Разработка приложений на платформе .NET

Лекция 3

Способы передачи параметров
Типы, допускающие неопределенное значение
Работа со строками

Сегодня

- Способы передачи параметров
- Типы, допускающие неопределенное значение
- Работа со строками
  - **String**
  - Динамически изменяемые строки – класс **StringBuilder**
  - Регулярные выражения – класс **Regex**

Сегодня

- Способы передачи параметров
- Типы, допускающие неопределенное значение
- Работа со строками
  - String
  - Динамически изменяемые строки – класс StringBuilder
  - Регулярные выражения – класс Regex

Способы передачи параметров

- В C# два способа передачи параметров
  - по значению (по умолчанию)
  - по ссылке
    - **ref**
    - **out**
    - **in (C# 7.2)**
- Для каждого из способов важно понимать особенности при передаче:
  - ссылочного типа
  - типа-значения

Передача по значению

- При передаче типа-значения
  - Создается его локальная копия
  - Любые модификации внутри метода не влияют на исходное значение
- При передаче ссылочного типа
  - Передается значение ссылки
  - Любые модификации внутри метода влияют на исходное значение
  - Само значение исходной ссылки изменить нельзя (присвоить ссылку на другой объект)
- Пример:
  - `Console.WriteLine("Hello, World!");`

Передача по ссылке (**ref**, **out**, **in**)

- При передаче типа-значения
  - Передается ссылка на объект
  - Модификации объекта внутри метода влияют на исходные значения
  - Само значение исходной ссылки тоже можно изменить (присвоить ссылку на другой объект). Кроме **in**
- При передаче ссылочного типа
  - Передается ссылка на ссылку
  - Можно присвоить ссылке ссылку на другой объект
- **out** аналогичен **ref**, однако не предполагает предварительной инициализации переменной
- **in** запрещает изменить сам переданный параметр (ссылку)
- Пример:
  - `Interlocked.Add(ref number);`
  - `int.TryParse(inputString, out int i);`

* **ref**, **out** и **in** имеют одинаковую сигнатуру. Поэтому нельзя создать 2 метода, которые будут отличаться только наличием у одного параметра **ref**, а у другого **out** или **in**

Примеры передачи параметров

- По значению:
  - Console.WriteLine(“Hello, World!”);
- По ссылке:
  - При описании метода

```
void Swap(ref int x, ref int y)
{
    int z;   z = x;   x = y;   y = z;
}
```

  - При использовании **out**, параметр должен быть проинициализирован до любого выхода из метода
  - Не забывайте указывать **ref** и **out** при передаче фактических параметров!
  - in указывать при вызове не нужно

```
int x, y;
Swap(x, y); // неправильно
Swap(ref x, ref y); // правильно
```

Перегрузки метода с in параметром

- Передача по ссылке in появилась в C# 7.2
- Не обязательно указывать in при вызове, но не возбраняется
  - static Point Sum(in Point point1, in Point point2) { … };
  - Point p = Sum(p1, p2);
  - Point p = Sum(in p1, in p2);
- Неоднозначность при вызове
  - static void Print(DateTime date) { … };
  - static void Print(in DateTime date) { … };
  - Print(DateTime.Now); // В C# 7.3 – устранили неоднозначность, вызовется первый метод (без in)
  - Для избегания конфликта можно явно указать in при вызове метода
    Print(in DateTime.Now);

Демонстрация

Передача параметров

Переменное число параметров

- Для объявления переменного числа параметров используется **params**

  params *param_type*[] *param_name*
- **params** должен идти в конце списка формальных параметров
- С таким параметром можно работать, как с обычным массивом
- На место **params** можно передавать:
  - массив параметров
  - параметры через запятую
  - ничего

- Примеры:
  - `public static void WriteLine(string format, params object[] arg) {}`
  - `public static string Concat(params object[] args )`

  - `string.Concat(arrayVar);`
  - `string.Concat(new string[] {"5","4","3"});`
  - `string.Concat("Hello", " C#", " World");`
  - `string.Concat();`

Параметры по умолчанию

- Имя_типа имя_переменной **= значение**
- В определении метода должны идти последними
- Примеры:

```
public Complex(double re, double im = 0.0){}
public void Vector3D(double x = 0.0, double y = 0.0, double z = 0.0){ }

Complex c = new Complex(5.7);
Complex c = new Complex(5.7, 8.3);
Vector3D(5);
Vector3D(5, 6);
Vector3D(5, 9, 12);
```

- Компилируется в один метод с максимальным числом параметров, а не в несколько со всеми возможными вариантами параметров
  - Добавление нового параметра со значением по умолчанию в существующий метод приведет к необходимости перекомпилирования всех зависимых сборок, использующих этот метод

Передача параметров по имени

- Синтаксис:

  Имя_метода(имя_параметра: значение

  [,имя_параметра: значение…])

- Позволяет задавать только указанные по имени параметры
- Можно менять порядок следования параметров
- Можно совмещать позиционную передачу параметров и передачу параметров по имени

- Примеры:
  - `public Complex (int re, int im) {…}`
  - `Complex c = new Complex (im : 5, re : 6);`

  - `void Vector3D(int x, int y=0, int z =0);`
  - `Vector3D(3, y:4, z:5); // Передача по положению и по имени`
  - `Vector3D(3, z:5); // Передача по положению, по имени и значения по умолчанию`

Возвращаемые значения по ссылке

- Возвращает ссылку на объект (алиас)
- При описании метода

```
static ref int Find(int[] array, int findNumber)
{
    for (int i = 0; i < array.Length; i++)
    {
        if (array[i] == findNumber)
        return ref array[i];
    }
    throw new IndexOutOfRangeException();
}
```

- При вызове метода

```
ref int i = ref Find(arr,3);
 i = 27;
```

- Изменяет само значение в массиве

- C# 7

Сегодня

- Способы передачи параметров
- Типы, допускающие неопределенное значение
- Работа со строками
  - String
  - Динамически изменяемые строки – класс StringBuilder
  - Регулярные выражения – класс Regex

Типы-значения, допускающие неопределенное значение

- Типы с неопределенным значением – расширение типов-значений (ValueType) неопределенным значением **null**
- Структура

  struct Nullable<T>

  where T : struct, new()

- Свойства

  bool HasValue { get; } – определено ли значение или объект - **null**

  T Value {get; } - значение

- Синтаксис объявления

```
Nullable<value_type>
value_type?            - сокращенный вариант
```

- Примеры:

```
int? nj = 5, nj = null;
Nullable<float> f = null;
DateTime? ndt = GetNullableDate(), ndt2 = null;
```

Использование Nullable Types

- Неопределенные значения в базах данных*
- Трехзначная логика `bool`?
- Возможность неопределенного возвращаемого значения
- Расширение типов-значений неопределенным значением

Nullable Types

- Существует неявное преобразование в *value_type?* из *value_type*
- Обратное преобразование - только явное:

```
int? nj;
int j = (int)nj;
```

- Свойства **HasValue** – показывающее определено ли значение и **Value** - значение
  - int i; int? age = 5;
  - if (age.HasValue) i = age.Value;
- Значение по умолчанию - **null**
- Можно получать доступ ко всем членам объекта типа

```
DateTime? d = new DateTime();
d.Value.AddDays(4);
```

Операции с Nullable Types

- Можно выполнять обычные операции (переопределены)
  - `int? + int? => int?`
  - Если одно из значений не определено (**null**), то и результат не определен
  - Если оба определены, то результат – сумма этих значений
- Можно сравнивать с **null**

```
if (ni == null) ni = 5;
```

Оператор ??

- *object1 ?? object2*
  - Если **object1** равен **null**, то результатом этого выражения будет **object2**, иначе **object1**
- Использование
  - Присваивание значения по умолчанию, если не определен
- Применимо к ссылочным и **Nullabe<T>** типам
- Примеры:

```
int? a;
int i = a ?? 5;
Если a == null, то i = 5,
    иначе i = a.Value
```

Оператор условного доступа - ?.

- Применим к ссылочным и Nullabe<T> типам
- *string s = object1?.value1*
  - Если object1 равен null, то результатом этого выражения будет s= null, иначе s = object1.value1
- *int? item = collection?[index];*

  Если collection равена null, то результатом этого выражения будет null, иначе (int?)collection[index]
- Использование
  - Сильно сокращает запись

```
public static string GetTrimedFio(Employee employee)
{
    string trimedFio = employee?.Fio?.Trim();

    if (employee != null && employee.Fio != null) trimedFio = employee.Fio.Trim();
    else  trimedFio = null;
    return trimedFio;
}
```

Демонстрации

Nullable Types

Сегодня

- Способы передачи параметров
- Типы, допускающие неопределенное значение
- Работа со строками
  - **String**
  - Динамически изменяемые строки – класс **StringBuilder**
  - Регулярные выражения – класс **Regex**

Тип string

- **String** – ссылочный тип
- Функционирует как тип-значение
- Строка неизменяема

- Сравнение строк **==**, **!=**
  - Сравнение содержимого (посимвольно с учетом регистра), а не ссылок на объект
- Конкатенация строк **+** или использование статического метода **Concat()**
  - string s = s1 + ”еще одна строчка” + s2;
  - string s += “добавленная строчка” – создает новый экземпляр типа string

- Свойства:
  - Свойство **Length** – возвращает длину строки (только чтение)
  - Индексатор **[ ]** – возвращает указанный символ в строке (только на чтение)
    - char c = myString[5];
  - Статическое поле **Empty** – представляет пустую строку
    - string s = string.Empty;

Методы тип string

- Статический метод Compare() – сравнение строк, возможно с учетом культуры и регистра
  - string result = string.Compare(string1, string2, new CultureInfo("en-US"), CompareOptions.IgnoreCase);
- Методы IndexOf(), LastIndexOf() – возвращают позицию символа или подстроки
- Метод Contains() – возвращает **true**, если строка содержит подстроку
- Статический метод Format() – создание форматированной строки
  - string result = String.Format("Температура {0:d}:\nв {1,11}: {2} градусов", date, time, temp);
- Метод Insert() – возвращает новую строку, в которой указанная подстрока вставлена в указанную позицию
  - string result = s.Insert(2, "вставляемая подстрока");
- Методы Remove(), Replace() – возвращают новые строки, в которых удалена или замена подстрока
- Метод Split() – разбивает строку на несколько строк по определенному символу
- Метод Trim() – удаляет все вхождения определенного набора символов сначала и с конца строки
- Методы ToUpper(), ToLower() – преобразование строки в верхний, нижний регистры (возвращают новые строки)
- Метод Join() – объединяет коллекцию в строку использую разделитель между элементами
  - int[] values = {5, 4189, 11434, .366 };
  - Console.WriteLine(string.Join(";", values));
- Статические методы IsNullOrEmpty(), IsNullOrWhiteSpace() – проверяют строку на пустоту

ВНИМАНИЕ !!!

Методы возвращают новый экземпляр **string** и не меняют текущую строку

Класс StringBuilder

- Класс предназначен для работы с часто изменяющимися строковыми данными
- Представляет изменяемую строку символов
- Расположен пространстве имен **System.Text**
- Методы:
  - **Append()** – добавляет строковое представление типа (подстроку) в конец (перегружен для различных типов данных)
  - **AppendFormat()** – добавляет форматированную строку
  - **AppendLine()** – добавляет строку и символ перевода строки
  - **Insert()** – вставляет строковое представление типа (подстроку) в указанное место (перегружен для различных типов данных)
  - **Replace()** – заменяет символы или подстроки на новые
  - **Clear()** – очищает содержимое
  - **ToString()** – возвращает содержащуюся строку
- В отличии от **string** меняет сам объект, а не возвращает новый при изменении данных

Форматированный вывод

- Применяется для форматирования вывода строки
  - Console.Write(), Console.WriteLine(), string.Format(), stringBuilder.AppendFormat()
- Первый параметр – строка-шаблон
- Метки-заполнители {0}, {1}, {2} …
  - Вместо меток подставляются параметры метода, следующие за строкой
  - Следующие параметры нумеруются с 0
  - Метки-заполнители {0}, {1}, {2} могут идти в произвольном порядке и повторятся сколько угодно раз
  - При недостаточности параметров будет вызвано исключение
  - Console.Write("x = {0}, y = {2}, z = {1}, x ={0}", dx, dz, dy);
- Форматирование числовых данных
  - C, c – денежный формат
  - D, d – числовой формат (с минимальный кол-вом цифр)
  - E, e – экспоненциальный формат числа
  - F, f – формат числа с фиксированной точкой
  - X, x – шестнадцатеричный формат
  - P, p – представление в процентах
  - G, g – общий формат

```
string s = string.Format("{0:C}", value);
```

Управляющие символы

- Управляющие символы начинаются с \
  - \n – перевод строки (для Windows)
  - \t – символ табуляция
  - \r – возврат каретки
  - \a – звуковой сигнал
  - \\ – символ \
  - * – символ *
  - \" – символ “
  - \' – символ ’
  - string s = “d:\\MyFolder\\SubFolder\\Example.txt”
- Дословные строки – @
  - Отключение управляющих символов
  - Сохраняет пробелы, символы перевода строк и т.д.
  - string s = @“d:\MyFolder\SubFolder\Example.txt”
  - string s = @“String на две
    строки (да еще и с пробелами)”

Interpolated Strings

- Используется для создания строк.
- Выглядит как шаблонная строка, которая содержит выражения. Интерполированное строковое выражение создает строку, заменяя содержащиеся выражения представлениями ToString() результатов выражений.
- Интерполированную строку проще понять
- Структура интерполированной строки

  $ “<text> { <interpolation-expression> <optional-comma-field-width> <optional-colon-format> } <text> …“
- Примеры:
  - string s = $"hello, {name}“
  - sb.Append($"Hello, {name}");
  - Console.WriteLine($"{person.Name, 20} is {person.Age:D3} year {(person.Age == 1 ? "" : "s")} old.“)

Демонстрации

Работа со строками

Regex

- Позволяет осуществлять поиск, замену, проверку, разбор строки, используя регулярные выражения
- Пространство имен **System.Text.RegularExpressions**
- Позволяет работать как с статическими методами, так и создать экземпляр класса для проведения множества однотипных операций, используя один и тот же паттерн
- Методы (все методы экземпляра имеют и соответствующий статический вариант):
  - IsMatch() – проверяет строку на соответствие регулярному выражению (статический метод и метод экземпляра)
  - Match() – возвращает первое вхождение регулярного выражения в строку (статический метод и метод экземпляра)
  - Matchs() – возвращает коллекцию всех вхождений регулярного выражения в строку (статический метод и метод экземпляра)
  - Replace() – заменяет все вхождения регулярного выражения на новую подстроку
  - Split() – разделяет строку на подстроки по позициям совпадения с регулярным выражениям (найденная строка выкидывается). Статический метод и метод экземпляра

- Примеры:
  - `Regex r = new Regex(@"^\d{3}-\d{2}-\d{2}$");`
  - `bool ok = r.IsMatch("555-55-55");`
  - `bool validTel = Regex.IsMatch("555-55-55", @"^\d{3}-\d{2}-\d{2}$");`
  - `string s = Regex.Replace(inputString, @“[^\w\.@-]”, “”); // Удаление недопустимых символов`

Символы в регулярных выражения

^	Начало строки
$	Конец строки
\n	Символ новой строки
\r	Возврат каретки
\x20	ASCII символ в шестнадцатеричном формате (2 разряда)
\u0020	Unicode символ в шестнадцатеричном формате (4 разряда)
\	Позволяет использовать управляющие символы как простой символ. Например * просто *, а не повтор \
*	Повтор предшествующего символа или подстроки от 0 до ∞ раз
+	Повтор предшествующего символа или подстроки от 1 до ∞ раз
?	Предшествующий символ или подстрока или его (ее) отсутствие
{n}	Где n – число повторов предшествующего символа
{n,}	Где n – минимальное число повторов предшествующего символа
{n,m}	Где n, m – минимальное и максимальные числа повторов предшествующего символа
x|y	Соответствует x или y
[xyz]	Соответствует любому символу из набора
[a-z]	Соответствует любому символу из перечисления
[^a-z]	Соответствует любому символу, кроме перечисленных
.	Соответствует любому одному символу
\d	Соответствует цифрам (аналог [0-9]).
\D	Соответствует не цифрам (аналог [^0-9]).
\s	Соответствует пробелу, табуляции или разрыву страницы
\S	Соответствует любому символу кроме пробела, табуляции и разрыва страницы
\w	Любой алфавитно-цифровой символ, включая символ подчеркивания (аналог [A-Za-z0-9_])
\W	Любой символ кроме алфавитно-цифровых символов и символа подчеркивания (аналог [^A-Za-z0-9_])

Демонстрации

Работа с регулярными выражениями