun возвращает innerFn

// вызываем innerFun

fn**();** // 6

fn**();** // 7

fn**();** // 8

### Применение параметров

Func**<**int**,** int**>** Multiply**(**int n**)**

**{**

int Inner**(**int m**)** **=>** n **\*** m**;**

**return** Inner**;**

**}**

Func**<**int**,** int**>** func **=** Multiply**(**5**);**

// var multiply = (int n) => (int m) => n \* m;

// var func = multiply(5);

Console**.**WriteLine**(**func**(**5**));** // 25

Console**.**WriteLine**(**func**(**6**));** // 30

Console**.**WriteLine**(**func**(**7**));** // 35

Интерфейсы

## Определение интерфейсов

**Интерфейсы могут определять:**

**- Методы**

**- Свойства**

**- Индексаторы**

**- События**

**- Статические поля и константы (с C# 8.0)**

**Не могут определять нестатические переменные. Члены интерфейсов public по умолчанию. Сами интерфейсы, как и классы, по умолчанию internal. Но и к членам и к самим интерфейсам можно применять другие модификаторы доступа.**

**interface** IMyInterface

**{**

// константа

const int myConst **=** 0**;**

// статическая переменная

static int myStatic **=** 0**;**

// свойство

string MyProperty **{** **get;** **set;** **}** // это не автосвойство, а свойство без //реализации

// реализация свойства по умолчанию

int MyIntProperty **{** **get** **{** **return** 0**;** **}** **}**

// метод

void MyMethod**();**

// метод с реализацией по умолчанию

void MyMethod2**()** **{** **...** **}** // с C# 8.0

// статический метод должен иметь реализацию по умолчанию

static void MyStaticMethod**()** **{** **...** **}**

**delegate** void MyDelegate**(**string text**);** // определение делегата для //события

// событие

**event** MyDelegate MyEvent**;**

**}**

## Применение интерфейсов

**interface** IMyInterface **{** **...** **}**

class MyClass **:** IMyInterface **{** **...** **}**

struct MyStruct **:** IMyInterface **{** **...** **}**

**Если методы и свойства интерфейса не имеют модификатора доступа, то по умолчанию они public. При их реализации можно применять только public.**

**interface** IMovable

**{**

void Move**()** **=>** **...** // реализация по умолчанию

**}**

class Person **:** IMovable**;**

**...**

Person p **=** **new** Person**();**

// p.Move(); - ошибка, т.к. класс Person не имеет реализации метода Move(), но:

IMovable p **=** **new** Person**();**

p**.**Move**();** // используется реализация по умолчанию интерфейса

Множественная реализация интерфейсов

class MyClass **:** IMyInterface1**,** IMyInterface2 **...** **{** **...** **}**

## Явная реализация интерфейсов

**interface** ISchool

**{**

void Study**();**

**}**

**interface** IUniversity

**{**

void Study**();**

**}**

class Person **:** ISchool**,** IUniversity

**{**

// public void Study() => ...

// реализация метода Stydy() сразу обоих интерфейсов

//или :

**public** ISchool**.**Study**()** **=>** **...** // метод существует только для интерфейса //ISchool

**public** IUniversity**.**Study**()** **=>** **...** // метод существует только для //интерфейса IUniversity

**}**

Person person **=** **new** Person**();**

// person.Study(); ошибка, т.к. в Person нет реализации этого метода, она //только для интерфейсов

**((**ISchool**)** person**).**Study**();**

// или :

// безопасное приведение типа

**if** **(**person **is** IUniversity univirsity1**)** univirsity1**.**Study**();**

// или :

IUniversity univirsity2 **=** **new** Person**();**

univirsity2**.**Study**();**

Модификаторы доступа

**Если члены интерфейса не public, то можно использовать явную реализацию интерфейса без модификаторов:**

**interface** IMyInterface

**{**

**protected** **internal** string Name **{** **get;** **}**

**}**

class MyClass **:** IMyInterface

**{**

string name**;**

// явная реализация свойства в виде автосвойства

string IMyInterface**.**Name **{** **get** **=>** name**;** **}** // обратиться к такому свойству //можно только через объект типа интерфейса

**}**

**Либо использовать неявную реализацию с модификатором public:**

**interface** IMyInterface

**{**

**protected** **internal** string Name **{** **get;** **}**

**}**

class MyClass **:** IMyInterface

**{**

string name**;**

// неявная реализация свойства в виде автосвойства

**public** string Name **{** **get** **=>** name**;** **}** // обратиться к такому свойству можно //через объект типа класса

**}**

## Реализация интерфейсов в базовых и производных классах

**В абстрактном классе, расширяющем интерфейс, можно не реализовывать метод интерфейса, а сделать его абстрактным.**

**interface** MyInterface **{** string MyMethod**();** **}**

**abstract** class MyAbstractClass **{** **abstract** string MyMethod**();** **}**

class MyClass **:** MyAbstractClass **{** **public** **override** string MyMethod**()** **=>** "MyClass"**;** **}**

**Название класса-родителя идет перед названием интерфейсов:**

class MyClass **:** MyBaseClass**,** MyInterface1**,** MyInterface2**...**

**1. Если в классе-родителе и в расширяемом интерфейсе одинаковые методы, то его реализация в родителе переопределяет метод из интерфейса. В дочернем классе тогда его можно не переопределять.**

**interface** MyInterface **{** string MyMethod**();** **}**

class MyBaseClass **{** **public** string MyMethod**();** **}**

class MyClass **:** MyBaseClass**,** MyInterface **{** **}**

**2. Либо переопределить (override), если в родительском классе этот метод virtual (или abstract).**

**interface** MyInterface **{** string MyMethod**();** **}**

class MyBaseClass **:** MyInterface

**{**

**public** **virtual** string MyMethod**()** **=>** "MyBaseClass"**;**

**}**

class MyClass **:** MyBaseClass

**{**

**public** **override** string MyMethod**()** **=>** "MyClass"**;**

**}**

MyBaseClass myBaseClass **=** **new** MyClass**();**

string result **=** myBaseClass**.**MyMethod**();**

// MyClass - т.к. есть переопределенный метод

MyInterface myInterface **=** **new** MyClass**();**

result **=** myInterface**.**MyMethod**();** // MyClass

**3. Либо в дочернем классе скрыть метод**

**interface** MyInterface **{** string MyMethod**();** **}**

class MyBaseClass **:** MyInterface

**{**

**public** string MyMethod**()** **=>** "MyBaseClass"**;**

**}**

class MyClass **:** MyBaseClass // класс наследуется от класса-родителя

**{**

**public** **new** string MyMethod**()** **=>** "MyClass"**;**

**}**

MyBaseClass myBaseClass **=** **new** MyClass**();**

string result **=** myBaseClass**.**MyMethod**();**

// MyBaseClass - т.к. нет переопределенного метода

MyInterface myInterface **=** **new** MyClass**();**

result **=** myInterface**.**MyMethod**();**

// MyBaseClass - т.к. интерфейс реализован только в базовом классе

MyClass myClass **=** **new** MyClass**();**

result **=** myClass**.**MyMethod**();** // MyClass

**4. Либо повторно реализовать интерфейс в классе-наследнике.**

**interface** MyInterface **{** string MyMethod**();** **}**

class MyBaseClass **:** MyInterface

**{**

**public** string MyMethod**()** **=>** "MyBaseClass"**;**

**}**

class MyClass **:** MyBaseClass**,** MyInterface

// класс также напрямую наследуется от интерфейса

**{**

**public** **new** string MyMethod**()** **=>** "MyClass"**;**

**}**

MyBaseClass myBaseClass **=** **new** MyClass**();**

string result **=** myBaseClass**.**MyMethod**();**

// MyBaseClass - т.к. для BaseClass нет переопределенного метода

MyInterface myInterface **=** **new** MyClass**();**

result **=** myInterface**.**MyMethod**();**

// MyClass - т.к. получается, что интерфейс реализован в классе-наследнике

MyClass myClass **=** **new** MyClass**();**

result **=** myClass**.**MyMethod**();** // MyClass

**5. Либо явно реализовать интерфейс**

**interface** MyInterface **{** string MyMethod**();** **}**

class MyBaseClass **:** MyInterface

**{**

**public** string MyMethod**()** **=>** "MyBaseClass"**;**

**}**

class MyClass **:** MyBaseClass**,** MyInterface

// класс также напрямую наследуется от интерфейса

**{**

**public** **new** string MyMethod**()** **=>** "MyClass"**;**

// явная реализация интерфейса

string MyInterface**.**MyMethod**()** **=>** "MyInterface"**;**

**}**

MyBaseClass myBaseClass **=** **new** MyClass**();**

string result **=** myBaseClass**.**MyMethod**();**

// MyBaseClass - т.к. для BaseClass нет переопределенного метода

MyInterface myInterface **=** **new** MyClass**();**

result **=** myInterface**.**MyMethod**();**

// MyInterface - т.к. интерфейс реализован в классе-наследнике

MyClass myClass **=** **new** MyClass**();**

result **=** myClass**.**MyMethod**();** // MyClass

## Наследование интерфейсов

**В отличие от классов к интерфейсам мы не можем применять модификатор sealed, чтобы запретить наследование интерфейсов. Но интерфейсы могут ипользовать new для сокрытия методов.**

**interface** MyInterface1 **{** string MyMethod**()** **=>** "MyInterface1"**;** **}**

**interface** MyInterface2 **{** **new** string MyMethod**)()** **=>** "MyInterface2"**;** **}**

class MyClass **:** MyInterface2 **{}**

MyInterface1 myInterface1 **=** **new** MyClass**();**

string result **=** myInterface1**.**MyMethod**();** // MyInterface1

MyInterface2 myInterface2 **=** **new** MyClass**();**

result **=** myInterface2**.**MyMethod**();** // MyInterface2

**При наследовании интерфейсов, как и классов, дочерний интерфейс должен иметь тот же уровень доступа или более строгий. Т.к., если он будет более открытый, то он откроет и методы и поля своего родительского класса / интерфейса.**

**public** **interface** MyInterface1 **{}**

**internal** **interface** MyInterface2 **:** MyInterface1 **{}**

## Интерфейсы как ограничения обобщений

**interface** MyInterface1 **{}**

**interface** MyInterface2 **{}**

class MyClass1 **:** MyInterface1**,** MyInterface2 **{}**

// ниже параметр типа Т должен реализовывать сразу оба интерфейса, как //MyClass1

class MyClass2**<**T**>** **where** T**:** MyInterface1**,** MyInterface2

**{**

void MyMethod**(**T myInterface**)** **{** **...** **}**

**}**

MyClass1 myClass1 **=** **new** MyClass1**();**

MyClass2**<**MyClass1**>** myClass2 **=** **new** MyClass2**<**MyClass1**>();**

myClass2**.**MyMethod**(**myClass1**);**

### Обобщенные интерфейсы

**interface** MyInterface**<**T**>** **{** void MyMethod**(**T attribute**);** **}**

class MyClass1**<**T**>** **:** MyInterface**<**T**>** **{** **public** void MyMethod**(**T attribute**)** **{** **...** **}** **}**

// можно явно указать, какой будет тип параметра Т

class MyClass2 **:** MyInterface**<**int**>** **{** **public** void MyMethod**(**int attribute**)** **{** **...** **}** **}**

## Копирование объектов. Интерфейс ICloneable

**Реализация интерфейса ICloneable позволяет клонировать объекты.**

### Поверхностное копирование

class MyClass **:** ICloneable

**{**

**public** int Id **{** **get;** **set** **}**

**public** MyClass**(**int id**)** **=>** Id **=** id**;**

**public** **object** Clone**()** // реализация метода интерфейса ICloneable

**{**

**return** **new** MyClass**(**Id**);**

// MemberwiseClone(); - либо так - это поверхностное копирование

// если в классе есть ссылочные поля, то скопируются ссылки этих //полей

**}**

**}**

MyClass myClass **=** **new** MyClass**(**1**);** // id = 1

MyClass myClass2 **=** myClass**.**Clone**();** // создание клона с id = 1

### 

### Глубокое копирование

class Person **:** ICloneable

**{**

**public** Company MyCompany **{** **get;** **set;** **}**

**public** Person**(**Company company**)** **=>** MyCompany **=** company**;**

**public** **object** Clone**()** **=>** **new** Person**(new** Company**());**

**}**

class Company **{}**

## Сортировка объектов. Интерфейс IComparable

**Для сортировки наборов сложных объектов применяется интерфейс IComparable. Он имеет всего один метод:**

**public** **interface** IComparable

**{**

int CompareTo**(object?** o**);**

**}**

**Результат метода:**

**- 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;**

class Person **:** IComparable

**{**

**public** int Age **{** **get;** **set** **}**

**public** int CompareTo**(object?** o**)**

**{**

**if** **(**o **is** Person person**)**

**return** Age**.**CompareTo**(**person**.**Age**);**

// или return Age - person.Age;

**else** **throw** **new** ArgumentException**();**

**}**

**}**

**Интерфейс IComparable имеет обобщенную версию:**

class Person **:** IComparable**<**Person**>**

**{**

**...**

**public** int CompareTo**(**Person**?** person**)** **{** **...** **}**

**}**

### 

### Применение компаратора

**public** **interface** IComparer**<in** T**>**

**{**

int Compare**(**T**?** x**,** T**?** y**);**

**}**

**Компаратор в качестве второго параметра принимает метод Array.Sort(Array array, IComparer comparer).**

class MyComparer **:** IComparer**<**Person**>**

**{**

**public** int Compare**(**Person**?** p1**,** Person**?** p2**)** **{** **...** **}**

**}**

Person**[]** people **=** **new** Person**[...];**

Array**.**Sort**(**people**,** **new** MyComparer**);**

## Ковариантность и контравариантность обобщенных интерфейсов

**- Ковариантность: позволяет использовать более конкретный (дочерний) тип, чем заданный изначально;**

**- Контравариантность: позволяет использовать более универсальный (родительский) тип, чем заданный изначально;**

**- Инвариантность: позволяет использовать только заданный тип.**

### Ковариантные интерфейсы

**out - можно использовать вместо более конкретного (дочернего) типа более общий (родительский)**

// объект, реализующий этот интерфейс с типом Т, можно присвоить объекту //интерфейса с //родительским по отношению к Т типом

**interface** MyInterface**<out** T**>**

**{**

T MyMethod**();** // метод может возвращать тип Т и его родителей

**}**

### Контравариантные интерфейсы

**in - можно использовать вместо более общего (родительского) типа более конкретный (дочерний)**

// объект, реализующий этот интерфейс с типом Т, можно присвоить объекту //интерфейса с //дочерним по отношению к Т типом

**interface** MyInterface**<in** T**>**

**{**

void MyMethod**(**T attribute**);** // в параметры можно положить данный тип или //дочерний

**}**

Дополнительные возможности ООП

## Определение операторов

**public** static возвращаемый\_тип **operator** оператор**(**параметры**)** **{** **}**

**Этот метод должен иметь модификаторы public static. Хотя бы один параметр должен быть типа, в котором он определяется.**

class Counter

**{**

**public** int Value **{** **get;** **set;** **}**

**public** static Counter **operator** **+** **(**Counter c1**,** Counter c2**)**

**=>** **new** Counter **{** Value **=** c1**.**Value **+** c2**.**Value **};**

**public** static Counter **operator** **+** **(**Counter c**,** int val**)**

// перегрузка метода

**=>** **new** Counter **{** Value **=** c**.**Value **+** val **};**

**public** static bool **operator** **>** **(**Counter c1**,** Counter c2**)** **=>** c1**.**Value **>** c2**.**Value**;**

**...**

**}**

**...**

Counter counter3 **=** counter1 **+** counter2**;**

bool result **=** counter1 **>** counter2**;**

**Можно 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 >=**

**Нельзя определить логику операторов равенства = , тернарный оператор ?: и некоторые др.**

### Определение инкремента и декремента

**В коде оператора не должны изменяться те объекты, которые передаются в оператор через параметры.**

// префиксный и постфиксный инкремент будет работать правильно

**public** static Counter **operator** **++** **(**Counter c**)**

**{**

// c.Value += 10; - неправильная логика

// return c;

**return** **new** Counter **{** Value **=** c**.**Value **+** 10 **};**

**}**

Counter c1 **=** **new** Counter **{** Value **=** 10 **};**

**++**c1**;** // 20

Counter c2 **=** c1**++;** // c2 = 20, c1 = 30

### Определение операций true и false

**public** static bool **operator** **true(**Counter c**)** **=>** c**.**Value **!=** 0**;**

**public** static bool **operator** **false(**Counter c**)** **=>** c**.**Value **==** 0**;**

**public** static bool **operator** **!** **(**Counter c**)** **=>** c**.**Value **==** 0**;**

**...**

**if** **(**counter**)** **...** // true

**else** **...** // false

// if (!counter) ... // false

## Перегрузка операций преобразования типов

**explicit - если преобразование явное, то есть нужна операция приведения типов**

**implicit - если преобразование неявное**

**public** static **implicit|explicit** **operator** **Тип\_в\_который\_надо\_преобразовать(исходный\_тип param)**

**{**

// логика преобразования

**}**

**Например,**

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**;**

**}**

**}**

Counter counter1 **=** **new** Counter **{** Seconds **=** 23 **};**

int x **=** **(**int**)**counter1**;** // 23

Counter counter2 **=** x**;** // counter2.Seconds == 23

**Какие операции преобразования делать явными, а какие неявные, в данном случае не столь важно, это решает разработчик по своему усмотрению. Оператор преобразования, определенный в типе Counter, должен либо принимать в качестве параметра объект типа Counter, либо возвращать объект типа Counter.**

## Индексаторы

**Индексатор должен иметь как минимум один параметр.**

**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. Индексатор можно перегружать подобно методам.**

возвращаемый\_тип **this** **[**Тип параметр1**,** **...]**

**{**

**get** **{** **...** **}**

**set** **{** **...** **}**

**}**

**Пример,**

class Person**;**

class Company

**{**

**public** Person**[]** personal**;**

**...**

// индексатор

**public** Person **this[**int index**]**

**{**

// get или set как и в свойствах можно опускать или ограничивать модификаторами

**get** **=>** personal**[**index**];**

**set** **=>** personal**[**index**]** **=** **value;**

**}**

**}**

var company **=** **new** Company **{** personal **=** **new[]** **{** **...** **}** **};**

// получаем объект из индексатора

Person person **=** company**[**0**];**

// переустанавливаем объект

company**[**0**]** **=** **new** Person**();**

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

### Переменная-ссылка

**Переменной-ссылке обязательно надо присвоить значение.**

int x **=** 5**;**

**ref** int xRef **=** **ref** x**;**

### Ссылка как результат функции

**ref** int Method**(**int**[]** array**)** // ref в сигнатуре

**{**

**return** **ref** array**[**0**];** // ref после return

**}**

**ref** int i **=** **ref** Method**(new[]** **{** 100 **});** // ref перед переменной и перед //методом

**Другой пример:**

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**;**

**}**

**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

**Mетоды расширения действуют на уровне пространства имен.**

**public** static class StringExtension

**{**

**public** static char GetChar**(this** string str**,** int i**)** **=>** str**[**i**];** // расширение класса string

**}**

"hello"**.**GetChar**(**1**);** // e

## Частичные классы и методы

**Mы можем иметь несколько файлов с определением одного и того же класса, и при компиляции все эти определения будут скомпилированы в одно.**

Program1**.**cs

**public** **partial** class Program **{** **public** static void Move**()** **{** **...** **}** **}**

Program2**.**cs

**public** **partial** class Program **{** **public** static void Eat**()** **{** **...** **}** **}**

Program**.**Move**();**

Program**.**Eat**();**

### Частичные методы

**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.**

// первая реализация класса и его методов

**public** **partial** class Person

**{**

**public** **partial** void Read**();**

**}**

// вторая реализация класса и его методов

**public** **partial** class Person

**{**

**public** **partial** void Read**()**

**{**

Console**.**WriteLine**(**"I am reading a book"**);**

**}**

**}**

## Анонимные типы

**Свойства анонимных типов доступны только для чтения.**

var user **=** **new** **{** Name **=** "Tom"**,** Age **=** 34 **};**

Console**.**WriteLine**(**user**.**Name**);**

### Инициализаторы с проекцией

class Person

**{**

**public** string Name **{** **get;** **set;** **}**

**public** Person**(**string name**)** **=>** Name **=** name**;**

**}**

Person tom **=** **new** Person**(**"Tom"**);**

int age **=** 34**;**

var student **=** **new** **{** tom**.**Name**,** age **};** // инициализатор с проекцией

Console**.**WriteLine**(**$"{student.Name} {student.age}"**);**

## Кортежи

var tuple **=** **(**5**,** 10**);** // (int, int) tuple = (5, 10);

Console**.**WriteLine**(++**tuple**.**Item1**);** // 6

**можем дать названия полям кортежа:**

**(**string**,** int**,** double**)** person **=** **(**name**:**"Tom"**,** age**:**35**,** height**:**180.5**);**

**можем выполнить декомпозицию кортежа на отдельные переменные:**

var **(**name**,** age**)** **=** **(**"Tom"**,** 23**);**

Console**.**WriteLine**(**name**);** // Tom

Console**.**WriteLine**(**age**);** // 23

**можем выполнить обмен значениями:**

string java **=** "Java"**;**

string cs **=** "C#"**;**

**(**java**,** cs**)** **=** **(**cs**,** java**);**

Console**.**WriteLine**(**java**);** // C#

Console**.**WriteLine**(**cs**);** // Java

// это можно использовать, например, в пузырьковой сортировке массива:

// if (nums[i] > nums[j])

// (nums[i], nums[j]) =(nums[j], nums[i]);

### Кортеж как результат метода

var tuple **=** GetValues**();**

Console**.**WriteLine**(**tuple**.**Item1**);** // 1

Console**.**WriteLine**(**tuple**.**Item2**);** // 3

tuple **=** GetNewValues**();**

Console**.**WriteLine**(**tuple**.**a**);** // 2

Console**.**WriteLine**(**tuple**.**b**);** // 5

**(**int**,** int**)** GetValues**()** **=>** **(**1**,** 3**);**

**(**int a**,** int b**)** GetNewValues**()** **=>** **(**2**,** 5**);**

### Кортеж как параметр метода

PrintPerson**((**"Tom"**,** 37**));** // Tom - 37

void PrintPerson**((**string name**,** int age**)** person**)**

**{**

Console**.**WriteLine**(**$"{person.name} - {person.age}"**);**

**}**

## Records **(**C#9**)**

**Records могут представлять неизменяемый (immutable) тип. Начиная с версии C# 10 добавлена поддержка структур record, соответственно мы можем создавать record-классы и record-структуры.**

**public** record Person // public record class Person { ... }

**{**

**public** string Name **{** **get;** init**;** **}**

// чтобы сделать его действительно неизменяемым,

// необходимо с помощью модификатора init сделать свойства неизменяемыми

**}**

**public** record struct Person **{** **...** **}**

### Сравнение records

**public** record Person

**{**

**public** string **{** **get;** init**;** **}**

**}**

var person1 **=** **new** Person**(**"Tom"**);**

var person2 **=** **new** Person**(**"Tom"**);**

Console**.**WriteLine**(**person1**.**Equals**(**person2**));** // true

Console**.**WriteLine**(**person1 **==** person2**);** // true

### Оператор with

**public** record Person

**{**

**public** string Name **{** **get;** init**;** **}**

**public** string Age **{** **get;** init**;** **}**

**public** Person**(**string name**,** int age**)**

**{**

Name **=** name**;** Age **=** age**;**

**}**

**}**

var tom **=** **new** Person**(**"Tom"**,** 37**);**

var sam **=** tom with **{** Name **=** "Sam" **};**

// можно скопировать полностью: var sam = tom with {};

Console**.**WriteLine**(**$"{sam.Name} - {sam.Age}"**);** // Sam - 37

### Позиционные классы records

**public** record Person**(**string Name**,** int Age**);**

**это аналогично следующей записи:**

**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**);**

**}**

**результат можно использовать так:**

var person **=** **new** Person **(**"Tom"**,** 37**);**

var **(**personName**,** personAge**)** **=** person**;**

Console**.**WriteLine**(**personAge**);** // 37

Console**.**WriteLine**(**personName**);** // Tom

**можно совмещать стандартное определение свойств и определение свойств через конструктор:**

var person **=** **new** Person**(**"Tom"**,** 37**)** **{** Company **=** "Google"**};**

**public** record Person**(**string Name**,** int Age**)** // доступны только для чтения

// (для свойств класса record по умолчанию устанавливается модификатор init)

**{**

**public** string Company **{** **get;** **set;** **}** **=** ""**;** // можно изменять

**}**

### Позиционные структуры для чтения

**public** record struct Person**(**string Name**,** int Age**);**

// по умолчанию свойства структур record имеют стандартные сеттеры

**public** **readonly** record struct Person**(**string Name**,** int Age**);**

// readonly record структура имеет свойства с модификатором init

### ToString

**Для record реализован ToString по-умолчанию.**

### Наследование

**public** record Person**(**string Name**,** int Age**);**

**public** record Employee**(**string Name**,** int Age**,** string Company**)** **:** Person**(**Name**,** Age**);**

Pattern matching

## Паттерн типов

### type pattern

**значение is тип переменная\_типа**

**Например**

**if** **(**employee **is** Manager manager**)** manager**.**Go**();**

// происходит создание переменной manager типа Manager через приведение типа Employee к Manager

### constant pattern

**if** **(**message **is** "hello"**)** **...**

**или**

**if** **(**message **is** **null)** **...**

**if** **(**message **is** not **null)** **...**

### в конструкции switch:

**switch** **(**employee**)**

**{**

**case** Manager manager**:**

// приведение к типу Manager и сохранение в переменной manager

manager**.**Go**();**

**break;**

**case** **null:**

**...**

**break;**

**...**

**}**

### выражение when в конструкции switch:

Employee employee **=** **new** Manager**()** **{** Value **=** **true** **};**

**switch** **(**employee**)**

**{**

**case** Manager manager **when** manager**.**Value**:**

manager**.**Go**();**

**break;**

**...**

**}**

## Паттерн свойств

**{**свойство**:** значение**}**

class Person

**{**

**public** string Name **{** **get;** **set;** **}** **=** ""**;**

**public** string Status **{** **get;** **set;** **}** **=** ""**;**

**public** string Language **{** **get;** **set;** **}** **=** ""**;**

**}**

**Проверка по классу и свойству:**

**if** **(**person **is** Person **{** Language**:** "english"**,** Status**:** "admin" **}** **...**

**В конструкции switch:**

string message **=** person **switch**

**{**

**{** Language**:** "french" **}** **=>** "Salut!"**,**

// можно определять переменные (например, name) и передавать им значения //свойств

**{** Language**:** "english"**,** Name**:** var name **}** **=>** $"Hello, {name}"**,**

**...**

**}**

**С C#10:**

class Employee

**{**

**public** Company MyCompany **{** **get;** **set;** **}**

**}**

class Company

**{**

**public** string Title **{** **get;** **}**

**}**

Employee employee **=** **new** Employee **{** MyCompany **=** **new** Company **{** Title **=** "Microsoft" **}** **};**

**if** **(**employee **is** Employee **{** Company**:** **{** Title**:** "Microsoft" **}** **})** **...**

**то же самое:**

**if** **(**employee **is** Employee **{** Company**.**Title**:** "Microsoft" **})** **...**

## Паттерны кортежей

string MyMethod**(**string a**,** string b**)** **=>** **(**a**,** b**)** **switch**

**{**

**(**"C"**,** "D"**)** **=>** "CD"**,**

**(**\_**,** "E"**)** **=>** "E"**,**

\_ **=>** "F"

**};**

## Позиционный паттерн

**Позиционный паттерн применяется к типу, у которого определен метод деконструктора.**

class MyClass

**{**

**public** string A **{** **get;** **set;** **}** **=** ""**;**

**public** string B **{** **get;** **set;** **}** **=** ""**;**

**public** void Deconstruct**(out** string a**,** **out** string b**)**

**{**

a **=** A**;**

b **=** B**;**

**}**

**}**

string MyMethod**(**MyClass myClass**)** **=>** myClass **switch**

**{**

**(**"C"**,** "D"**)** **=>** "CD"**,**

**(**\_**,** "E"**)** **=>** "E"**,**

**(**var a**,** var b**)** **=>** $"{a} {b}"**;**

\_ **=>** "F"

**};**

## Реляционный и логический паттерны **(**C#9.0**)**

### relational pattern

int MyMethod**(**int a**)** **=>** a **switch**

**{**

**<** 0 **=>** 0**,**

**>** 10 **=>** 2**,**

\_ **=>** 1

**}**

### logical pattern

int MyMethod**(**int a**)** **=>** a **switch**

**{**

**<** 0 or **>** 10 **=>** 0**,**

**>=** 3 and **<=** 5 **=>** 2**,**

not 7 **=>** 1**,**

\_ **=>** 7

**}**

## Паттерны списков **(**C#11**)**

string MyMethod**(**int**[]** array**)** **=>** array **switch**

**{**

**[**1**,** 2**,** 3**]** **=>** "1"**,**

**[**1**,** 2**,** **..,** 5**]** **=>** "2"**,**

// соответствует массиву из произвольного количества элементов,

// первые 2 из которых - 1, 2, затем - любые, а заканчивается на 5

**[**1**,**\_**]** **=>** "3"**,**

// соответствует массиву из 2 элементов,

// первый из которых - 1, а второй - любое число

**[**\_**,** **..,** \_**]** **=>** "4"**,**

// соответствует массиву как минимум из 2 элементов

**[**var a**,** var b**,** **..,** var d**]** **=>** $"{a}, {b}, {d}"**,**

// получение значений из элементов массива

**[**1**,** **..** var a**,** 2**]** **=>** $"{string.Join("**,**", a)}"**,**

// a - массив, в котором сохраняются элементы между 1 и 2

**[]** **=>** "6"**,**

\_ **=>** "7"

**};**

**Аналогично со списками. Также можно использовать в конструкции if:**

**if** **(**array **is** **[**"a"**,** "b"**,** "c"**])** **...**

### Свойства коллекций

// можно объединить паттерн списков и паттерн свойств

int **[]** array **=** **{** 1**,** 2**,** 3 **};**

**if** **(**array **is** **{** Length**:** 3**}** and **[**var a**,** var b**,** var c**])** // свойство массива Length

Console**.**WriteLine**(**$"{a}, {b}, {c}"**);**

Коллекции

## Список List**<**T**>**

**Большая часть классов коллекций содержится в пространстве имен** System**.**Collections**.**Generic

### Создание пустого списка

List**<**string**>** list **=** **new** List**<**string**>();**

### Создание списка, содежащего начальные значения

List**<**string**>** list **=** **new** List**<**string**>()** **{** "A"**,** "B"**,** "C" **};**

### Создание списка с элементами из другой коллекции

var list **=** **new** List**<**string**>()** **{** "A"**,** "B"**,** "C" **};**

var newList **=** **new** List**<**string**>(**list**);**

**или можно так:**

var newList **=** **new** List**<**string**>(**list**)** **{** "D"**,** "E" **};**

### С C#12

List**<**string**>** list **=** **{** "A"**,** "B"**,** "C" **};**

List**<**string**>** emptyList **=** **[];** // пустой список

### Установка начальной емкости списка

List**<**string**>** list **=** **new** List**<**string**>(**10**);**

**или так:**

list**.**Capacity **=** 10**;**

### Обращение к элементам списка

string element **=** list**[**0**];** // обращение к первому элементу коллекции

### Длина списка

int size **=** list**.**Count**;**

### Методы списка

**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**

## Двухсвязный список LinkedList**<**T**>**

### Создание связанного списка

LinkedList**<**string**>** list **=** **new** LinkedList**<**string**>();**

**можно из List сделать LinkedList так:**

List**<**string**>** list **=** **new** List**<**string**>()** **{** "A"**,** "B"**,** "C" **};**

LinkedList**<**string**>** linkedList **=** **new** LinkedList**<**string**>(**list**);**

### Свойства LinkedList

**- 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>**

### Свойства LinkedListNode

**- 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**

### Циклы:

// от начала до конца

var node **=** linkedList**.**First**;**

**while(**node **!=** **null)**

**{**

Console**.**WriteLine**(**node**.**Value**);**

node **=** node**.**Next**;**

**}**

// от конца до начала

var node **=** linkedList**.**Last**;**

**while(**node **!=** **null)**

**{**

Console**.**WriteLine**(**node**.**Value**);**

node **=** node**.**Previous**;**

**}**

### Методы LinkedList

**- 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**

## Очередь Queue

### Создание очереди

Queue**<**string**>** queue **=** **new** Queue**<**string**>();**

**можно указать емкость очереди:**

Queue**<**string**>** queue **=** **new** Queue**<**string**>(**16**);**

**также можно инициализировать очередь элементами из другой коллекции или массивом:**

var list **=** **new** List**<**string**>** **{** "A"**,** "B"**,** "C" **};**

Queue**<**string**>** queue **=** **new** Queue**<**string**>(**list**);**

**для перебора можно использовать цикл foreach:**

**foreach** **(**var q **in** queue**)** **...**

queue**.**Count **- количество элементов в очереди**

### Методы Queue

**- void Clear(): очищает очередь**

**- bool Contains(T item): возвращает true, если элемент item имеется в очереди**

**- T Dequeue(): извлекает и возвращает первый элемент очереди (есть Exception)**

**- void Enqueue(T item): добавляет элемент в конец очереди**

**- T Peek(): просто возвращает первый элемент из начала очереди без его удаления (есть Exception)**

**- 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**

## Коллекция Stack**<**T**>**

### Создание стека

**можно создать пустой стек:**

Stack**<**string**>** stack **=** **new** Stack**<**string**>();**

**можно указать емкость стека:**

Stack**<**string**>** stack **=** **new** Stack**<**string**>(**16**);**

**можно инициализировать стек элементами из другой коллекции или массивом:**

var list **=** **new** List**<**string**>** **{** "A"**,** "B"**,** "C" **};**

Stack**<**string**>** stack **=** **new** Stack**<**string**>(**list**);**

**можно использовать цикл foreach:**

**foreach** **(**var s **in** stack**)** **...**

stack**.**Count **- показывает количество элементов в коллекции**

### Методы Stack

**- 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 (имеет Exception)**

**- Peek: просто возвращает первый элемент из стека без его удаления (имеет Exception)**

**- 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**

## Коллекция Dictionary**<**K**,** V**>**

### Создание и инициализация словаря

Dictionary**<**int**,** string**>** dictionary **=** **new** Dictionary**<**int**,** string**>();**

**или можно сразу инициализировать с помощью инициализатора:**

var dictionary **=** **new** Dictionary**<**int**,** string**>()**

**{**

**{** 1**,** "A" **},**

**{** 2**,** "B" **},**

**{** 3**,** "C" **}**

**};**

**или так:**

var dictionary **=** **new** Dictionary**<**int**,** string**>()**

**{**

**[**1**]** **=** "A"**,**

**[**2**]** **=** "B"**,**

**[**3**]** **=** "C"

**};**

### KeyValuePair

**Каждый элемент в словаре представляет структуру KeyValuePair<TKey, TValue>**

var keyValuePair **=** **new** KeyValuePair**<**int**,** string**>(**1**,** "A"**);**

var list **=** **new** List**<**KeyValuePair**<**int**,** string**>>()** **{** keyValuePair **};**

var dictionary **=** **new** Dictionary**<**int**,** string**>(**list**);**

**Можно объединить 2 способа инициализации:**

var dictionary **=** **new** Dictionary**<**int**,** string**>(**list**)**

**{**

**[**2**]** **=** "B"**,**

**[**3**]** **=** "C"

**};**

### Перебор словаря

**можно использовать цикл foreach**

**foreach(**var keyValuePair **in** dictionary**)** **...**

### Получение элементов

**словарь[ключ]**

var dictionary **=** **new** Dictionary**<**int**,** string**>()**

**{**

**[**1**]** **=** "A"**,**

**[**2**]** **=** "B"**,**

**[**3**]** **=** "C"

**};**

// получаем элемент по ключу

string s **=** dictionary**[**2**];** // B

// переустанавливаем значение по ключу

dictionary**[**1**]** **=** "D"**;**

// добавляем новый элемент

dictionary**[**4**]** **=** "F"**;**

### Методы и свойства Dictionary

**- 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**

**- bool Remove(K key, out V value): позволяет получить удленный элемент в выходной параметр**

**- 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**

**dictionary.Count - возвращает количество элементов**

## Класс ObservableCollection

**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

**using** System**.**Collections**.**ObjectModel**;**

ObservableCollection**<**string**>** collection **=** **new** ObservableCollection**<**string**>();**

**или с инициализацией через другую коллекцию или массив:**

var collection **=** **new** ObservableCollection**<**string**>(**

**new** string**[]** **{** "A"**,** "B"**,** "C" **}**

**);**

**или так:**

var collection **=** **new** ObservableCollection**<**string**>**

**{**

"A"**,** "B"**,** "C"

**};**

**или можно сочетать оба способа инициализации:**

var collection **=** **new** ObservableCollection**<**string**>(new** string**[]** **{** "A"**,** "B"**,** "C" **})**

**{**

"D"**,** "E"**,** "F"

**};**

### Обращение к элементам коллекции

collection**[**0**]** **=** "D"**;** // изменяем первый элемент

string s **=** collection**[**1**];** // получаем второй элемент

### Перебор коллекции

**можно применять стандартные циклы**

### Методы и свойства ObservableCollection

**collection.Count - количество элементов коллекции**

**- 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**

### Уведомление об измении коллекции

**CollectionChanged - событие, накоторое можно подписаться**

**NotifyCollectionChangedEventHandler - делегат, представляющий событие**

**NotifyCollectionChangedEventArgs - объект, который хранит всю информацию о событии. В частности его свойство Action позволяет узнать характер изменений:**

**NotifyCollectionChangedAction.Add: добавление**

**NotifyCollectionChangedAction.Remove: удаление**

**NotifyCollectionChangedAction.Replace: замена**

**NotifyCollectionChangedAction.Move: перемещение объекта внутри коллекции**

**NotifyCollectionChangedAction.Reset: сброс содержимого коллекции**

**Свойства NewItems и OldItems позволяют получить добавленные и удаленные объекты.**

void NotifyCollectionChangedEventHandler**(object?** sender**,** NotifyCollectionChangedEventArgs e**);**

### Пример

class MyObject**;**

**using** System**.**Collections**.**ObjectModel**;**

**using** System**.**Collections**.**Specialized**;**

// обработчик изменения коллекции

void MyNotifyCollectionChangedEventHandler**(object?** sender**,** NotifyCollectionChangedEventArgs e**)**

**{**

**switch** **(**e**.**Action**)**

**{**

**case** NotifyCollectionChangedAction**.**Add**:** // если добавление

**if(**e**.**NewItems**?[**0**]** **is** MyObject myObject**)** **...** myObject **...**

**break;**

**case** NotifyCollectionChangedAction**.**Remove**:** // если удаление

**if(**e**.**OldItems**?[**0**]** **is** MyObject myObject**)** **...** myObject **...**

**break;**

**case** NotifyCollectionChangedAction**.**Replace**:** // если замена

**if((**e**.**NewItems**?[**0**]** **is** MyObject newMyObject**)**

**&&** **(**e**.**OldItems**?[**0**]** **is** MyObject oldMyObject**))** **...** newMyObject **...** oldMyObject **...**

**break;**

**}**

**}**

var collection **=** **new** ObservableCollection**<**MyObject**>()**

**{**

**new** MyObject**(),**

**new** MyObject**()**

**}**

// подписываемся на событие изменения коллекции

collection**.**CollectionChanged **+=** MyNotifyCollectionChangedEventHandler**;**

collection**.**Add**(new** MyObject**());** // вызывается обработчик изменения коллекции

collection**.**RemoveAt**(**0**);** // вызывается обработчик изменения коллекции

collection**[**0**]** **=** **new** MyObject**();** // вызывается обработчик изменения коллекции

## Интерфейсы IEnumerable и IEnumerator

**Перебераемая с помощью foreach коллекция должна реализовывать интерфейс IEnumerable**

**foreach(**var item **in** перечисляемый\_объект**)** **...**

**IEnumerable имеет метод, возвращающий ссылку на перечислитель IEnumerator**

**public** **interface** IEnumerable

**{**

IEnumerator GetEnumerator**();**

**}**

**IEnumerator определяет функционал для перебора внутренних объектов в контейнере**

**public** **interface** IEnumerator

**{**

bool MoveNext**();** // перемещение на одну позицию вперед

// если коллекция закончилась, вернет false

// можно использовать, например, так:

// while (collection.MoveNext()) ...

**object** Current **{** **get;** **}** // текущий объект в контейнере

void Reset**();** // перемещение в начало контейнера

**}**

## Итераторы и оператор **yield**

**Итератор это блок кода, который использует оператор yield для перебора набора значений.**

**Итератор использует две специальных инструкции:**

* **yield return: определяет возвращаемый элемент**
* **yield break: указывает, что последовательность больше не имеет элементов**

### Пример с расширением Int32

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**;**

**}**

**}**

**foreach(**var n **in** 5**)** Console**.**Write**(**$"n "**);** // 0 1 2 3 4 5

**foreach(**var n **in** **-**5**)** Console**.**Write**(**$"n "**);** // -5 -4 -3 -2 -1 0

**При каждом вызове оператора yield return итератор будет запоминать текущее местоположение и при последующих вызовах начинать с него.**

### Именованный итератор

**Именованные итераторы - это методы, использующие yield return. Они должны возвращать IEnumerable.**

**public** IEnumerable**<**T**>** MyMethod**(**T**[]** array**)**

**{**

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

**{**

**if** **(**i **==** 5**)**

**yield** **break;** // этот оператор вернет 5 и прервет итератор

**else**

**yield** **return** array**[**i**];**

**}**

**}**

## Класс Array и массивы

### Свойства

**- Length: возвращает длину массива**

**- Rank: возвращает размерность массива**

### Методы

**- 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 индекса 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 (имеет перегрузки)**

**- void Sort (Array array): сортирует элементы одномерного массива (имеет много перегрузок)**

## Span

**Span - структура, цель которой - повысить производительность использования памяти**

### Создание Span

**- 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

Span**<**string**>** span **=** **[** "A"**,** "B"**,** "C" **];**

**или так:**

string**[]** array **=** **{** "A"**,** "B"**,** "C" **};**

Span**<**string**>** span **=** **new** Span**<**string**>(**array**);**

**или так:**

string**[]** array **=** **{** "A"**,** "B"**,** "C" **};**

Span**<**string**>** span **=** array**;** // неявное преобразование массива в Span

**Работать со Span можно, как с массивом.**

### 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, но предназначена для неизменяемых данных.**

**С помощью метода AsSpan() у string можно получить ReadOnlySpan из символов строки:**

ReadOnlySpan**<**char**>** readOnlySpan **=** "Hello"**.**AsSpan**();**

### Ограничения Span

**- не может быть присвоена переменной типа интерфейса, Object или dynamic.**

**- не может быть полем в объекте ссылочного типа (только внутри ref-структур)**

**- не может использоваться в пределах операций await или yield**

## Индексы и диапазоны

### Индексы

Index myIndex **=** 2**;** // третий с начала элемент

**или так:**

Index myIndex **=** **^**3**;** // третий с конца элемент

**Например, array[^4];**

### Диапазоны

**Начальный индекс включается в диапазон, а конечный индекс НЕ входит в диапазон.**

Range myRange **=** 1..4**;** // со 2-го включая до 5 не включая

**или так:**

Range myRange **=** 1..**^**1**;**

**или так:**

Range myRange **=** **.**.4**;** // 0..4

**или так:**

Range myRange **=** 1..**;** // 1 - до конца

**Например, array[^4..6];**

Работа со строками

## Строки и класс String

**string является псевдонимом для класса String. Максимальный размер объекта String может составлять в памяти 2 ГБ, или около 1 миллиарда символов.**

### Создание строк

string s **=** "String"**;**

string s **=** **new** String**(**'s' 5**);** // sssss

string s **=** **new** String**(new** char**[]** **{** 'S'**,** 't'**,** 'r'**,** 'i'**,** 'n'**,** 'g' **});** // String

string s **=** **new** String**(new** char**[]** **{** '0'**,** '1'**,** '2'**,** '3'**,** '4' **},** 2**,** 2**);** // 23 // (array, startIndex, count)

### Строка как набор символов

string s **=** "hello"**;**

char ch **=** s**[**1**];** // e

int count **=** s**.**Length**;** // 5

### Перебор строк

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

**foreach(**char ch **in** s**)** **...**

### Сравнение строк

**строки сравниваются по значению их символов, а не по ссылкам:**

bool b **=** "hello" **==** "hello"**;** // true

### Многострочные строки

### C C# 11

string text **=** """

Я помню чудное мгновение**,**

Передо мной явилась ты**,**

Как мимолетное видение**...**

""";

int i **=** 100**;**

string text **=** """

Hello**,**

**{**i**}**

World

**!**

""";

### Основные методы строк

**- 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**

## Операции со строками

### Объединение строк

string s1 **=** "Hello "**;**

string s2 **=** "world"**;**

string res1 **=** s1 **+** s2**;** // Hello world

string res2 **=** string**.**Concat**(**s1**,** s2**);** // Hello world

string**[]** strs **=** **new** string**[]** **{** "A"**,** "B"**,** "C" **};**

string res3 **=** string**.**Join**(**", "**,** strs**);** // A, B, C // (разделитель, массив)

### Сравнение строк

int result **=** string**.**Compare**(**"A"**,** "S"**);** // -1

### Поиск в строке

int i **=** "Hello"**.**IndexOf**(**'l'**);** // 2

int i **=** "Hello"**.**IndexOf**(**"llo"**);** // 2

int i **=** "Hello"**.**LastIndexOf**(**'l'**);** // 3

bool b **=** "file.exe"**.**EndsWith**(**".exe"**);** // true

bool b **=** "C:\Files\file.exe"**.**StartsWith**(**"C:"**);** // true

### Разделение строк

string**[]** words **=** "A B C"**.**Split**(new** char**[]** **{** ' ' **},** StringSplitOptions**.**RemoveEmptyEntries**);**

// StringSplitOptions.RemoveEmptyEntries - значит удалить все пустые подстроки

**или**

string**[]** words **=** "A B C"**.**Split**(**' '**);**

### Обрезка строки

string text **=** " Hello world "**.**Trim**();** // "Hello world"

string text **=** " Hello world "**.**Trim**(new** char**[]** **{** 'd'**,** 'h' **});** // "ello worl"

string text **=** " Hello world "**.**TrimStart**();** // "Hello world "

string text **=** " Hello world "**.**TrimEnd**();** // " Hello world"

string text **=** "Hello world"**.**Substring**(**2**);** // llo world

string text **=** "Hello world"**.**Substring**(**2**,** 3**);** // llo

### Вставка

string text **=** "Hello world"**.**Insert**(**5**,** " my"**);** // Hello my world

### Удаление строк

string text **=** "Hello"**.**Remove**(**2**);** // He

string text **=** "Hello"**.**Remove**(**1**,** 2**);** // Hlo

### Замена

string text **=** "Hello world"**.**Replace**(**"l"**,** ""**);** // Heo word

### Смена регистра

string text **=** "Hello world"**.**ToLower**();** // hello world

string text **=** "Hello world"**.**ToUpper**();** // HELLO WORLD

## Форматирование и интерполяция строк

### Форматирование строк

string name **=** "Tom"**;**

int age **=** 20**;**

string**.**Format **/** Console**.**WriteLine **(**"Name: {0}, Age: {1}"**,** name**,** age**);**

### Спецификаторы форматирования

double num **=** 20.4567**;**

int digit **=** 20**;**

**C / c - Задает формат денежной единицы, указывает количество десятичных разрядов после запятой;**

string**.**Format**(**"{0}:C2"**,** num**);**

// 20.46 $ - округление + валюта в зависимости от культуры компьютера

**D / d - Целочисленный формат, указывает минимальное количество цифр;**

string**.**Format**(**"{0:d4}"**,** digit**);**

// 0020

**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;**

string**.**Format**(**"{0}:f}"**,** digit**);** // 20.00

string**.**Format**(**"{0}:f2}"**,** num**);** // 20.46 - округление

**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 отображение знака процентов рядом с числом, указывает количество десятичных разрядов после запятой;**

string**.**Format**(**"{0:P2}"**,** num**);** // 20.46%

**X / x - Шестнадцатеричный формат числа;**

**# - Moжно настроить формат вывода;**

long phoneNumber **=** 89511234567

string**.**Format**(**"{0:+# (###) ###-##-##}"**,** phoneNumber**);** // +8 (951) 123-45-67

### Метод ToString

**Так же, как и string.Format и Console.WriteLine, метод ToString поддерживает спецификаторы форматирования.**

long phoneNumber **=** 89511234567

phoneNumber**.**ToString**(**"# (###) ###-##-##"**);** // +8 (951) 123-45-67

### Интерполяция строк

string**.**Format **/** Console**.**WriteLine **(**$"Name: {name}, Age: {age}"**);**

string**.**Format**(**$"{x} \* {y} = {Multiply(x,y)}"**);**

// можно вызывать метод

string**.**Format**(**$"{phoneNumber:+# (###) ###-##-##}"**);**

// +8 (951) 123-45-67

string**.**Format**(**$"Name: {name, -5}, Age: {age}"**);**

// пробелы после: Name: Tom , Age: 20

string**.**Format**(**$"Name: {name, 5}, Age: {age}"**);**

// пробелы до: Name: Tom, Age: 20

## Класс StringBuilder

### Создание StringBuilder

**using** System**.**Text**;**

**Length - длина строки**

**Capacity - емкость билдера**

StringBuilder builder **=** **new** StringBuilder**();**

// начальная емкость 16 по дефолту

StringBuilder builder **=** **new** StringBuilder**(**32**);**

// начальная емкость 32

StringBuilder builder **=** **new** StringBuilder**(**"Hello world"**);**

StringBuilder builder **=** **new** StringBuilder**(**"Hello world"**,** 32**);**

// начальная емкость 32

string str **=** builder**.**ToString**();** // Hello world

### Операции со строками в StringBuilder

* **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**

### 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

**using** System**.**Text**.**RegularExpressions**;**

string text **=** "... "**;**

Regex regex **=** **new** Regex**(**@"regex..."**);**

MatchCollection matches **=** regex**.**Matches**(**text**);**

**if** **(**matches**.**Count **>** 0**)**

**{**

**foreach** **(**Match match **in** matches**)**

Console**.**WriteLine**(**match**.**Value**);**

**}** // else - совпадений не найдено

### Параметр RegexOptions

**Две версии конструкторов в качестве одного из параметров принимают перечисление RegexOptions.**

**- 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**

Regex regex **=** **new** Regex**(**@"regex..."**,** RegexOptions**.**IgnoreCase**);**

Regex regex **=** **new** Regex**(**@"regex..."**,** RegexOptions**.**IgnoreCase

**|** RegexOptions**.**Compiled**);**

### Синтаксис регулярных выражений

[Элементы языка регулярных выражений — краткий справочник**.**](https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference)

**^: 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**

**{n} : соответствует n символам. Например, @”\d{3}” - 3 цифры**

**[a-z] : соответствует символам из диапазона внутри квадратных скобочек. Например, @”[0-9]” - все цифры**

**| : задает альтернативные символы. Например, @”(2|3)” - 2 или 3**

**\ : экранирует символы. Например, @”\\*” - это просто символ звездочки**

### Проверка на соответствие строки формату (IsMatch)

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}))$"**;**

// проверка email от Microsoft

var text **=** "tom@gmail.com"**;**

Regex**.**IsMatch**(**text**,** pattern**,** RegexOptions**.**IgnoreCase**);** // true

### Замена и метод Replace

string phoneNumber **=** "+8 (912) 345-67-89"**;**

string pattern **=** @"\D"**;** // все не цифры

string targer **=** ""**;** // будут удалены

Regex regex **=** **new** Regex**(**pattern**);**

string result **=** regex**.**Replace**(**phoneNumber**,** target**);** // 89123456789

Работа с датами и временем

## Структура DateTime

**DateTime dateTime = new DateTime(); // 01.01.0001 0:00:00**

**// DateTime.MinValue - 01.01.0001 0:00:00**

**DateTime date1 = new DateTime(2015, 7, 20); // год - месяц - день**

**// 20.07.2015 0:00:00**

**DateTime date1 = new DateTime(2015, 7, 20, 18, 30, 25);**

**// год - месяц - день - час - минута - секунда**

**// 20.07.2015 18:30:25**

**DateTime.Now - 20.07.2015 11:43:33 - текущая дата и время с компьютера**

**DateTime.UtcNow - 20.07.2015 8:43:33 - дата и время относительно времени по Гринвичу (GMT)**

**DateTime.Today - 20.07.2015 0:00:00 - только текущая дата**

### Операции с DateTime

**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**

**Для вычитания дат используется метод:**

**- Subtract(DateTime date)**

**есть ряд методов форматирования дат:**

**DateTime date1 = new DateTime(2015, 7, 20, 18, 30, 25);**

**- ToLocalTime() : 20.07.2015 21:30:25 - добавляет смещение относительно времени по Гринвичу**

**- ToUniversalTime() : 20.07.2015 15:30:25 - вычитает смещение относительно времени по Гринвичу**

**- ToLongDateString() : 20 июля 2015 г.**

**- ToShortDateString() : 20.07.2015**

**- ToLongTimeString() : 18:30:25**

**- ToShortTimeString() : 18:30**

## Форматирование дат и времени

|  |  |
| --- | --- |
| **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** |

**DateTime now = DateTime.Now;**

**Console.WriteLine($"D: {now.ToString("D")}"); // D: 6 января 2022 г.**

**Console.WriteLine($"d: {now.ToString("d")}"); // d: 06.01.2022**

**Console.WriteLine($"F: {now.ToString("F")}"); // F: 6 января 2022 г. 14:45:20**

**Console.WriteLine($"f: {now:f}"); // f: 6 января 2022 г. 14:45**

**Console.WriteLine($"G: {now:G}"); // G: 06.01.2022 14:45:20**

**Console.WriteLine($"g: {now:g}"); // g: 06.01.2022 14:45**

**Console.WriteLine($"M: {now:M}"); // M: 6 января**

**Console.WriteLine($"O: {now:O}"); // O: 2022-01-06T14:45:20.3942344+04:00**

**Console.WriteLine($"o: {now:o}"); // o: 2022-01-06T14:45:20.3942344+04:00**

**Console.WriteLine($"R: {now:R}"); // R: Thu, 06 Jan 2022 14:45:20 GMT**

**Console.WriteLine($"s: {now:s}"); // s: 2022-01-06T14:45:20**

**Console.WriteLine($"T: {now:T}"); // T: 14:45:20**

**Console.WriteLine($"t: {now:t}"); // t: 14:45**

**Console.WriteLine($"U: {now:U}"); // U: 6 января 2022 г. 10:45:20**

**Console.WriteLine($"u: {now:u}"); // u: 2022-01-06 14:45:20Z**

**Console.WriteLine($"Y: {now:Y}"); // Y: январь 2022 г.**

### Настройка формата времени и даты

|  |  |
| --- | --- |
| **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.** |

**DateTime now = DateTime.Now;**

**now.ToString("hh:mm:ss") // 02:46:38**

**now.ToString("dd.MM.yyyy") // 06.01.2022**

## DateOnly и TimeOnly

### DateOnly

#### Конструкторы:

**DateOnly() // 01.01.0001**

**DateOnly(int year, int month, int day)**

**DateOnly(int year, int month, int day, System.Globalization.Calendar calendar)**

**DateOnly now = new DateOnly(2022,1,6);**

#### Свойства DateOnly:

**- Day: возвращает день даты**

**now.Day // 6**

**- DayNumber: возвращает количество прошедших дней с 1 января 0001 года относительно григорианского календаря**

**now.DayNumber // 738160**

**- DayOfWeek: возвращает день недели**

**now.DayOfWeek // Thursday**

**- DayOfYear: возвращает день года**

**now.DayOfYear // 6**

**- MaxValue: возвращает максимально возможную дату (статическое свойство)**

**- MinValue: возвращает самую раннюю возможную дату (статическое свойство)**

**- Month: возвращает месяц**

**now.Month // 1**

**- Year: возвращает год**

**now.Year // 2022**

#### Методы DateOnly:

**- 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**

#### Статические методы DateOnly:

**- 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**

### TimeOnly

#### Конструкторы:

**TimeOnly() // 0:00**

**TimeOnly(long ticks)**

**TimeOnly(int hour, int minute)**

**TimeOnly(int hour, int minute, int second)**

**TimeOnly(int hour, int minute, int second, int millisecond)**

#### Свойства TimeOnly:

**- 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)**

#### Методы TimeOnly:

**- 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**

#### Статические методы TimeOnly:

**- 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**

Дополнительные классы и структуры .NET

## Отложенная инициализация и тип Lazy

**class Reader**

**{**

**Lazy<Library> library = new Lazy<Library>();**

**public void ReadBook()**

**{**

**library.Value.GetBook(); // library.Value - экземпляр класса Library**

**// экземпляр класса Library будет создан при первом вызове library.Value**

**Console.WriteLine("Read book");**

**}**

**public void ReadEbook() => Console.WriteLine("Read the ebook");**

**}**

**class Library**

**{**

**private string[] books = new string[99];**

**public void GetBook() => Console.WriteLine("Get the book");**

**}**

**Reader reader = new Reader(); // экземпляр класса Library еще не создан**

**reader.ReadEbook(); // ... пока не создан**

**reader.ReadBook(); // теперь создан**

## Математические вычисления и класс Math

### Основные методы класса Math

* **Abs(double value): возвращает абсолютное значение для аргумента value**

**double result = Math.Abs(-12.4); // 12.4**

* **Acos(double value): возвращает арккосинус value. Параметр value должен иметь значение от -1 до 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**

**double result = Math.BigMul(100, 9340); // 934000**

* **Ceiling(double value): возвращает наименьшее целое число с плавающей точкой, которое не меньше value**

**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**

**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**

**double result = Math.Floor(2.56); // 2**

* **IEEERemainder(double a, double b): возвращает остаток от деления a на b**

**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**

**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**

**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**

**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**

**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**

**double result = Math.Truncate(16.89); // 16**

### Константы

**Math.PI - число Пи**

**Math.E - экспонента**

## Преобразование типов и класс Convert

### Метод Parse

**int a = int.Parse("10");**

**double b = double.Parse("23,56");**

**decimal c = decimal.Parse("12,45");**

**byte d = byte.Parse("4");**

**// парсинг дробных чисел зависит от настроек текущей культуры**

**Чтобы не зависеть от культурных различий мы можем установить четкий формат с помощью класса NumberFormatInfo и его свойства NumberDecimalSeparator:**

**using System.Globalization;**

**IFormatProvider formatter = new NumberFormatInfo**

**{**

**NumberDecimalSeparator = "."**

**};**

**double number = double.Parse("23.56", formatter);**

### Метод TryParse

**bool TryParse(string input, out var number):**

**string? input = Console.ReadLine();**

**bool result = int.TryParse(input, out var number);**

### Convert

* **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)**

Многопоточность

## Введение в многопоточность.

## Класс Thread

**namespace System.Threading**

### Свойства класса Thread

* **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 // по умолчанию не установлено**
* **ManagedThreadId: возвращает числовой идентификатор текущего потока**
* **Priority: хранит приоритет потока - значение перечисления ThreadPriority:**
  + **Lowest**
  + **BelowNormal**
  + **Normal // по умолчанию потоку задается значение Normal**
  + **AboveNormal**
  + **Highest**
* **ThreadState: возвращает состояние потока - одно из значений перечисления:**
  + **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**
* **CurrentThread: позволяет получить текущий поток**

### Методы класса Thread

* **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**

## Создание потоков. Делегат ThreadStart

**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**

**using System.Threading;**

**// public delegate void ThreadStart(); // Action<>**

**void Print() => Console.WriteLine("Hello Threads");**

**// создание потока myThread**

**Thread myThread = new Thread(Print);**

**или:**

**Thread myThread = new Thread(new ThreadStart(Print));**

**или:**

**Thread myThread = new Thread(() => Console.WriteLine("Hello Threads"));**

**myThread.Start(); // запуск потока myThread**

## Потоки с параметрами и ParameterizedThreadStart

**using System.Threading;**

**record class Person(string Name, int Age);**

**// public delegate void ParameterizedThreadStart(object? obj);**

**// Action<object?>**

**void Print(object? obj)**

**{**

**if (obj is Person person)**

**Console.WriteLine($"Name = {person.Name}, Age = {person.Age}");**

**}**

**// создание потока myThread**

**Thread myThread = new Thread(Print);**

**или:**

**Thread myThread = new Thread(new ParameterizedThreadStart(Pring));**

**или:**

**Thread myThread = new Thread**

**(obj =>**

**{**

**if (obj is Person person)**

**Console.WriteLine($"Name = {person.Name}, Age = {person.Age}");**

**}**

**);**

**myThread.Start(new Person(Tom, 37)); // запуск потока myThread**

**Лучше метод делегат написать внутри объекта, чтобы не выполнять приведение типов:**

**record class Person(string Name, int Age)**

**{**

**void Print() => Console.WriteLine($"Name = {Name}, Age = {Age}");**

**}**

## Синхронизация потоков

**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.**

**int x = 0; // то, что будем изменять в потоках**

**object locker = new(); // объект-заглушка**

**void Print()**

**{**

**lock (locker)**

**{**

**... что-то делаем с x**

**}**

**}**

**Thread myThread1 = new(Print);**

**Thread myThread2 = new(Print);**

## Мониторы

**class System.Threading.Monitor**

### 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.**

### Пример

**int x = 0; // то, что будем изменять в потоках**

**object locker = new(); // объект-заглушка**

**void Print()**

**{**

**// пока мы не заблокировали locker, поэтому false**

**bool acquiredLock = false;**

**try**

**{**

**// пытаемся получить locker, чтобы заблокировать его**

**Monitor.Enter(locker, ref acquiredLock);**

**... что-то делаем с x**

**}**

**finally**

**{**

**// освобождаем locker**

**if (acquiredLock) Monitor.Exit(locker);**

**}**

**}**

**Thread myThread1 = new(Print);**

**Thread myThread2 = new(Print);**

## Класс AutoResetEvent

### Методы

* **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.**
* **WaitAll(WaitHandle w) и WaitAny(WaitHandle w), где WaitHandle - базовый класс для AutoResetEvent: используются для отслеживания состояния объектов AutoResetEvent.**

### Пример

**int x = 0; // то, что будем изменять в потоках**

**AutoResetEvent waitHandler = new AutoResetEvent(true); // объект-событие**

**// true указывает на то, что создаваемый объект изначалльно будет в //сигнальном состоянии**

**void Print()**

**{**

**waitHandler.WaitOne(); // ожидаем сигнала**

**// или AutoResetEvent.WaitAll(new WaitHandle[] {waitHandler});**

**... что-то делаем с x**

**waitHandler.Set(); // сигнализируем, что waitHandler в сигнальном //состоянии**

**}**

**Thread myThread1 = new(Print);**

**Thread myThread2 = new(Print);**

## Мьютексы

**class System.Threading.Mutex**

**int x = 0; // то, что будем изменять в потоках**

**Mutex mutex = new();// создаем объект мьютекса**

**void Print()**

**{**

**mutex.WaitOne(); // приостанавливаем объект до получения мьютекса**

**... что-то делаем с x**

**mutex.ReleaseMutex(); // освобождаем мьютекс**

**}**

**Thread myThread1 = new(Print);**

**Thread myThread2 = new(Print);**

## Семафоры

**class System.Threading.Semaphore**

### Конструкторы

* **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**

### Методы

* **WaitOne(): ожидает получения свободного места в семафоре**
* **Release(): освобождает место в семафоре**

### Пример

**Semaphore semaphore = new Semaphore(3, 3); // максимум 3 потока работают //одновременно**

**void Print()**

**{**

**semaphore.WaitOne(); // ожидаем, когда освободится место**

**... что-то делаем в потоке**

**semaphore.Release(); // освобождаем место**

**}**

**for (int i = 0; i < 10; i++) // запускаем 10 потоков**

**{**

**Thread myThread = new Thread(Read);**

**myThread.Start();**

**}**

**// работать одновременно благодаря семафору в методе Read будут только 3 //потока**

Параллельное программирование и библиотека TPL (Task Parallel Library)

## Задачи и класс Task

**using System.Threading.Tasks;**

**Задача (Task) - это не поток! Но ее можно запускать асинхронно в пуле потоков.**

**//Task(Action)**

**Task task = new Task(() => Console.WriteLine("Hello Task!"));**

**task.Start(); // запускает задачу**

**// Task.Factory.StartNew(Action) - возвращается запущенная задача**

**Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello Task!"));**

**// Task.Run(Action) - возвращается запущенная задача**

**Task task = Task.Run(() => Console.WriteLine("Hello Task!"));**

### Ожидание завершения задачи

**Метод Wait() блокирует вызывающий поток, в котором запущена задача, пока эта задача не завершит свое выполнение.**

**Task task = new Task(() => ...);**

**task.Start();**

**task.Wait(); // ожидаем завершения задачи task**

### Синхронный запуск задачи

**По умолчанию задачи запускаются асинхронно. Однако с помощью метода RunSynchronously() можно запускать синхронно:**

**Task task = new Task(() => ...);**

**task.RunSynchronously(); // запускаем задачу синхронно с основым потоком**

### Свойства класса Task

* **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**

## Работа с классом Task

### Вложенные задачи

**var outher = Task.Factory.StartNew(() =>**

**{**

**var inner = Task.Factory.StartNew(() => ...);**

**inner.Wait();**

**});**

**outher.Wait();**

**или:**

**var outher = Task.Factory.StartNew(() =>**

**{**

**var inner = Task.Factory.StartNew(() => ...);**

**}, TaskCreationOptions.AttachedToParent);**

**// внещняя задача завершится тогда,**

**// когда завершатся все прикрепленные к ней задачи**

**outher.Wait();**

### Массив задач

**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):**

**Task[] tasks = new Task[3];**

**for(var i = 0; i < tasks.Length; i++)**

**tasks[i] = Task.Run(() => ...);**

**Task.WaitAll(tasks); // ожидаем завершения всех задач**

**Метод Task.WaitAny(tasks) ждет, пока завершится хотя бы одна из массива задач.**

### Возвращение результатов из задач

**Свойство задачи Result хранит в себе результат выполнения задачи, приведенный к нужному типу. При обращении к свойству Result текущий поток останавливает выполнение и ждет результата.**

**record class Person(string Name, int Age);**

**Task<Person> personTask = new Task<Person>(() => new Person("Tom", 37));**

**personTask.Start();**

**Person person = personTask.Result();// ожидаем результат задачи**

## Задачи продолжения **(**continuation task**)**

**Task.ContinueWith(Action<Task>)**

**Task task1 = new Task(() => ...);**

**// задача продолжения - task2 - выполняется после завершения task1**

**Task task2 = task1.ContinueWith(task => ...); // где task - task1**

**// запускаем первую задачу**

**task1.Start();**

**// ждем окончания второй задачи**

**task2.Wait();**

**или можно передавать результат выполнения первой задачи во вторую, второй - в третью и т.д.:**

**Task<int> task1 = new Task<int>(() => 1);**

**Task<int> task2 = task1.ContinueWith(task => task.Result + 1); // task = task1**

**Task<int> task3 = task2.ContinueWith(task => task.Result + 1); // task = task2**

**task1.Start(); // запускаем первую задачу**

**var result = task3.Result; // ждем результат от последней задачи**

**Console.WriteLine(result); // 3**

## Класс Parallel

### Методы

* **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.**

**Parallel.Invoke(**

**() => ...,**

**() => ...,**

**...**

**); // запускается параллельное выполнение нескольких задач**

* **Parallel.For позволяет выполнять итерации цикла параллельно:**

**For(int a, int b, Action<int> action)**

**a - начальный индекс элемента в цикле (включительно),**

**b - конечный индекс (исключается),**

**action - метод, который будет выполняться раз за итерацию,**

**и принимает в качестве параметра этот индекс (от a до b)**

**ParallelLoopResult result = Parallel.For(**

**2,**

**5,**

**i => Console.WriteLine(i)**

**); // запускаем**

**// 3**

**// 4**

**// 2**

**// задачи выполняются параллельно**

* **Parallel.ForEach осуществляет итерацию по коллекции, реализующей интерфейс IEnumerable.**

**ParallelLoopResult ForEach<TSource>(IEnumerable<TSource> source, Action<TSource> body)**

**ParallelLoopResult result = Parallel.ForEach<int>(**

**new List<int> { 1, 2, 3 },**

**n => Console.WriteLine(n)**

**); // запускаем**

**// 2**

**// 1**

**// 3**

**// задачи выполняются параллельно**

### Выход из цикла

**ParallelLoopResult result = Parallel.For(1, 10, MyMethod);**

**void MyMethod(int n, ParallelLoopState p)**

**{**

**if (n == 5) p.Break(); // если передано 5, выходим из цикла**

**...**

**}**

**то же самое для Parallel.ForEach**

### Свойства ParallelLoopResult

* **IsCompleted: определяет, завершилось ли полное выполнение параллельного цикла**
* **LowestBreakIteration: возвращает индекс, на котором произошло прерывание работы цикла**

## Отмена задач и параллельных операций**.** CancellationToken

**struct CancellationToken из System.Threading**

### Общий алгоритм отмены задачи

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.**

**CancellationTokenSource cancelTokenSource = new CancellationTokenSource();**

**CancellationToken token = cancelTokenSource.Token;**

**Для передачи токена в задачу можно применять один из конструкторов класса Task:**

**Task task = new Task(() => { ... }, token);**

1. **Определяем в задаче действия на случай ее отмены. 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.**
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).**

### Мягкий выход из задачи без исключения OperationCanceledException

**CancellationTokenSource tokenSource = new CancellationTokenSource();**

**CancellationToken token = tokenSource.Token;**

**Task task = new Task(() =>**

**{**

**// проверяем наличие сигнала отмены задачи**

**if (token.IsCansellationRequested)**

**return;**

**...**

**}, token); // добавляем token вторым аргументом в конструктор Task**

**task.Start(); // запускаем задачу**

**if (...) tokenSource.Cancel(); // отменяем выполнение задачи**

**tokenSource.Dispose(); // освобождаем ресурсы**

**var status = task.Status; // RunToCompletion**

### Отмена задачи с помощью генерации исключения

**CancellationTokenSource tokenSource = new CancellationTokenSource();**

**CancellationToken token = tokenSource.Token;**

**Task task = new Task(() =>**

**{**

**// проверяем наличие сигнала отмены задачи**

**if (token.IsCansellationRequested)**

**token.ThrowIfCancellationRequested(); // генерируем исключение**

**...**

**}, token); // добавляем token вторым аргументом в конструктор Task**

**try**

**{**

**task.Start(); // запускаем задачу**

**if (...) tokenSource.Cancel(); // отменяем выполнение задачи**

**task.Wait(); // ожидаем завершения задачи, иначе не будет выброшено исключение**

**}**

**catch (AggregateException ae) // набор исключений**

**{**

**foreach (Exception e in ae.InnerExceptions)**

**{**

**if (e is TaskCanceledException)**

**... // нужное нам исключение в наборе**

**}**

**}**

**finally**

**{**

**tokenSource.Dispose(); // освобождаем ресурсы**

**}**

**var status = task.Status; // Canceled**

### Регистрация обработчика отмены задачи

**Метод Register() позволяет зарегистрировать обработчик отмены задачи в виде делегата Action:**

**CancellationTokenSource tokenSource = new CancellationTokenSource();**

**CancellationToken token = tokenSource.Token;**

**Task task = new Task(() =>**

**{**

**int i = 0;**

**// что происходит в случае вызова Cancel:**

**token.Register(() => i = -1;);**

**// цикл прервется, когда будет вызван метод Cancel и i станет равна -1**

**while(i != -1) { ... }**

**}, token); // добавляем token вторым аргументом в конструктор Task**

**task.Start(); // запускаем задачу**

**if (...) tokenSource.Cancel(); // отменяем выполнение задачи**

**tokenSource.Dispose(); // освобождаем ресурсы**

**var status = task.Status; // RunToCompletion**

### Передача токена во внешний метод

**void OutherMethod(CacellationToken token)**

**{**

**// проверяем наличие сигнала отмены задачи**

**if (token.IsCansellationRequested)**

**token.ThrowIfCancellationRequested(); // генерируем исключение**

**...**

**}**

**CancellationTokenSource tokenSource = new CancellationTokenSource();**

**CancellationToken token = tokenSource.Token;**

**Task task = new Task(() => OutherMethod(token), token);**

**try**

**{**

**task.Start(); // запускаем задачу**

**if (...) tokenSource.Cancel(); // отменяем выполнение задачи**

**task.Wait(); // ожидаем завершения задачи, иначе не будет выброшено исключение**

**}**

**catch (AggregateException ae) // набор исключений**

**{**

**foreach (Exception e in ae.InnerExceptions)**

**{**

**if (e is TaskCanceledException)**

**... // нужное нам исключение в наборе**

**}**

**}**

**finally**

**{**

**tokenSource.Dispose(); // освобождаем ресурсы**

**}**

**var status = task.Status; // Canceled**

### Отмена параллельных операций Parallel

**CancellationTokenSource tokenSource = new CancellationTokenSource();**

**CancellationToken token = tokenSource.Token;**

**// в другой задаче посылаем сигнал отмены**

**new Task(() =>**

**{**

**Thread.Sleep(500);**

**tokenSource.Cancel();**

**}).Start();**

**try**

**{**

**Parallel.ForEach<int>(**

**new List<int> { 1, 2, 3 },**

**new ParallelOptions { CancellationToken = token },**

**() => { ... }**

**);**

**// или так:**

**// Parallel.For(**

**// 1,**

**// 3,**

**// new ParallelOptions { CancellationToken = token },**

**// ) => { ... }**

**// );**

**}**

**catch (OperationCanceledException) { ... }**

**finally**

**{**

**tokenSource.Dispose();**

**}**

**В результате программа выбрасывает исключение OperationCanceledException, и выполнение параллельных операций прекращается.**

Aсинхронное программирование

## Асинхронные методы**,** **async** и **await**

**Асинхронный метод обладает следующими признаками:**

* **В заголовке метода используется модификатор async, который указывает, что метод содержит выражения await**
* **Метод содержит одно или несколько выражений await, которое можно использовать только в async-методе**
* **В качестве возвращаемого типа используется один из следующих:**
  + **void**
  + **Task**
  + **Task<T>**
  + **ValueTask<T>**
* **Может использовать любое количество параметров, но не может определять параметры с модификаторами out, ref и in**

### Пример

**PrintAsync(); // вызываем сразу async-метод без выражения await, чтобы не тормозить Main**

**"Start Main".Print(); // Main продолжает выполняться параллельно с Task внутри PrintAsync**

**Thread.Sleep(4000); // ждем 4 сек, чтобы Main не завершил работу быстрее PrintAsync**

**"End Main".Print();**

**// асинхронный метод**

**// в данном случае метод внутри не возвращает Task, но имеет внутри выражение await,**

**// поэтому в качестве возвращаемого типа можно поставить Task**

**async static Task PrintAsync()**

**{**

**"Start PrintAsync".Print(); // выполняется синхронно**

**await Task.Run(Print); // await - ждем завершения выполнения задачи**

**// Task выполняется в другом потоке асинхронно,**

**// тем временем метод Main продолжает свою работу**

**"End PrintAsync".Print(); // из-за await ждет окончания Task,**

**// а Main в это время продолжает свою работу**

**}**

**// метод будет задерживать на 3 сек Task в async-методе PrintAsync**

**static void Print()**

**{**

**"Start Print".Print();**

**Thread.Sleep(3000);**

**"End Print".Print();**

**}**

**// класс-расширение для String, чтобы сразу печатать сообщение**

**static class StringExtension**

**{**

**public static void Print(this string s) => Console.WriteLine(s);**

**}**

**// Output**

**/\***

**Start PrintAsync**

**Start Print**

**Start Main**

**End Print**

**EndPrintAsync**

**End Main**

**\*/**

### Асинхронный метод Main

**async static Task Main(string[] args)**

### Задержка асинхронной операции и Task.Delay

**Task.Delay(int ms) / Task.Delay(TimeSpan ts) - метод, который можно применять в асинхронных методах для остановки метода**

**await PrintAsync(); // вызов асинхронного метода**

**// определение асинхронного метода**

**async Task PrintAsync()**

**{**

**await Task.Delay(3000); // Delay - асинхронная операция, поэтому нужен await**

**// await Task.Delay(TimeSpan.FromMilliseconds(3000));**

**...**

**}**

### Преимущества асинхронности

**var tomTask = PrintNameAsync("Tom"); // задача запускается в этот момент**

**var bobTask = PrintNameAsync("Bob"); // задача запускается в этот момент**

**var samTask = PrintNameAsync("Sam"); // задача запускается в этот момент**

**// по сути, все задачи запускаются одновременно и закончатся одновременно**

**await tomTask; // в случае, если задача еще не выполнена,**

**// поток ждет ее завершения**

**await bobTask; // после завершения первой задачи,**

**// если еще не закончилась вторая, ждем ее завершения**

**await samTask; // если вдруг третья задача еще не закончилась,**

**// ждем ее завершения**

**// определение асинхронного метода**

**async Task PrintNameAsync(string name)**

**{**

**await Task.Delay(3000); // имитация продолжительной работы**

**Console.WriteLine(name);**

**}**

**// Output**

**/\***

**Sam**

**Tom**

**Bob // по-разному**

**\*/**

### Определение асинхронного лямбда-выражения

**// асинхронное лямбда-выражение**

**Func<string, Task> printer = async (message) =>**

**{**

**await Task.Run(() => ...);**

**...**

**}**

**// вызов асинхронного метода**

**await printer("Hello World");**

## Возвращение результата из асинхронного метода

### void

**void-методов следует избегать:**

* **к подобным методам нельзя применить метод await**
* **в таких методах сложно обрабатывать исключения**
* **их сложно тестировать**

**async void Method(int n)**

**{**

**await Task.Run(() => ... );**

**}**

**Method(10);**

### Task

**async Task Method(int n)**

**{**

**await Task.Run(() => ... );**

**}**

**Task task = Method(10); // задача начинает выполняться**

**...**

**await task; // ожидаем завершения задачи**

### Task<T>

**async Task<int> Method(int n)**

**{**

**await Task.Run(() => ... );**

**return n \* n;**

**}**

**Task<int> task = Method(10); // задача начинает выполняться**

**...**

**int n = await task; // ожидаем завершения задачи, получаем результат**

### ValueTask<T>

**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.**

**async ValueTask<int> Method(int n)**

**{**

**await Task.Run(() => ... );**

**return n \* n;**

**}**

**ValueTask<int> valueTask = Method(10); // задача начинает выполняться**

**...**

**int n = await valueTask; // ожидаем завершения задачи, получаем результат**

**// или**

**// int n = await valueTask.AsTask();**

## Последовательное и параллельное выполнение**.** Task**.**WhenAll и Task**.**WhenAny

### Последовательное выполнение:

**// запускаем и ждем выполнения задачи 1**

**await AsyncMethod1();**

**// запускаем и ждем выполнения задачи 2 после завершения задачи 1**

**await AsyncMethod2();**

**// запускаем и ждем выполнения задачи 3 после завершения задачи 2**

**await AsyncMethod3();**

### Параллельное выполнение:

**// определяем и запускаем задачи**

**Task task1 = AsyncMethod1();**

**Task task2 = AsyncMethod2();**

**Task task3 = AsyncMethod3();**

**// ожидаем завершения задач**

**await task1;**

**await task2;**

**await task3;**

### Task.WhenAll

**Task.WhenAll - аналог метода Task.WaitAll, но для асинхронных методов и позволяет применять оператор await. Task.WhenAll создает новую задачу, которая будет автоматически выполнена после выполнения всех предоставленных задач.**

**// определяем и запускаем задачи**

**Task task1 = AsyncMethod1();**

**Task task2 = AsyncMethod2();**

**Task task3 = AsyncMethod3();**

**// ожидаем завершения всех задач**

**await Task.WhenAll(task1, task2, task3);**

### Task.WhenAny

**Task.WhenAny - аналог метода Task.WaitAny, но для асинхронных методов и позволяет применять оператор await. Task.WaitAny завершает выполнение, когда завершается хотя бы одна задача.**

**// определяем и запускаем задачи**

**Task task1 = AsyncMethod1();**

**Task task2 = AsyncMethod2();**

**Task task3 = AsyncMethod3();**

**// ожидаем завершения хотя бы одной задачи**

**await Task.WhenAny(task1, task2, task3);**

### Получение результата

**// определяем и запускаем задачи**

**Task<int> task1 = AsyncMethod1();**

**Task<int> task2 = AsyncMethod2();**

**Task<int> task3 = AsyncMethod3();**

**// ожидаем завершения всех задач и получаем результаты**

**int[] results = await Task.WhenAll(task1, task2, task3);**

**// или можно отдельно получить результат из задачи**

**int task2Result = task2.Result;**

## Обработка ошибок в асинхронных методах

**async Task AsyncMethod(int n)**

**{**

**if (n < 0)**

**throw new ArgumentException();**

**else**

**Console.WriteLine(n);**

**}**

**try**

**{**

**await AsyncMethod(-10); // будет выброшено исключение,**

**// но программа не остановит свою работу аварийно,**

**// а обработает исключение и продолжит работу**

**await AsyncMethod(10); // будет выполнено**

**}**

**catch (Exception e)**

**{**

**...**

**}**

**Если асинхронный метод имеет тип void, то в этом случае исключение во вне не передается!**

### Исследование исключения

**Свойства ошибок у объекта Task:**

* **IsFaulted - true, если в задаче произошла ошибка**
* **Exception - содержит всю информацию об ошибке**

**async Task AsyncMethod()**

**{**

**...**

**throw new ArgumentException("ArgumentException in the Task");**

**}**

**Task task = AsyncMethod();**

**try**

**{**

**await task;**

**}**

**catch**

**{**

**Console.WriteLine(task.Exception?.InnerException?.Message);**

**// ArgumentException in the Task**

**Console.WriteLine(task.IsFaulted);**

**// true**

**Console.WriteLine(task.Status);**

**// Faulted**

**}**

### Обработка нескольких исключений. WhenAll

**async Task AsyncMethod()**

**{**

**...**

**throw new ArgumentException("ArgumentException in the Task");**

**}**

**Task task1 = AsyncMethod();**

**Task task2 = AsyncMethod();**

**Task tasks = Task.WhenAll(task1, task2);**

**try**

**{**

**await tasks;**

**}**

**catch (Exception e)**

**{**

**Console.WriteLine(e.Message);**

**// ArgumentException in the Task**

**Console.WriteLine(tasks.IsFaulted);**

**// true**

**if (tasks.Exception is not null)**

**{**

**foreach (Exception ex in tasks.Exception.InnerExceptions)**

**{**

**Console.WriteLine(ex.Message); // ArgumentException in the Task**

**}**

**}**

**}**

## Асинхронные стримы с C# 8.0

**Асинхронный стрим представляет метод, который обладает тремя характеристиками:**

1. **метод имеет модификатор async**
2. **метод возвращает объект IAsyncEnumerable<T>. Интерфейс IAsyncEnumerable определяет метод GetAsyncEnumerator, который возвращает IAsyncEnumerator:**

**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();**

**}**

1. **метод содержит выражения yield return для последовательного получения элементов из асинхронного стрима**

### Пример

**// определение асинхронного метода - асинхронного стрима**

**async static IAsyncEnumerable<int> AsyncMethod()**

**{**

**for (int i = 0; i < 10; i++)**

**{**

**await Task.Delay(new Random().Next(1000));**

**yield return i;**

**}**

**}**

**await foreach (int i in AsyncMethod())**

**{**

**Console.Write(i);**

**}**

**// 123456789 - выводятся последовательно с разной задержкой**

LINQ

## Основы LINQ

**using System.LINQ;**

**Существует несколько разновидностей LINQ:**

* **LINQ to Objects: применяется для работы с массивами и коллекциями**
* **LINQ to Entities: используется при обращении к базам данных через технологию Entity Framework**
* **LINQ to XML: применяется при работе с файлами XML**
* **LINQ to DataSet: применяется при работе с объектом DataSet**
* **Parallel LINQ (PLINQ): используется для выполнения параллельных запросов**

### Операторы запросов LINQ

**from p in list**

**// проходит по всем элементам списка list, определяет каждый элемент как p**

**select p;**

**// передает выбранные значения в результирующую выборку**

### Методы расширения LINQ

* **Select: определяет проекцию выбранных значений**
* **Where: определяет фильтр выборки**
* **OrderBy: упорядочивает элементы по возрастанию**
* **OrderByDescending: упорядочивает элементы по убыванию**
* **ThenBy: задает дополнительные критерии для упорядочивания элементов возрастанию**
* **ThenByDescending: задает дополнительные критерии для упорядочивания элементов по убыванию**
* **Join: соединяет две коллекции по определенному признаку**
* **Aggregate: применяет к элементам последовательности агрегатную функцию, которая сводит их к одному объекту**
* **GroupBy: группирует элементы по ключу**
* **ToLookup: группирует элементы по ключу, при этом все элементы добавляются в словарь**
* **GroupJoin: выполняет одновременно соединение коллекций и группировку элементов по ключу**
* **Reverse: располагает элементы в обратном порядке**
* **All: определяет, все ли элементы коллекции удовлятворяют определенному условию**
* **Any: определяет, удовлетворяет хотя бы один элемент коллекции определенному условию**
* **Contains: определяет, содержит ли коллекция определенный элемент**
* **Distinct: удаляет дублирующиеся элементы из коллекции**
* **Except: возвращает разность двух коллекцию, то есть те элементы, которые создаются только в одной коллекции**
* **Union: объединяет две однородные коллекции**
* **Intersect: возвращает пересечение двух коллекций, то есть те элементы, которые встречаются в обоих коллекциях**
* **Count: подсчитывает количество элементов коллекции, которые удовлетворяют определенному условию**
* **Sum: подсчитывает сумму числовых значений в коллекции**
* **Average: подсчитывает cреднее значение числовых значений в коллекции**
* **Min: находит минимальное значение**
* **Max: находит максимальное значение**
* **Take: выбирает определенное количество элементов**
* **Skip: пропускает определенное количество элементов**
* **TakeWhile: возвращает цепочку элементов последовательности, до тех пор, пока условие истинно**
* **SkipWhile: пропускает элементы в последовательности, пока они удовлетворяют заданному условию, и затем возвращает оставшиеся элементы**
* **Concat: объединяет две коллекции**
* **Zip: объединяет две коллекции в соответствии с определенным условием**
* **First: выбирает первый элемент коллекции**
* **FirstOrDefault: выбирает первый элемент коллекции или возвращает значение по умолчанию**
* **Single: выбирает единственный элемент коллекции, если коллекция содержит больше или меньше одного элемента, то генерируется исключение**
* **SingleOrDefault: выбирает единственный элемент коллекции. Если коллекция пуста, возвращает значение по умолчанию. Если в коллекции больше одного элемента, генерирует исключение**
* **ElementAt: выбирает элемент последовательности по определенному индексу**
* **ElementAtOrDefault: выбирает элемент коллекции по определенному индексу или возвращает значение по умолчанию, если индекс вне допустимого диапазона**
* **Last: выбирает последний элемент коллекции**
* **LastOrDefault: выбирает последний элемент коллекции или возвращает значение по умолчанию**

**Можно объединять операторы LINQ и методы расширения LINQ, например:**

**int n = (from p in people where p.StartsWith("T") select p).Count();**

## Проекция данных (Select, let, SelectMany)

**Select(Func<TSource, TResult> selector)**

**Пример:**

**record class Person(string Name, int Age);**

**List<Person> people = [...];**

**IEnumerable<string> names = from person in people select person.Name;**

**// или**

**IEnumerable<string> names = people.Select(person => person.Name);**

### Переменые в запросах и оператор let

**В запросах LINQ (но не в методах расширения LINQ) можно создавать промежуточные переменные с помощью оператора let. Например:**

**var newPeople =**

**from person in people**

**let newName = $"Mr. {person.Name}" // промежуточная переменная**

**let dateOfBirth = DateTime.Now.Year - person.Age // еще одна**

**select new // на выходе создаем объект анонимного типа**

**{**

**MrName = newName, // придумываем свойства объекта**

**Year = dateOfBirth // и используем промежуточные переменные**

**};**

### Выборка из нескольких источников

**record class Course(string Title); // учебный курс**

**record class Student(string Name); // студент**

**List<Course> courses = [...];**

**List<Student> students = [...];**

**var result =**

**from course in courses // выбираем по одному курсу**

**from student in students // выбираем по одному студенту**

**select new // на выходе создаем объект анонимного типа**

**{**

**Student = student.Name, // придумываем свойства объекта**

**Course = course.Title // и используем элементы из обоих коллекций**

**};**

### SelectMany и сведение объектов

**Метод SelectMany позволяет свести набор коллекций в одну коллекцию.**

* **SelectMany(Func<TSource, IEnumerable<TResult>> selector);**

**record class Company(string Name, List<Person> Staff);**

**List<Company> companies = [...];**

**IEnumerable<Person> employees = companies**

**.SelectMany(company => company.Staff);**

**// или**

**IEnumerable<Person> emploees =**

**from company in companies**

**from employee in company.Staff**

**select employee**

* **SelectMany(Func<TSource, IEnumerable<TCollection>> collectionSelector,**

**Func<TSource,TCollection,TResult> resultSelector);**

**var employees =**

**companies**

**.SelectMany(**

**company => company.Staff, // TSourse -> TCollection**

**(company, employee) => new // (TSourse, TCollection) -> TResult**

**{**

**Name = employee.Name,**

**Company = company.Name**

**});**

**// или**

**var emploees =**

**from company in companies**

**from employee in company.Staff**

**select new**

**{**

**Name = emploee.Name,**

**Company = company.Name**

**};**

## Фильтрация коллекции

**Where<TSource> (Func<TSource,bool> predicate)**

**int[] numbers = { ... };**

**IEnumerable<int> evens = numbers.Where(i => i % 2 == 0 && i > 10);**

**// или**

**IEnumerable<int> evens = from i in numbers**

**where i => i % 2 == 0 && i > 10**

**select i;**

### Сложные фильтры

**record class Person(string Name, int Age, List<string> Languages);**

**List<Person> people = [...];**

**IEnumerable<Person> selectedPeople = people**

**.SelectMany(person => person.Languages,**

**(person, language) => new**

**{**

**Person = person,**

**Language = language**

**})**

**.Where(abstractType => abstractType.Age > 18**

**&& abstractType.Language == "English")**

**.Select(abstractType => abstractType.Person);**

**// или**

**IEnumerable<Person> selectedPeople =**

**from person in people**

**from language in person.Languages**

**where person.Age > 18**

**where language == "English"**

**select person;**

### Фильтрация по типу данных

**OfType() позволяет отфильтровать данные коллекции по определенному типу, например:**

**record class Person(string Name);**

**record class Student(string Name): Person(Name);**

**record class Employee(string Name): Person(Name);**

**List<Person> people =**

**[**

**new Student("Tom"),**

**new Person("Sam"),**

**new Student("Bob"),**

**new Employee("Mike")**

**];**

**List<Person> students = people.OfType<Student>();**

## Сортировка

### Оператор orderby и метод OrderBy

**OrderBy (Func<TSource,TKey> keySelector)**

**OrderBy (Func<TSource,TKey> keySelector, IComparer<TKey>? comparer)**

**record class Person(string Name, int Age);**

**Person[] people = [...];**

**IEnumerable<Person> orderedPeople = people.OrderBy(person => person.Age);**

**// можно использовать метод Sort массивов:**

**// Array.Sort(people, (p1, p2) => p1.Age.CompareTo(p2.Age));**

**// или**

**IEnumerable<Person> orderedPeople =**

**from person in people**

**orderby person.Age**

**select person;**

### Сортировка по убыванию

**IEnumerable<Person> orderedPeople =**

**people.OrderByDescending(person => person.Age);**

**// или**

**IEnumerable<Person> orderedPeople =**

**from person in people**

**orderby person.Age descending**

**select person;**

### Множественные критерии сортировки

**IEnumerable<Person> orderedPeople =**

**people**

**.OrderBy(person => person.Name)**

**.ThenBy(person => person.Age);**

**// ThenByDescending(person => person.Age);**

**// или**

**IEnumerable<Person> orderedPeople =**

**from person in people**

**orderby person.Name, person.Age**

**// orderby person.Name, person.Age descending**

**select person;**

### Переопределение критерия сортировки

**// Создание компаратора**

**class CustomComparer : ICompared<Person>**

**{**

**public int Compare(Person? p1, Person? p2)**

**=> p1.Age.CompareTo(p2.Age);**

**}**

**IEnumerable<Person> orderedPeople =**

**people**

**.OrderBy(person => person, new CustomComparer());**

**// или**

**IEnumerable<Person> orderedPeople =**

**people**

**.OrderBy(person => person,**

**Comparer<Person>**

**.Create((p1, p2) => p1.Age.CompareTo(p2.Age)));**

## Объединение, пересечение и разность коллекций

### Разность последовательностей (Except)

**int[] array1 = { 1, 2, 3 };**

**int[] array2 = { 3, 4, 5 };**

**var result = array1.Except(array2); // 1, 2**

**// из коллекции array1 убрали 3, т.к. 3 есть в обоих коллекциях**

### Пересечение последовательностей (Intersect)

**int[] array1 = { 1, 2, 3 };**

**int[] array2 = { 3, 4, 5 };**

**var result = array1.Intersect(array2); // 3**

**// в результат попало только 3, т.к. 3 есть в обоих коллекциях**

### Удаление дубликатов

**int[] array = { 1, 2, 2, 3 };**

**var result = array.Distinct(); // 1, 2, 3**

### Объединение последовательностей (Union, Concat)

**int[] array1 = { 1, 2, 3 };**

**int[] array2 = { 3, 4, 5 };**

**var result = array1.Union(array2); // 1, 2, 3, 4, 5**

**// в результат попало объединение коллекций**

**// дублирующиеся элементы добавляются один раз**

**var result = array1.Concat(array2); // 1, 2, 3, 3, 4, 5**

**// в результат копируются все элементы из обоих коллекций**

### Работа со сложными объектами

**Чтобы работать с последовательностями, которые содержат объекты своих классов и структур, то необходимо определить для них методы Equals и GetHashCode.**

**class Person**

**{**

**...**

**public override bool Equals(object? obj) ...**

**public override int GetHashCode() ...**

**}**

**Person[] people = { ... };**

**var result = people.Distinct();**

## Агрегатные операции

### Метод Aggregate

**int[] array = { 1, 2, 3 };**

**var result = array.Aggregate((x, y) => x - y); // -4**

**// аналогично 1 - 2 - 3**

**// или**

**var result = array.Aggregate(5, (x, y) => x - y); // -1**

**// аналогично 5 - 1 - 2 - 3**

### Получение размера выборки (Count)

**int[] array = { 1, 2, 3, 4, 5 };**

**var result = array.Count(); // 5**

**// или**

**var result = array.Count(i => i % 2 != 0); // 3**

**// количество нечетных чисел**

### Получение суммы (Sum)

**int[] array = { 1, 2, 3, 4, 5 };**

**var result = array.Sum(); // 15**

**// или**

**var result = array.Sum(i => i \* i); // 55**

**// сумма квадратов чисел**

### Максимальное, минимальное и среднее значения (Max, Min, Average)

**int[] array = { 1, 2, 3, 4, 5 };**

**int min = array.Min(); // 1**

**int max = array.Max(); // 5**

**double average = array.Average(); // 3**

**// (1 + 2 + 3 + 4 + 5) / 5**

**// или**

**int min = array.Min(i => i \* i); // 1**

**int max = array.Max(i => i \* i); // 25**

**double average = array.Average(i => i \* i); // 11**

**// (1 \* 1 + 2 \* 2 + 3 \* 3 + 4 \* 4 + 5 \* 5) / 5**

## Методы Skip и Take

### Skip и SkipLast

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

**var result = numbers.Skip(2); // 3, 4, 5**

**var result = numbers.SkipLast(2); // 1, 2, 3**

### SkipWhile

**SkipWhile(Func<TSource, bool> predicate);**

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

**var result = numbers.SkipWhile(i => i < 3); // 3, 4, 5**

### Take и TakeLast

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

**var result = numbers.Take(2); // 1, 2**

**var result = numbers.TakeLast(2); // 4, 5**

### TakeWhile

**TakeWhile(Func<TSource, bool> predicate);**

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

**var result = numbers.TakeWhile(i => i < 3); // 1, 2**

## Группировка

### Оператор group by

**IGrouping<K, V>**

**record class Person(string Name, string Company);**

**Person[] people = { ... };**

**IEnumerable<IGrouping<string, Person>> companies =**

**from person in people**

**group person by person.Company;**

**foreach (IGrouping<string, Person> company in companies)**

**{**

**string companyName = company.Key;**

**foreach (Person person in company)**

**{**

**...**

**}**

**}**

**Если в выражении LINQ последним оператором, выполняющим операции над выборкой, является group, то оператор select не применяется.**

### GroupBy

**GroupBy<TSource,TKey> (Func<TSource,TKey> keySelector);**

**...**

**IEnumerable<IGrouping<string, Person>> companies =**

**people.GroupBy(person => person.Company);**

### Создание нового объекта при группировке

**...**

**var companies =**

**from person in people**

**group person by person.Company into g**

**select new { CompanyName = g.Key, NumberOfPeople = g.Count() };**

**// или**

**var companies = people**

**.GroupBy(person => person.Company)**

**.Select(g => new { CompanyName = g.Key, NumberOfPeople = g.Count() });**

### Вложенные запросы

**...**

**var companies =**

**from person in people**

**group person by person.Company into g**

**select new**

**{**

**CompanyName = g.Key,**

**NumberOfPeople = g.Count(),**

**Employees = from employee in g select employee // IEnumerable<Person>**

**};**

**// или**

**var companies = people**

**.GroupBy(person => person.Company)**

**.Select(g => new**

**{**

**CompanyName = g.Key,**

**NumberOfPeople = g.Count(),**

**Employees = from employee in g select employee // IEnumerable<Person>**

**});**

## Соединение коллекций

### Оператор join

**from объект1 in набор1**

**join объект2** **in набор2** **on объект2.свойство2 equals объект1.свойство1**

**Если объект1.свойство1 и объект2.свойство2 равны, то оба объекта попадают в финальный результат.**

**record class Person(string Name, string Company);**

**record class Company(string Title, string Language);**

**// объекты обоих классов имеют один общий критерий**

**// - название компании Person.Company и Company.Title**

**Person[] people = { ... };**

**Company[] companies = { ... };**

**var result =**

**from person in people**

**join company in companies**

**on person.Company** **equals company.Title**

**select new**

**{**

**Name = person.Name,**

**Company = company.Title,**

**Language = companiy.Language**

**};**

### Метод Join

**Join(IEnumerable<TInner> inner,**

**Func<TOuter,TKey> outerKeySelector,**

**Func<TInner,TKey> innerKeySelector,**

**Func<TOuter,TInner,TResult> resultSelector);**

**Метод Join() принимает четыре параметра:**

1. **второй список, который соединяем с текущим**
2. **делегат, который определяет свойство объекта из текущего списка, по которому идет соединение**
3. **делегат, который определяет свойство объекта из второго списка, по которому идет соединение**
4. **делегат, который определяет новый объект в результате соединения**

**...**

**var result = people.Join(**

**// вторая коллекция**

**companies,**

**// свойство-селектор объекта из первой коллекции**

**person => person.Company,**

**// свойство-селектор объекта из второй коллекции**

**company => company.Title,**

**// результат**

**(person, company) => new // результат**

**{**

**Name = person.Name,**

**Company = company.Title,**

**Language = companiy.Language**

**}); //**

### GroupJoin

**Метод GroupJoin() кроме соединения последовательностей также выполняет и группировку.**

**GroupJoin(IEnumerable<TInner> inner,**

**Func<TOuter,TKey> outerKeySelector,**

**Func<TInner,TKey> innerKeySelector,**

**Func<TOuter, IEnumerable<TInner>,TResult> resultSelector);**

**Метод GroupJoin() принимает четыре параметра:**

**второй список, который соединяем с текущим**

**делегат, который определяет свойство объекта из текущей коллекции, по которому идет соединение и по которому будет идти группировка**

**делегат, который определяет свойство объекта из второй коллекции, по которому идет соединение**

**делегат, который определяет новый объект в результате соединения. Этот делегат получает группу - объект текущей коллекции, по которому шла группировка, и набор объектов из второй коллекции, которые сооставляют группу**

**...**

**var result =**

**from company in companies**

**join person in people**

**on company.Title equals person.Company into g**

**// результат - название компании и список работников**

**select new**

**{**

**Title = company.Title,**

**Employees = g // IEnumerable<Person>**

**};**

**// или**

**var result = companies.GroupJoin(**

**// второй набор**

**people,**

**// свойство-селектор объекта из первой коллекции**

**company => company.Title,**

**// свойство-селектор объекта из второй коллекции**

**person => person.Company,**

**// результат - название компании и список работников**

**(company, employees) => new**

**{**

**Title = company.Title,**

**Employees = employees // IEnumerable<Person>**

**});**

### Метод Zip

**Результат метода Zip - коллекция кортежей, каждый из которых хранит пару соответсвенных элементов из обоих коллекций (первый из одной коллекции соответствует первому из второй и т.д.). Лишние элементы отбрасываются.**

**int[] ages = { ... };**

**string[] names = { ... };**

**var result = ages.Zip(names);**

**// IEnumerable<(int First, string Second)>**

**foreach ((int First, string Second> e in result)**

**{**

**// First - обращение к первому элементу**

**// Second - обращение ко второму элементу**

**}**

## Проверка наличия и получение элементов

### All

**Метод All() проверяет, соответствуют ли все элементы условию.**

**string[] people = { ... };**

**// проверяем, все ли элементы имеют длину в 3 символа**

**bool result = people.All(person => person.Length == 3);**

### Any

**Метод Any() проверяет, соответствуют ли хотя бы один элемент условию.**

**...**

**bool result = people.Any(person => person.Length == 3);**

### Contains

**Метод Contains() возвращает true, если коллекция содержит определенный элемент. Должен быть определен метод Equals.**

**// в record class уже определены Equals и GetHashCode**

**record class Person(string Name, int Age);**

**Person[] people = { ... };**

**Person person = new Person("Tom", 34);**

**bool result = people.Contains(person);**

**// Или можно использовать специальный компаратор IEqualityComparer:**

**// Создание компаратора**

**class CustomComparer: IEqualityComparer<Person>**

**{**

**public bool Equals(Person? x, Person? y)**

**=> x is null || y is null**

**? false**

**: x.Age == y.Age**

**&& x.Name == y.Name;**

**public int GetHashCode(Person? obj)**

**=> obj?.Name.GetHashCode() \* obj?.Age ?? -1;**

**}**

**bool result = people.Contains(person, new CustomComparer);**

**// или динамическое создание специального компаратора IEqualityComparer**

**var comparer = EqualityComparer<Person>**

**.Create(**

**(p1, p2) // Equals()**

**=> p1 is null || p2 is null**

**? false**

**: p1.Age == p2.Age && p1.Name == p2.Name,**

**(p) => p.Name.GetHashCode() \* p.Age); // GetHashCode()**

**bool result = people.Contains(person, comparer);**

### First

**Метод First() возвращает первый элемент последовательности.**

**string[] people = { ... };**

**string person = people.First();**

**// или можно передать условие в виде предиката**

**string person = people.First(person => person.Length == 4);**

### FirstOrDefault

**Метод FirstOrDefault() возвращает первый элемент последовательности или значение по умолчанию.**

**string[] people = { ... };**

**string person = people.FirstOrDefault();**

**// или можно передать условие в виде предиката**

**string person = people.FirstOrDefault(person => person.Length == 4);**

**// также можно определить значение по умолчанию передав его в качестве аргумента**

**string person = people.FirstOrDefault("Default");**

**// или**

**string person = people.FirstOrDefault(person => person.Length == 4, "Default");**

### Last

**Метод Last() возвращает последний элемент последовательности.**

**string[] people = { ... };**

**string person = people.Last();**

**// или можно передать условие в виде предиката**

**string person = people.Last(person => person.Length == 4);**

### LastOrDefault

**Метод LastOrDefault() возвращает последний элемент последовательности или значение по умолчанию.**

**string[] people = { ... };**

**string person = people.LastOrDefault();**

**// или можно передать условие в виде предиката**

**string person = people.LastOrDefault(person => person.Length == 4);**

**// также можно определить значение по умолчанию передав его в качестве аргумента**

**string person = people.LastOrDefault("Default");**

**// или**

**string person = people.LastOrDefault(person => person.Length == 4, "Default");**

## Отложенное и немедленное выполнение LINQ

### Отложенное (deferred) выполнение

**При отложенном выполнении LINQ-выражение выполняется только во время очередной итерации (например, в foreach) и возвращается обычно IEnumerable<T> или IOrderedEnumerable<T>**

* **AsEnumerable**
* **Cast**
* **Concat**
* **DefaultIfEmpty**
* **Distinct**
* **Except**
* **GroupBy**
* **GroupJoin**
* **Intersect**
* **Join**
* **OfType**
* **OrderBy**
* **OrderByDescending**
* **Range**
* **Repeat**
* **Reverse**
* **Select**
* **SelectMany**
* **Skip**
* **SkipWhile**
* **Take**
* **TakeWhile**
* **ThenBy**
* **ThenByDescending**
* **Union**
* **Where**

**LINQ-запрос разбивается на три этапа:**

1. **Получение источника данных**

**string[] people = { "Tom", "Sam", "Bob" };**

1. **Создание запроса**

**IEnumerable<string> result**

**= people.Where(name => name.Length == 3);**

**// переменная result - переменная запроса - не хранит данных;**

**// она хранит только набор команд для получения результатов**

1. **Выполнение запроса и получение его результатов**

**foreach (string name in result) ...**

**// здесь происходит получение результатов**

### Немедленное (immediate) выполнение

**При немедленном выполнении методы возвращают одно значение или данные типов Array, List и Dictionary.**

* **Aggregate**
* **All**
* **Any**
* **Average**
* **Contains**
* **Count**
* **ElementAt**
* **ElementAtOrDefault**
* **Empty**
* **First**
* **FirstOrDefault**
* **Last**
* **LastOrDefault**
* **LongCount**
* **Max**
* **Min**
* **SequenceEqual**
* **Single**
* **SingleOrDefault**
* **Sum**
* **ToArray**
* **ToDictionary**
* **ToList**
* **ToLookup**

**string[] people = { "Tom", "Sam", "Bob" };**

**int result = people.Where(name => name.Length == 3).Count();**

**// метод Count неявно выполняет запрос и возвращает результат**

## Делегаты в запросах LINQ

**Вместо лямбды в LINQ-методы можно передавать соответствующие методы.**

**int[] array = { ... };**

**static bool GetInt(int num) => num > 3;**

**var result = array.Where(GetInt);**

**// или**

**Func<int, bool> func = GetInt;**

**var result = array.Where(func);**

Parallel LINQ

## Введение в Parallel LINQ. Метод AsParallel

**using System.Linq;**

**class ParallelEnumerable**

**Смысл применения PLINQ имеется преимущественно на больших коллекциях или при сложных операциях, где действительно выгода от распараллеливания запросов может перекрыть возникающие при этом издержки.**

### Метод AsParallel

**int[] numbers = { ... };**

**var result =**

**from number in numbers.AsParallel()**

**select number \* number;**

**// или**

**var result = numbers.AsParallel().Select(number => number \* number);**

### Метод ForAll

**Метод ForAll() в качестве параметра принимает делегат Action. ForAll выводит данные в том же потоке, в котором они обрабатываются.**

**int[] numbers = { ... };**

**(from number in numbers.AsParallel()**

**select number \* number)**

**.ForAll(Console.WriteLine);**

**// или**

**numbers**

**.AsParallel()**

**.Select(number => number \* number)**

**.ForAll(Console.WriteLine);**

## Метод AsOrdered

**При выполнении параллельного запроса порядок данных в результирующей выборки может быть не предсказуем. OrderBy упорядочивает элементы в соответствии с результатом, а не исходной последовательностью. Метод AsOrdered упорядочивает элементы в соответствии с исходной последовательностью (это требует больших издержек).**

**int[] numbers = { ... };**

**IEnumerable<int> result =**

**from number in numbers.AsParallel().AsOrdered()**

**select number \* number;**

**// или**

**IEnumerable<int> result =**

**numbers**

**.AsParallel()**

**.AsOrdered()**

**.Select(number => number \* number);**

**Если в программе предстоят какие-нибудь манипуляции с полученным набором, однако упорядочивание больше не требуется, мы можем применить метод AsUnordered:**

**IEnumerable<int> intermediateResult =**

**from number in numbers.AsParallel().AsOrdered()**

**select number \* number;**

**IEnumerable<int> finalResult =**

**from number in intermediateResult.AsUnordered()**

**where number > 10**

**select number;**

## Обработка ошибок и отмена операции

**Если возникнет ошибка в одном из потоков, то система прерывает выполнение всех потоков. При генерации исключений все они агрегируются в одном исключении типа AggregateException.**

**object[] numbers = new object[] { 1, 2, 3, "4" };**

**var result =**

**from number in numbers.AsParallel()**

**let x = (int) number // выбросится исключение**

**select x \* x;**

**try**

**{**

**result.ForAll(number => Console.WriteLine(number));**

**}**

**catch (AggregateException ex)**

**{**

**foreach (var e in ex.InnerExceptions)**

**Console.WriteLine(e.Message);**

**}**

### Прерывание параллельной операции

**int Square(int number)**

**{**

**Thread.Sleep(1000);**

**return number \* number;**

**}**

**CancellationTokenSource tokenSource = new CancellationTokenSource();**

**// запускаем дополнительную задачу, в которой через 400 миллисек. прерываем операцию**

**new Task(() =>**

**{**

**Thread.Sleep(400);**

**tokenSource.Cancel();**

**}).Start();**

**try**

**{**

**int[] numbers = { ... };**

**var result =**

**from number in**

**numbers**

**.AsParallel()**

**.WithCancellation(tokenSource.Token)**

**select Square(n);**

**}**

**catch (OperationCanceledException) { ... }**

**catch (AggregateException ex) { ... }**

**finally**

**{**

**tokenSource.Dispose();**

**}**

Рефлексия

## Введение в рефлексию. Класс System.Type

**Рефлексия представляет собой процесс выявления типов во время выполнения приложения. Основной функционал рефлексии сосредоточен в пространстве имен System.Reflection. В нем мы можем выделить следующие основные классы:**

* **Assembly: класс, представляющий сборку и позволяющий манипулировать этой сборкой**
* **AssemblyName: класс, хранящий информацию о сборке**
* **MemberInfo: базовый абстрактный класс, определяющий общий функционал для классов EventInfo, FieldInfo, MethodInfo и PropertyInfo**
* **EventInfo: класс, хранящий информацию о событии**
* **FieldInfo: хранит информацию об определенном поле типа**
* **MethodInfo: хранит информацию об определенном методе**
* **PropertyInfo: хранит информацию о свойстве**
* **ConstructorInfo: класс, представляющий конструктор**
* **Module: класс, позволяющий получить доступ к определенному модулю внутри сборки**
* **ParameterInfo: класс, хранящий информацию о параметре метода**

**Класс System.Type представляет изучаемый тип. Некоторые его методы и свойства:**

* **Метод FindMembers() возвращает массив объектов MemberInfo данного типа**
* **Метод GetConstructors() возвращает все конструкторы данного типа в виде набора объектов ConstructorInfo**
* **Метод GetEvents() возвращает все события данного типа в виде массива объектов EventInfo**
* **Метод GetFields() возвращает все поля данного типа в виде массива объектов FieldInfo**
* **Метод GetInterfaces() получает все реализуемые данным типом интерфейсы в виде массива объектов Type**
* **Метод GetMembers() возвращает все члены типа в виде массива объектов MemberInfo**
* **Метод GetMethods() получает все методы типа в виде массива объектов MethodInfo**
* **Метод GetProperties() получает все свойства в виде массива объектов PropertyInfo**
* **Свойство Name возвращает имя типа**
* **Свойство Assembly возвращает название сборки, где определен тип**
* **Свойство Namespace возвращает название пространства имен, где определен тип**
* **Свойство IsArray возвращает true, если тип является массивом**
* **Свойство IsClass возвращает true, если тип представляет класс**
* **Свойство IsEnum возвращает true, если тип является перечислением**
* **Свойство IsInterface возвращает true, если тип представляет интерфейс**

### Получение типа

**record class Person(string Name);**

**Тип можно получить 3 способами:**

1. **с помощью оператора typeof**

**Type type =** **typeof(Person);**

1. **с помощью метода GetType() класса Object**

**Person person = new Person("Tom");**

**Type type = person.GetType();**

1. **с помощью статического метода Type.GetType()**

**Type? type = Type.GetType("PeopleNamespace.Person", false, true);**

**// параметр 1 - полное имя класса с пространством имен**

**// если тип в другой сборке, то через запятую после имени**

**// указывается имя сборки**

**// Type? type = Type.GetType("PeopleNamespace.Person, MyLybrary", false, true);**

**// параметр 2 - генерировать ли исключение, если класса нет**

**// параметр 3 - игнорировать ли регистр в параметре 1**

**type.Name - краткое имя типа - Person**

**type.FullName - полное имя типа - PeopleNamespace.Person**

**type.Namespace - пространство имен - PeopleNamespace**

**type.IsValueType - структура ли - False**

**type.IsClass - класс ли - True**

### Поиск реализованных интерфейсов

**interface IPerson { ... }**

**interface IEducation { ... }**

**class Student : IPerson, IEducation { ... }**

**Type type = typeof(Student);**

**Type[]? interfaces = type.GetInterfaces();**

## Применение рефлексии и исследование типов

**using System.Reflection;**

### Получение всех компонентов типа

**Метод GetMembers() возвращает все доступные компоненты типа в виде объекта MemberInfo. Свойства объекта MemberInfo:**

* **DeclaringType: возвращает полное название типа.**
* **MemberType: возвращает значение из перечисления MemberTypes:**
  + **MemberTypes.Constructor**
  + **MemberTypes.Method**
  + **MemberTypes.Field**
  + **MemberTypes.Event**
  + **MemberTypes.Property**
  + **MemberTypes.NestedType**
* **Name: возвращает название компонента**

**using System.Reflection;**

**public class Person**

**{**

**// приватное поле**

**private string name;**

**// открытое свойство**

**public int Age { get; set; }**

**// конструктор**

**public Person(string name, int age) ...**

**// метод**

**public void Print() => ...**

**}**

**Type type = typeof(Person);**

**foreach (MemberInfo member in type.GetMembers())**

**Console.WriteLine(**

**$"{member.DeclaringType} - {member.MemberType} - {member.Name}");**

**/\* Output:**

**Person - Method - get\_Age**

**Person - Method - set\_Age**

**Person - Method - Print**

**System.Object - Method - GetType**

**System.Object - Method - ToString**

**System.Object - Method - Equals**

**System.Object - Method - GetHashCode**

**Person - Constructor - .ctor**

**Person - Property - Age**

**\*/**

**Мы получаем:**

* **только публичные компоненты класса**
* **для свойства - геттер и сеттер**
* **функционал, унаследованный от базовых классов**

### BindingFlags

**Другая форма метода GetMembers - MembersInfo[] GetMembers(BindingFlags).**

**BindingFlags принимает значения:**

* **DeclaredOnly: получает только методы непосредственно данного класса, унаследованные методы не извлекаются**
* **Instance: получает только методы экземпляра**
* **NonPublic: извлекает не публичные методы**
* **Public: получает только публичные методы**
* **Static: получает только статические методы**

**Можно их комбинировать с помощью побитового ИЛИ ( | )**

**...**

**foreach (MemberInfo member in type.GetMembers(**

**BindingFlags.DeclaredOnly**

**| BindingFlags.Instance**

**| BindingFlags.NonPublic**

**| BindingFlags.Public))**

**Console.WriteLine(**

**$"{member.DeclaringType} - {member.MemberType} - {member.Name}");**

**/\* Output:**

**Person - Method - get\_Age**

**Person - Method - set\_Age**

**Person - Method - Print**

**Person - Constructor - .clor**

**Person - Property - Age**

**Person - Field - name**

**Person - Field - <Age>k\_BackingField**

**\*/**

### Получение одного компонента по имени

**Для получения одного компонента можно использовать метод MemberInfo[] GetMember(string componentsName) или MemberInfo[] GetMember(string componentsName, BindingFlags flag)**

**...**

**MemberInfo[] print = type.GetMember("Print", BindingFlags.Instance | BindingFlags.Public)**

**foreach (MemberInfo member in print)**

**Console.WriteLine($"{member.MemberType} - {member.Name}");**

**/\* Output:**

**Method - Print**

**\*/**

## Исследование методов и конструкторов с помощью рефлексии

### Получение информации о методах

**Метод MethodInfo[] GetMethods().**

**Свойства MethodInfo:**

* **IsAbstract: возвращает true, если метод абстрактный**
* **IsFamily: возвращает true, если метод имеет модификатор доступа protected**
* **IsFamilyAndAssembly: возвращает true, если метод имеет модификатор доступа private protected**
* **IsFamilyOrAssembly: возвращает true, если метод имеет модификатор доступа protected internal**
* **IsAssembly: возвращает true, если метод имеет модификатор доступа internal**
* **IsPrivate: возвращает true, если метод имеет модификатор доступа private**
* **IsPublic: возвращает true, если метод имеет модификатор доступа public**
* **IsConstructor: возвращает true, если метод предоставляет конструктор**
* **IsStatic: возвращает true, если метод статический**
* **IsVirtual: возвращает true, если метод виртуальный**
* **ReturnType: возвращает тип возвращаемого значения**

**Методы MethodInfo:**

* **GetMethodBody(): возвращает тело метода в виде объекта MethodBody**
* **GetParameters(): возвращает массив параметров, где каждый параметр представлен объектом типа ParameterInfo**
* **Invoke(): вызывает метод**

**class Person**

**{**

**public string Name { get; set; } = "Bob";**

**public void Print(string message, int times = 1)**

**{**

**while (times-- > 0) Console.WriteLine(message);**

**}**

**public string GetName() => Name;**

**}**

**Type type = typeof(Person);**

**foreach (MethodInfo method in type.GetMethods())**

**{**

**string modificator = method.IsStatic ? "static" : "";**

**modificator += method.IsVirtual ? "virtual" : "";**

**Console.WriteLine(**

**$"{modificator} {method.ReturnType.Name} {method.Name}()");**

**}**

**/\* Output:**

**String get\_Name()**

**Void set\_Name()**

**Void Print()**

**String GetName()**

**Type GetType()**

**virtual String ToString()**

**virtual Boolean Equals()**

**virtual Int32 GetHashCode()**

**\*/**

### BindingFlags

**class Person**

**{**

**public string Name { get; set; } = "Bob";**

**protected internal void Print(string message, int times = 1)**

**{**

**while (times-- > 0) Console.WriteLine(message);**

**}**

**private string GetName() => Name;**

**}**

**Type type = typeof(Person);**

**MethodInfo[] methods = type.GetMethods(**

**BindingFlags.DeclaredOnly**

**| BindingFlags.Instance**

**| BindingFlags.NonPublic**

**| BindingFlags.Public)**

**foreach (MethodInfo method in methods)**

**{**

**string modificator = method.IsStatic ? "static" : "";**

**modificator += method.IsVirtual ? "virtual" : "";**

**Console.WriteLine(**

**$"{modificator} {method.ReturnType.Name} {method.Name}()");**

**}**

**/\* Output:**

**String get\_Name()**

**Void set\_Name()**

**Void Print()**

**String GetName()**

**\*/**

### Исследование параметров

**С помощью метода GetParameters() можно получить все параметры метода в виде массива объектов ParameterInfo.**

**Свойства ParameterInfo:**

* **Attributes: возвращает атрибуты параметра**
* **DefaultValue: возвращает значение параметра по умолчанию**
* **HasDefaultValue: возвращает true, если параметр имеет значение по умолчанию**
* **IsIn: возвращает true, если параметр имеет модификатор in**
* **IsOptional: возвращает true, если параметр является необязательным**
* **IsOut: возвращает true, если параметр является выходным, то есть имеет модификатор out**
* **Name: возвращает имя параметра**
* **ParameterType: возвращает тип параметра**

**class Printer**

**{**

**public void PrintMessage(string message, int times = 1) ...**

**public void CreateMessage(out string message) ...**

**}**

**foreach (MethodInfo method in typeof(Printer).GetMethods())**

**{**

**// получаем все параметры**

**ParameterInfo[] parameters = method.GetParameters();**

**foreach (ParameterInfo parameter in parameters)**

**{**

**string modificator = parameter.IsIn**

**? "in"**

**: parameter.IsOut ? "out" : "";**

**string parameterType = parameter.ParameterType.Name;**

**string parameterName = parameter.Name;**

**string defaultValue = parameter.HasDefaultValue**

**? parameter.DefaultValue**

**: "";**

**}**

**}**

**Если параметр имеет модификатор ref, in, out, то в конце названия типа добавляется амперсанд - String&, например:**

**//Void CreateMessage(String& out message)**

### Вызов методов

**С помощью метода Invoke() можно вызвать метод:**

**public object? Invoke (object? obj, object?[]? parameters);**

**Первый параметр представляет объект, для которого вызывается метод. Второй объект представляет массив значений, которые передаются параметрам метода. И также метод может возвращать результат в виде значения object?.**

**...**

**// получаем метод Print**

**MethodInfo? print = typeof(Printer).GetMethod("Print");**

**// или получаем метод с флагами**

**MethodInfo? print = typeof(Printer).GetMethod("Print",**

**BindingFlags.Instance**

**| BindingFlags.Public**

**| BindingFlags.NonPublic);**

**// или получаем обобщенный(дженерик) метод**

**MethodInfo? print = typeof(Printer)**

**.GetMethod("PrintValue")?**

**.MakeGenericMethod(typeof(string)); // типизируем типом string**

**// вызываем метод Print без возвращения результата и без параметров**

**print?.Invoke(new Printer, parameters: null);**

**// или с возвращаением результата**

**var result = print?.Invoke(new Printer(), parameters: null);**

**// или добавим параметры (string, int)**

**var result = print?.Invoke(new Printer(), new object[] {"Hello", 3});**

### Получение конструкторов

**Для получения конструкторов применяется метод GetConstructors(), который возвращает массив объектов класса ConstructorInfo.**

**Свойства и методы:**

* **Свойство IsFamily: возвращает true, если конструктор имеет модификатор доступа protected**
* **Свойство IsFamilyAndAssembly: возвращает true, если конструктор имеет модификатор доступа private protected**
* **Свойство IsFamilyOrAssembly: возвращает true, если конструктор имеет модификатор доступа protected internal**
* **Свойство IsAssembly: возвращает true, если конструктор имеет модификатор доступа internal**
* **Свойство IsPrivate: возвращает true, если конструктор имеет модификатор доступа private**
* **Свойство IsPublic: возвращает true, если конструктор имеет модификатор доступа public**
* **Метод GetMethodBody(): возвращает тело конструктора в виде объекта MethodBody**
* **Метод GetParameters(): возвращает массив параметров, где каждый параметр представлен объектом типа ParameterInfo**
* **Метод Invoke(): вызывает конструктор**

**...**

**foreach (ConstructorInfo ctor in typeof(Person).GetConstructors(**

**BindingFlags.Instance**

**| BindingFlags.NonPublic**

**| BindingFlags.Public)) ...**

## Исследование полей и свойств с помощью рефлексии

### Получение информации о полях

**Для извлечения всех полей применяется метод GetFields(), который возвращает массив объектов класса FieldInfo.**

**Свойства и методы FieldInfo:**

* **Свойство IsFamily: возвращает true, если поле имеет модификатор доступа protected**
* **Свойство IsFamilyAndAssembly: возвращает true, если поле имеет модификатор доступа private protected**
* **Свойство IsFamilyOrAssembly: возвращает true, если поле имеет модификатор доступа protected internal**
* **Свойство IsAssembly: возвращает true, если поле имеет модификатор доступа internal**
* **Свойство IsPrivate: возвращает true, если поле имеет модификатор доступа private**
* **Свойство IsPublic: возвращает true, если поле имеет модификатор доступа public**
* **Свойство IsStatic: возвращает true, если поле статическое**
* **Метод GetValue(): возвращает значение поля**
* **Метод SetValue(): устанавливает значение поля**

**foreach (FieldInfo field in typeof(Person).GetFields(**

**BindingFlags.Instance**

**| BindingFlags.Static**

**| BindingFlags.NonPublic**

**| BindingFlags.Public)) ...**

### Получение и изменение значения поля

**GetField() возвращает одно поле:**

**var nameField = typeof(Person).getField("name");**

**// или с набором флагов**

**// например, можно получить приватное поле "name"**

**var nameField = typeof(Person)**

**.getField("name",**

**BindingFlags.Instance**

**| BindingFlags.NonPublic);**

**var person = new Person();**

**// получаем значение поля name у объекта person**

**var nameValue = name?.GetValue(person);**

**// изменяем значение поля name у объекта person**

**nameField?.SetValue(person, "Bob");**

### Свойства

**Для извлечения всех свойств типа применяется соответственно метод GetProperties(), который возвращает массив объектов PropertyInfo. Для получения одного свойства по имени применяется метод GetProperty(), в который передается название свойства и который возвращает объект PropertyInfo?.**

**Свойства и методы PropertyInfo:**

* **Свойство Attributes: возвращает коллекцию атрибутов свойства**
* **Свойство CanRead: возвращает true, если свойство доступно для чтения (есть get)**
* **Свойство CanWrite: возвращает true, если свойство доступно для записи (есть set)**
* **Свойство GetMethod: возвращает get-акссесор в виде объекта MethodInfo?**
* **Свойство SetMethod: возвращает set-акссесор в виде объекта MethodInfo?**
* **Свойство PropertyType: возвращает тип свойства**
* **Метод GetValue(): возвращает значение свойства**
* **Метод SetValue(): устанавливает значение свойства**

**...**

**foreach (PropertyInfo property in typeof(Person).GetProperties(**

**BindingFlags.Instance**

**| BindingFlags.Static**

**| BindingFlags.Public**

**| BindingFlags.NonPublic)) ...**

**...**

**var person = new Person();**

**// получаем свойство Age**

**PropertyInfo ageProperty = typeof(Person).GetProperty("Age");**

**// получаем значение свойства Age у объекта person**

**var ageValue = ageProperty?.GetValue(person);**

**// устанавливаем новое значение для свойства Age объекта person**

**ageProperty?SetValue(person, 33);**

## Динамическая загрузка сборок и позднее связывание

**Для управления сборками в пространстве имен System.Reflection имеется класс Assembly. Чтобы динамически загрузить сборку в приложение, надо использовать статические методы Assembly.LoadFrom() или Assembly.Load(). Метод LoadFrom() принимает в качестве параметра путь к сборке.**

**// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

***Проект 1 - C:\TestLibrary\bin\Debug\net8.0\TestLibrary.dll***

***Файл - Programm.cs***

**record class Person(string Name, int Age);**

**// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

***Проект 2 - Test***

***Файл - Programm.cs***

**using System.Reflection;**

**string path =**

**"C:\\VSProjects\\TestLibrary\\bin\\Debug\\net8.0\\TestLibrary.dll";**

**Assembly assembly = Assembly.LoadFrom(path);**

**// или**

**Assembly assembly = Assembly.Load("TestLibrary");**

**string fullName = assembly.FullName;**

**// TestLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null**

**Type[] types = assembly.GetTypes();**

**// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

### Позднее связывание

**Позднее связывание позволяет создавать экземпляры некоторого типа, а также использовать его во время выполнения приложения. С помощью статического метода Activotor.CreateInstance() класса System.Activator можно создавать экземпляры заданного типа.**

**// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

***Проект 1 - C:\TestLibrary\bin\Debug\net8.0\TestLibrary.dll***

***Файл - Programm.cs***

**class Program**

**{**

**static int Method(int n) => n \* n;**

**}**

**// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

***Проект 2 - Test***

***Файл - Programm.cs***

**using System.Reflection;**

**string path =**

**"C:\\VSProjects\\TestLibrary\\bin\\Debug\\net8.0\\TestLibrary.dll";**

**Assembly assembly = Assembly.LoadFrom(path);**

**Type? type = assembly.GetType("Program");**

**MethodInfo? method = type?.GetMethod("Method",**

**BindingFlags.NonPublic**

**| BindingFlags.Static);**

**object? result = method?**

**.Invoke(null, new object[] { 5 }); // 25**

**// первый параметр - объект, для которого вызывается метод,**

**// - null, т.к. метод статический**

**// второй параметр - набор параметров метода**

**// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

## Атрибуты в .NET

**Атрибуты могут применяться как ко всему типу (классу, интерфейсу и т.д.), так и к отдельным его частям (методу, свойству и т.д.). Основу атрибутов составляет класс System.Attribute.**

**using System;**

**class AgeValidationAttribute : Attribute**

**{**

**public int Age { get; }**

**public AgeValidationAttribute() { }**

**public AgeValidationAttribute(int age) => Age = age;**

**}**

**[AgeValidation(18)] // или [AgeValidation(Age = 18)]**

**class Person(string name, int age)**

**{**

**public string Name { get => name; }**

**public int Age { get => age; }**

**}**

**class Program**

**{**

**public static void Main()**

**{**

**Person tom = new Person("Tom", 35);**

**Person bob = new Person("Bob", 16);**

**bool tomIsValid = ValidateUser(tom); // true**

**bool bobIsValid = ValidateUser(bob); // false**

**}**

**static bool ValidateUser(Person person)**

**{**

**Type type = typeof(Person);**

**// получаем все атрибуты класса Person**

**object[] attributes = type.GetCustomAttributes(false);**

**// проходим по всем атрибутам**

**foreach (Attribute attribute in attributes)**

**{**

**// если атрибут представляет тип AgeValidationAttrinbute**

**if (attribute is AgeValidationAttribute ageAttribute)**

**return person.Age >= ageAttribute.Age;**

**}**

**return true;**

**}**

**}**

### Ограничение применения атрибута

**С помощью атрибута AttributeUsage можно ограничить типы, к которым будет применяться атрибут. Например, мы хотим, чтобы выше определенный атрибут мог применяться только к классам или методам:**

**[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]**

**class AgeValidationAttribute : Attribute**

**{**

**//....................................**

**}**

**Ограничение задает перечисление AttributeTargets, которое может принимать еще ряд значений:**

* **All: используется всеми типами**
* **Assembly: атрибут применяется к сборке**
* **Constructor: атрибут применяется к конструктору**
* **Delegate: атрибут применяется к делегату**
* **Enum: применяется к перечислению**
* **Event: атрибут применяется к событию**
* **Field: применяется к полю типа**
* **Interface: атрибут применяется к интерфейсу**
* **Method: применяется к методу**
* **Property: применяется к свойству**
* **Struct: применяется к структуре**

Dynamic Language Runtime

## DLR в C#**.** Ключевое слово **dynamic**

**Ключевое слово dynamic позволяет опустить проверку типов во время компиляции. Объекты, объявленные как dynamic, могут в течение работы программы менять свой тип. Например:**

**dynamic x = 3; // int**

**x++; // c dynamic так можно, а, например,**

**// object x = 3; x++; - так нельзя**

**x = "Hello world"; // string**

**x = new Person(); // Person**

**// еще пример со свойством и методом:**

**class Person**

**{**

**dynamic Age { get; set; } // свойства**

**public Person(dynamic age) ... // параметры**

**public dynamic Method(dynamic a) ... // возвращаемый тип**

**}**

## DynamicObject и ExpandoObject

### ExpandoObject

**class ExpandoObject в namespace System.Dynamic позволяет создавать динамические объекты, их свойства и методы:**

**// определяем объект, который будет хранить рад значений**

**dynamic person = new System.Dynamic.ExpandoObject();**

**person.Name = "Tom";**

**person.Age = 33;**

**person Languages = new List<string> { ... };**

**// объявляем метод**

**person.IncrementAge = (Action<int>) (x => person.Age += x);**

### DynamicObject

**DynamicObject похож на ExpandoObject, но используется в более сложных ситуациях.**

**Необходимо создать наследник DynamicObject и реализовать его методы:**

* **TryBinaryOperation(): выполняет бинарную операцию между двумя объектами. Эквивалентно стандартным бинарным операциям, например, сложению x + y)**
* **TryConvert(): выполняет преобразование к определенному типу. Эквивалентно базовому преобразованию в C#, например, (SomeType) obj**
* **TryCreateInstance(): создает экземпляр объекта**
* **TryDeleteIndex(): удаляет индексатор**
* **TryDeleteMember(): удаляет свойство или метод**
* **TryGetIndex(): получает элемент по индексу через индексатор. В C# может быть эквивалентно следующему выражению int x = collection[i]**
* **TryGetMember(): получаем значение свойства. Эквивалентно обращению к свойству, например, string n = person.Name**
* **TryInvoke(): вызов объекта в качестве делегата**
* **TryInvokeMember(): вызов метода**
* **TrySetIndex(): устанавливает элемент по индексу через индексатор. В C# может быть эквивалентно следующему выражению collection[i] = x;**
* **TrySetMember(): устанавливает свойство. Эквивалентно присвоению свойству значения, например: person.Name = "Tom"**
* **TryUnaryOperation(): выполняет унарную операцию подобно унарным операциям в C#: x++**

**using System.Dynamic;**

**class PersonObject : DynamicObject**

**{**

**// словарь для хранения всех свойств**

**// ключи - названия свойств и методов,**

**// значения - значения свойств**

**Dictionary<string, object> members = new Dictionary<string, object>();**

**// установка свойства**

**// binder хранит название свойства, а value - значение**

**public override bool TrySetMember(SetMemberBinder binder, object? value)**

**{**

**if (value in not null)**

**{**

**members[binder.Name] = value;**

**return true;**

**}**

**return false;**

**}**

**// получение свойства**

**// binder хранит название свойства,**

**// а result - значение получаемого свойства**

**public override bool TryGetMember(**

**GetMemberBinder binder,**

**out object? result)**

**{**

**result = null;**

**if (members.ContainKey(binder.Name))**

**{**

**result = members[binder.Name];**

**return true;**

**}**

**return false;**

**}**

**// вызов метода**

**// binder хранит название метода, args - параметры, result - результат**

**public override bool TryInvokeMember(**

**InvokeMemberBinder binder,**

**object?[]? args,**

**out object? result)**

**{**

**result = null;**

**if (args?[0] is int number)**

**{**

**// получаем метод по имени**

**dynamic method = members[binder.Name];**

**// вызываем метод, передавая его параметру значение args?[0]**

**result = method(number);**

**}**

**// если result не равен null, то вызов прошел успешно**

**return result != null;**

**}**

**}**

**// создаем объект**

**dynamic person = new PersonObject();**

**// устанавливаем ряд свойств**

**// (вызывается метод TrySetMember)**

**person.Name = "Tom";**

**person.Age = 33;**

**// определяем метод для изменения свойства Age**

**// return person.Age - вызывается метод TryGetMember**

**Func<int, int> increment = (int n) =>**

**{**

**person.Age += n;**

**return person.Age;**

**};**

**person.IncrementAge = increment;**

**// вызов метода**

**// (происходит обращение к методу TryInvokeMember)**

**person.IncrementAge(4);**

## Использование IronPython в .NET

**Для использования IronPython необходимо добавить несколько пакетов через NuGet:**

***Dependencies -> ПКМ -> Manage NuGet Packages...***

***DynamicLanguageRuntime by DLR ... -> Install***

***(появится библиотека Microsoft.Scripting)***

***IronPython by IronPython ... -> Install***

***(появится библиотека IronPython)***

**using IronPython.Hosting;**

**using Microsoft.Scripting.Hosting;**

**ScriptEngine engine = Python.CreateEngine();**

**engine.Execute("здесь пишется команда на Phyton");**

**// например**

**engine.Execute("print('Hello world')");**

**// или можно создать текстовый файл с кодом на Python,**

**// например, файл hello.py (py - python) в папке D:/PythonFolder/**

**print("hello world")**

**// и вызвать его из программы**

**engine.ExecuteFile("D://PythonFolder//hello.py");**

### ScriptScope

***Файл hello.py***

**x = 10**

**z = x + y**

**print(z)**

***Программа на C#:***

**using IronPython.Hosting;**

**using Microsoft.Scripting.Hosting;**

**int y = 22;**

**ScriptEngine engine = Python.CreateEngine();**

**ScriptScope scope = engine.CreateScope();**

**// SetVariable устанавливает переменные в скрипте**

**scope.SetVariable("y", y);**

**engine.ExecuteFile("D://PythonFolder//hello.py", scope);**

**// GetVariable возвращает переменные из скрипта**

**dynamic x = scope.GetVariable("x");**

**dynamic z = scope.GetVariable("z");**

**string result = $"{x} + {y} = {z}"; // 10 + 22 = 32**

### Вызов функций из IronPython

***Файл hello.py***

**def square(n):**

**return n \* n**

***Программа на C#:***

**int number = 5;**

**ScriptEngine engine = Python.CreateEngine();**

**ScriptScope scope = engine.CreateScope();**

**engine.ExecuteFile("D://PythonFolder//hello.py");**

**// возвращает объект функции**

**dynamic square = scope.GetVariable("square");**

**// вызываем функцию и получаем результат**

**dynamic result = square(number); // 25**

Сборка мусора, управление памятью и указатели

## Сборщик мусора в C#

**void Test()**

**{**

**// new - CLR выделяет память под объект Person в куче**

**// и под ссылку на объект в стеке**

**Person person = new Person();**

**...**

**}**

**// вызываем метод**

**// в стеке выделяется память под метод;**

**Test();**

**После отработки метода ссылка на объект в стеке удаляется, а место в куче будет подчищено, когда потребуется дополнительная память. Объекты в куче неупорядочены, между ними есть пустоты. Поэтому после очистки мусора оставшиеся объекты перемещаются в один непрерывный блок. Происходит обновление ссылок.**

**Для крупных объектов (> 85 000 байт) есть своя куча - Large Object Heap. Сжатие в ней не проводится из-за больших издержек.**

**В куче есть 3 поколения объектов: 0, 1 и 2.**

1. **0 - ни разу не подвергались сборке мусора;**
2. **1 - пережили одну сборку мусора;**
3. **2 - пережили более одной сборки мусора.**

**Сборщик мусора вначале анализирует 0 поколение, потом перемещает оставшиеся объекты в 1 поколение. Если требуется еще память, то же происходит с 1 поколением.**

### Класс System.GC

**Методы и свойства класса System.GC:**

* **Метод AddMemoryPressure информирует среду CLR о выделении большого объема неуправляемой памяти, которую надо учесть при планировании сборки мусора. В связке с этим методом используется метод RemoveMemoryPressure, который указывает CLR, что ранее выделенная память освобождена, и ее не надо учитывать при сборке мусора.**
* **Метод Collect приводит в действие механизм сборки мусора. Перегруженные версии метода позволяют указать поколение объектов, вплоть до которого надо произвести сборку мусора**

**GC.Collect();**

**GC.Collect(1);**

**GC.Collect(1, GCCollectionMode.Forced);**

**GCCollectionMode может принимать 3 значения:**

* + **Default: значение по умолчанию для данного перечисления (Forced)**
  + **Forced: вызывает немедленное выполнение сборки мусора**
  + **Optimized: позволяет сборщику мусора определить, является ли текущий момент оптимальным для сборки мусора**
* **Метод GetGeneration(Object) позволяет определить номер поколения, к которому относится переданый в качестве параметра объект**
* **Метод GetTotalMemory возвращает объем памяти в байтах, которое занято в управляемой куче**

**long totalMemory = GC.GetTotalMemory(false);**

* **Метод WaitForPendingFinalizers приостанавливает работу текущего потока до освобождения всех объектов, для которых производится сборка мусора**

**GC.WaitForPendingFinalizers();**

## Финализируемые объекты

**CLR управляет управляемыми объектами и сборщик мусора их удаляет. Чтобы сборщик мусора справлялся с неуправляемыми объектами (теми, которые обращаются к API ОС - подключение к файлам, БД, сети), необходимо реализовать один из двух механизмов:**

1. **Создание деструктора**
2. **Реализация классом интерфейса System.IDisposable**

### Создание деструкторов

**Деструкторы можно определить только в классах, только один на класс. Деструкторы в отличие от конструкторов не могут иметь модификаторов доступа и параметров. Деструкторы носят имя класса, перед которым стоит тильда (~).**

**class Person**

**{**

**public Person() => ... // конструктор**

**~Person() => ... // деструктор**

**...**

**}**

**На деле компилятор компилирует деструктор в метод Finalize.**

**protected override void Finalize()**

**{**

**try**

**{**

**// здесь идут инструкции деструктора**

**}**

**finally**

**{**

**base.Finalize();**

**}**

**}**

**Чтобы вызвать сборщик мусора, можно вызвать его статический метод GC.Collect(). А GC вызовет Finalize. Но не факт, что это произойдет сразу же.**

### Интерфейс IDisposable

**IDisposable имеет один метод Dispose, в котором должно происходить освобождение неуправляемых ресурсов.**

**class Person : IDisposable**

**{**

**public void Dispose() => ...**

**...**

**}**

**Для вызова метода Dispose предпочтительно использовать конструкцию try...finally:**

**Person? person = null;**

**try**

**{**

**person = new Person();**

**...**

**}**

**finally**

**{**

**person.Dispose();**

**}**

### Комбинирование подходов

**Microsoft предлагает использовать следующий шаблон:**

**public class SomeClass : IDisposable**

**{**

**private bool disposed = false;**

**// реализация интерфейса IDisposable**

**public void Dispose()**

**{**

**// освобождаем неуправляемые ресурсы**

**Dispose(true);**

**// подавляем финализацию, т.е. запрещаем вызвать**

**// метод Finalize (который вызывает деструктор) для данного объекта**

**GC.SuppressFinalize(this);**

**}**

**// перегруженный метод Dispose(bool disposing):**

**protected virtual void Dispose(bool disposing)**

**{**

**if (disposed) return;**

**if (disposing)**

**{**

**// Освобождаем управляемые ресурсы**

**...**

**}**

**// освобождаем неуправляемые объекты**

**...**

**disposed = true;**

**}**

**// Деструктор**

**~SomeClass()**

**{**

**Dispose(false);**

**}**

**}**

**При вызове Dispose() освобождаются вначале управляемые, затем неуправляемые объекты. При вызове деструктора освобождаются только неуправляемые объекты, т.к. мы не уверены, что они все еще находятся в памяти - мы полагаемся на их деструкторы.**

### Общие рекомендации по использованию Finalize и Dispose

* **Деструктор следует реализовывать только у тех объектов, которым он действительно необходим, так как метод Finalize оказывает сильное влияние на производительность**
* **После вызова метода Dispose необходимо блокировать у объекта вызов метода Finalize с помощью GC.SuppressFinalize**
* **При создании производных классов от базовых, которые реализуют интерфейс IDisposable, следует также вызывать метод Dispose базового класса:**

**public class Derived : Base**

**{**

**private bool IsDisposed = false;**

**protected override void Dispose(bool disposing)**

**{**

**if (IsDisposed) return;**

**if (disposing)**

**{**

**// Освобождаем управляемые ресурсы**

**...**

**}**

**IsDisposed = true;**

**// Обращение к методу Dispose базового класса**

**base.Dispose(disposing);**

**}**

**}**

* **Отдавайте предпочтение комбинированному шаблону, реализующему как метод Dispose, так и деструктор**

## Конструкция **using**

**Для автоматического вызова метода Dispose можно использовать конструкцию using:**

**// Person должен реализовывать IDisposable**

**using (Person person = new Person())**

**{**

**// person доступен только в этом блоке кода**

**...**

**}**

***Начиная с C#8.0:***

**void Test()**

**{**

**using Person person = new Person();**

**// person доступен в области метода Test**

**...**

**}**

### Освобождение множества ресурсов

**using (Person tom = new Person())**

**{**

**using (Person bob = new Person())**

**{**

**// доступны tom и bob**

**...**

**} // вызывается Disopose для bob**

**...**

**} // вызывается Dispose для tom**

**// или**

**using (Person tom = new Person())**

**using (Person bob = new Person())**

**{**

**// доступны tom и bob**

**...**

**} // вызывается Disopose для tom и bob**

***Начиная с C#8.0:***

**void Test()**

**{**

**using Person tom = new Person();**

**using Person bob = new Person();**

**// доступны tom и bob**

**...**

**} // вызывается Disopose для tom и bob**