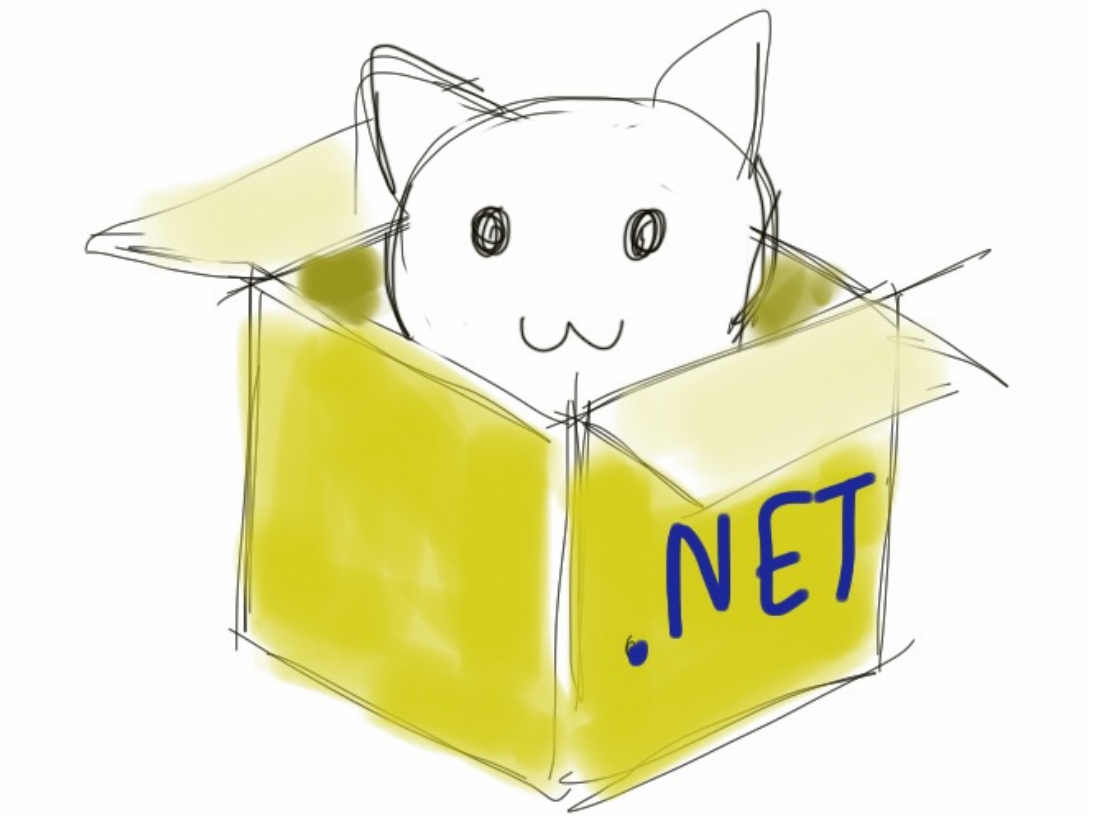
Задачник.NET



«Задачник.NET»

Под редакцией Ивана Пащенко

Версия текста: v-2014-Nov-10
Исходные коды и актуальная версия текст доступны по адресу:
https://github.com/AndreyAkinshin/ProblemBook.NET

Оглавление

Введение **4**

Задачи **6**
1 ООП (Задачи) . . . 6
2 LINQ (Задачи) . . . 10
3 Математика (Задачи) . . . 13
4 Значимые типы (Задачи) . . . 16
5 Строки (Задачи) . . . 18

Решения **22**
6 ООП (Решения) . . . 22
7 LINQ (Решения) . . . 24
8 Математика (Решения) . . . 30
9 Значимые типы (Решения) . . . 32
10 Строки (Решения) . . . 34

Библиография **39**

Введение

Данная книга представляет собой сборник задач на знание платформы .NET и языка программирования C#. Дабы у читателя и автора не возникло недопонимания, сразу хочется сказать, чем *не является* эта книга:

- Эта книга не является универсальным способом проверить ваше знание платформы .NET. Если вы легко прорешали все задачки, то это не значит, что вы замечательный .NET-программист. А если вы встретили много новых для себя вещей, то из этого вовсе не следует, что вы плохо знаете .NET.
- Эта книга не является подборкой новых, ранее нигде не виданных задач. Многие примеры можно встретить в литературе, в вопросах на Stackoverflow, в программистских блогах. Просто потому, что они уже давно стали классическими.
- Эта книга не является ориентированной на тех, кто уже считает себя Senior .NET Developer и хочет узнать много нового.

Так чем же тогда является эта книга? Задачник.NET — это попытка собрать в одном месте разные интересные практические задания на знание платформы. Скорее всего, наибольшую пользу извлекут для себя .NET-разработчики, которое ещё просто не сталкивались с теми или иными областями. Задачи разбиты на главы, так что можно читать не всё подряд, а только вопросы из тех областей, которые для вас представляют интерес. В этой книге вы не найдёте глубоко философских вопросов типа «Что такое класс?» или «Зачем нужен полиморфизм?». Большая часть заданий представляет собой фрагмент C#-кода, для которого необходимо определить результат работы. Каждый вопрос снабжён ответом с описанием того, отчего .NET ведёт себя именно так.

Рекомендуется относиться к заданиям не как к способу проверить ваши знания, а как к начальной точке для обсуждения тех или иных аспектов платформы. Если вы обнаружили что-то новое для себя, то это отличный повод поизучать .NET чуть подробней. Попробуйте поиграться с кодом: модифицируйте его и изучайте, как изменения влияют на результат. Почитайте соответствующую литературу. Подумайте, как новые знания могут пригодиться вам в вашей работе.

Задачник.NET распространяется в электронном виде, разработка ведётся на GitHub. Стоит также отметить, что на сегодняшний задачник далёк от финального варианта. Книга будет пополняться новыми задачами, а старые будут уточняться и совершенствоваться.

Технические детали

На сегодняшний день существует две популярные реализации платформы .NET: оригинальный Microsoft .NET Framework (далее — MS.NET) и Mono. Как известно, поведение

некоторых фрагментов кода может измениться при смене реализации. Точно также, поведение может измениться при изменении версии CLR или архитектуры процессора. Если результат работы приведённого фрагмента кода зависит от окружения, то скорее всего в ответах будет дано объяснение: в каком окружении какой ответ получится почему. Однако, это *не гарантируется*. Если в одном из примеров вы получили ответ, отличный от данного в книге, то просьба связаться с автором, чтобы он исправил эту досадную недоработку.

Примеры кода во всех заданиях можно запускать из программы LINQPad (в режиме C# Statements или C# Program, в зависимости от наличия метода `Main`), если предварительно подключить следующие пространства имён (Query —> Query Properties —> Additional Namespace Imports):

```
System.Globalization
System.Runtime.InteropServices
System.Runtime.CompilerServices
```

Благодарности

Автор искренне благодарит главного редактора Ивана Пащенко за полезные обсуждения в ходе работы над книгой. Этот человек внёс и продолжает вносить множество конструктивных комментариев с новыми интересными вопросами и улучшением старых.

Автор выказывает признательность Екатерине Демидовой за любезно предоставленные иллюстрации.

Задачи

1 ООП (Задачи)

1. Что выведет следующий код?

```
void Foo(object a)
{
  Console.WriteLine("object");
}
void Foo(object a, object b)
{
  Console.WriteLine("object, object");
}
void Foo(params object[] args)
{
  Console.WriteLine("params object[]");
}
void Foo<T>(params T[] args)
{
  Console.WriteLine("params T[]");
}
class Bar { }
void Main()
{
  Foo();
  Foo(null);
  Foo(new Bar());
  Foo(new Bar(), new Bar());
  Foo(new Bar(), new object());
}
```

Решение

2. Что выведет следующий код?

```
class Foo
{
  public virtual void Quux(int a)
  {
    Console.WriteLine("Foo.Quux(int)");
  }
}
class Bar : Foo
{
  public override void Quux(int a)
  {
    Console.WriteLine("Bar.Quux(int)");
  }
  public void Quux(object a)
  {
    Console.WriteLine("Bar.Quux(object)");
  }
}
class Baz : Bar
{
  public override void Quux(int a)
  {
    Console.WriteLine("Baz.Quux(int)");
  }
  public void Quux<T>(params T[] a)
  {
    Console.WriteLine("Baz.Quux(params T[])");
  }
}
void Main()
{
  new Bar().Quux(42);
  new Baz().Quux(42);
}
```

Решение

3. Что выведет следующий код?

```
class Foo
{
  public Foo()
  {
    Quux();
  }
  public virtual void Quux()
  {
    Console.WriteLine("Foo.Quux()");
  }
}
class Bar : Foo
{
  protected string name;
  public Bar()
  {
    name = "Bar";
  }
  public override void Quux()
  {
    Console.WriteLine("Bar.Quux(),  " + name);
  }
  public void Quux(params object[] args)
  {
    Console.WriteLine("Bar.Quux(params object[])");
  }
}
class Baz : Bar
{
  public Baz()
  {
    name = "Baz";
    Quux();
    ((Foo)this).Quux();
  }
}
void Main()
{
  new Baz();
}
```

Решение

4. Что выведет следующий код?

```
class Foo
{
  protected class Quux
  {
    public Quux()
    {
      Console.WriteLine("Foo.Quux()");
    }
  }
}
class Bar : Foo
{
  new class Quux
  {
    public Quux()
    {
      Console.WriteLine("Bar.Quux()");
    }
  }
}
class Baz : Bar
{
  public Baz()
  {
    new Quux();
  }
}
void Main()
{
  new Baz();
}
```

Решение

2 LINQ (Задачи)

1. Что выведет следующий код?

```
public static string GetString(string s)
{
  Console.WriteLine("GetString: " + s);
  return s;
}
public static IEnumerable<string> GetStringEnumerable()
{
  yield return GetString("Foo");
  yield return GetString("Bar");
}
public static string[] EnumerableToArray()
{
  var strings = GetStringEnumerable();
  foreach (var s in strings)
    Console.WriteLine("EnumerableToArray: " + s);
  return strings.ToArray();
}
void Main()
{
  EnumerableToArray();
}
```

Решение

2. Что выведет следующий код?

```
IEnumerable<string> Foo()
{
  yield return "Bar";
  Console.WriteLine("Baz");
}
void Main()
{
  foreach (var str in Foo())
    Console.Write(str);
}
```

Решение

3. Что выведет следующий код?

```
var actions = new List<Action>();
foreach (var i in Enumerable.Range(1, 3))
  actions.Add(() => Console.WriteLine(i));
foreach (var action in actions)
  action();
```

Решение

4. Что выведет следующий код?

```
var list = new List<string> { "Foo", "Bar", "Baz" };
var query = list.Where(c => c.StartsWith("B"));
list.Remove("Bar");
Console.WriteLine(query.Count());
```

Решение

5. Что выведет следующий код?

```
var list = new List<string> { "Foo", "Bar", "Baz" };
var startLetter = "F";
var query = list.Where(c => c.StartsWith(startLetter));
startLetter = "B";
query = query.Where(c => c.StartsWith(startLetter));
Console.WriteLine(query.Count());
```

Решение

6. Что выведет следующий код?

```
int Inc(int x)
{
  Console.WriteLine("Inc: " + x);
  return x + 1;
}
void Main()
{
  var numbers = Enumerable.Range(0, 10);
  var query =
    (from number in numbers
     let number2 = Inc(number)
     where number2 % 2 == 0
     select number2).Take(2);
  foreach (var number in query)
    Console.WriteLine("Number: " + number);
}
```

Решение

7. В какой момент произойдёт `Exception`?

```
public static IEnumerable<int> GetSmallNumbers()
{
  throw new Exception();
  yield return 1;
  yield return 2;
}
void Main()
{
  var numbers = GetSmallNumbers();
  var evenNumbers = numbers.Select(n => n * 2);
  Console.WriteLine(evenNumbers.FirstOrDefault());
}
```

Решение

8. В какой момент произойдёт `Exception`?

```
public static IEnumerable<int> GetSmallNumbers()
{
  yield return 1;
  throw new Exception();
  yield return 2;
}
void Main()
{
  var numbers = GetSmallNumbers();
  var evenNumbers = numbers.Select(n => n * 2);
  Console.WriteLine(evenNumbers.FirstOrDefault());
}
```

Решение

3 Математика (Задачи)

1. Что выведет следующий код?

```
Console.WriteLine(
  "| Number | Round | Floor | Ceiling | Truncate | Format |");
foreach (var x in new[] { -2.9, -0.5, 0.3, 1.5, 2.5, 2.9 })
{
  Console.WriteLine(string.Format(CultureInfo.InvariantCulture,
    "| {0,6} | {1,5} | {2,5} | {3,7} | {4,8} | {0,6:N0} |",
    x, Math.Round(x), Math.Floor(x),
    Math.Ceiling(x), Math.Truncate(x)));
}
```

Решение

2. Что выведет следующий код?

```
Action<int,int> print = (a, b) =>
  Console.WriteLine("{0,2} = {1,2} * {2,3}   + {3,3}  ",
                    a, b, a/b, a%b);
Console.WriteLine(" a =  b * (a/b) + (a%b)");
print(7, 3);
print(7, -3);
print(-7, 3);
print(-7, -3);
```

Решение

3. Что выведет следующий код?

```
Console.WriteLine("0.1 + 0.2 {0} 0.3",
  0.1 + 0.2 == 0.3 ? "==" : "!=");
```

Решение

4. Что выведет следующий код?

```
var zero = 0;
try
{
  Console.WriteLine(42 / 0.0);
  Console.WriteLine(42.0 / 0);
  Console.WriteLine(42 / zero);
}
catch (DivideByZeroException)
{
  Console.WriteLine("DivideByZeroException");
}
```

Решение

5. Что выведет следующий код?

```
var maxInt32 = Int32.MaxValue;
var maxDouble = Double.MaxValue;
var maxDecimal = Decimal.MaxValue;
checked
{
  Console.Write("Checked   Int32   increased max: ");
  try { Console.WriteLine(maxInt32 + 42); }
  catch { Console.WriteLine("OverflowException"); }
  Console.Write("Checked   Double  increased max: ");
  try { Console.WriteLine(maxDouble + 42); }
  catch { Console.WriteLine("OverflowException"); }
  Console.Write("Checked   Decimal increased max: ");
  try { Console.WriteLine(maxDecimal + 42); }
  catch { Console.WriteLine("OverflowException"); }
}
unchecked
{
  Console.Write("Unchecked Int32   increased max: ");
  try { Console.WriteLine(maxInt32 + 42); }
  catch { Console.WriteLine("OverflowException"); }
  Console.Write("Unchecked Double  increased max: ");
  try { Console.WriteLine(maxDouble + 42); }
  catch { Console.WriteLine("OverflowException"); }
  Console.Write("Unchecked Decimal increased max: ");
  try { Console.WriteLine(maxDecimal + 42); }
  catch { Console.WriteLine("OverflowException"); }
}
```

Решение

6. Что выведет следующий код?

```
int a = 0;
int Foo()
{
  a = a + 42;
  return 1;
}
void Main()
{
  a += Foo();
  Console.WriteLine(a);
}
```

Решение

4 Значимые типы (Задачи)

1. Что выведет следующий код?

```
struct Foo
{
  int value;
  public override string ToString()
  {
    if (value == 2)
      return "Baz";
    return (value++ == 0) ? "Foo" : "Bar";
  }
}
void Main()
{
  var foo = new Foo();
  Console.WriteLine(foo);
  Console.WriteLine(foo);
  object bar = foo;
  object qux = foo;
  object baz = bar;
  Console.WriteLine(baz);
  Console.WriteLine(bar);
  Console.WriteLine(baz);
  Console.WriteLine(qux);
}
```

Решение

2. Что выведет следующий код?

```
public struct Foo
{
  public int Value;
  public void Change(int newValue)
  {
    Value = newValue;
  }
}
public class Bar
{
  public Foo Foo { get; set; }
}
void Main()
{
  var bar = new Bar { Foo = new Foo() };
  bar.Foo.Change(5);
  Console.WriteLine(bar.Foo.Value);
}
```

Решение

3. Что выведет следующий код?

```
var x = new
  {
    Items = new List<int> { 1, 2, 3 }.GetEnumerator()
  };
while (x.Items.MoveNext())
  Console.WriteLine(x.Items.Current);
```

Решение

4. Что выведет следующий код?

```
public struct Foo
{
  public byte Byte1;
  public int Int1;
}
public struct Bar
{
  public byte Byte1;
  public byte Byte2;
  public byte Byte3;
  public byte Byte4;
  public int Int1;
}
void Main()
{
  Console.WriteLine(Marshal.SizeOf(typeof(Foo)));
  Console.WriteLine(Marshal.SizeOf(typeof(Bar)));
}
```

Решение

5 Строки (Задачи)

1. Что выведет следующий код?

```
Console.WriteLine(1 + 2 + "A");
Console.WriteLine(1 + "A" + 2);
Console.WriteLine("A" + 1 + 2);
```

Решение

2. Что выведет следующий код?

```
Console.WriteLine(1 + 2 + 'A');
Console.WriteLine(1 + 'A' + 2);
Console.WriteLine('A' + 1 + 2);
```

Решение

3. Что выведет следующий код?

```
try
{
  Console.WriteLine(((string)null + null + null) == null);
}
catch (Exception e)
{
  Console.WriteLine(e.GetType());
}
```

Решение

4. Являются ли следующие способы сравнения срок `a` и `b` без учёта регистра эквивалентными:

```
Console.WriteLine(
  string.Compare(a.ToUpper(), b.ToUpper()));
Console.WriteLine(
  string.Compare(a, b, StringComparison.OrdinalIgnoreCase));
```

Решение

5. Допишите следующий код так, чтобы он выводил на консоль содержал в себе строку `Foo`, не переопределяя стандартный метод `Console.WriteLine`?

```
// ???
Console.WriteLine(-42);
```

Решение

6. Рассмотрим следующий код:

```
Thread.CurrentThread.CurrentUICulture =
  CultureInfo.CreateSpecificCulture("ru-RU");
Thread.CurrentThread.CurrentCulture =
  CultureInfo.CreateSpecificCulture("en-US");
var dateTime = new DateTime(2014, 12, 31, 13, 1, 2);
Console.WriteLine(dateTime);
```

В какой локали будет отформатирована дата?

Решение

7. Могут ли две строки совпасть по методу `String.CompareTo`, но различаться по методу `String.Equals`? Как с ними связан оператор `==`?

Решение

8. Допишите следующий код так, чтобы он вывел на консоль строку `"aaaaa"`, не переопределяя стандартный метод `Console.WriteLine`?

```
// ???
Console.WriteLine("Hello");
```

Решение

9. Что выведет следующий код?

```
var x = "AB";
var y = new StringBuilder().Append('A').Append('B').ToString();
var z = string.Intern(y);
Console.WriteLine(x == y);
Console.WriteLine(x == z);
Console.WriteLine((object)x == (object)y);
Console.WriteLine((object)x == (object)z);
```

Решение

10. Допустим, мы пометили сборку атрибутом:

```
[assembly: CompilationRelaxationsAttribute
  (CompilationRelaxations.NoStringInterning)]
```

Значит ли это, что литеральные строки в этой сборке не будут интернироваться?

Решение

11. Если механизм интернирования строк под данную версию CLR отключен, а в коде литеральная строчка `"Hello"` встречается дважды, то сколько раз она будет встречаться в метаданных сборки?

Решение

12. Хранить в обычных строках секретные данные (например, пароль) нельзя, т.к. строка хранится в памяти в открытом виде, злоумышленники могут легко до неё добраться. Какими стандартными средствами можно защитить данные?

Решение

13. Пусть у нас имеется массив различных строк. Может ли метод сортировки от раза к разу давать разные результаты?

Решение

14. Будет ли происходить копирование символьного массива при вызове метода `StringBuilder.ToString()`?

```
var builder = new StringBuilder(10);
for (int i = 0; i < 10; i++)
  builder.Append('a');
var str = builder.ToString();
```

Решение

15. Допустим, у нас имеется `StringBuilder`, хранящий строчку из трёх символов. Возможен ли сценарий, в котором при замене первого символа произойдёт выделение памяти под новый символьный массив?

Решение

16. Может ли количество текстовых элементов (которые можно получить через `TextElementEnumerator`) строки отличаться от количество образующих её `char`-символов?

Решение

17. Как включить в литеральную строку знак обратного слэша без использования управляющей последовательности?

Решение

Решения

6 ООП (Решения)

1. Задача «Oop-OverloadResolutionBasic»

```
params object[]
params object[]
params T[]
params T[]
object, object
```

Итак, у нас имеются методы:

- `void Foo(object a) -> object`
- `void Foo(object a, object b) -> object, object`
- `void Foo(params object[] args) -> params object[]`
- `void Foo<T>(params T[] args) -> params T[]`

Рассмотрим каждый вызов отдельно.

- `Foo() -> params object[]`

Варианты `object` и `object, object` не подходят по количеству аргументов. Вариант `params T[]` не подходит, т.к. невозможно определить тип `T`. Таким образом, правильный вариант — `params object[]`.

- `Foo(null) -> params object[]`

Вариант `object, object` не подходит по количеству аргументов. Вариант `params T[]` не подходит, т.к. невозможно определить тип `T`. Вариант `params object[]` подходит в `expanded` и `unexpanded` формах, в этом случае мы выбираем `expanded`-форму, т.е. сигнатура будет выглядеть как `Foo(object[] args)`. Из двух оставшихся вариантов `object` и `object[]` компилятор выберет более специфичный, т.е. `object[]` (он же `params object[]`).

- `Foo(new Bar()) -> params T[]`

Вариант `object, object` не походит по количеству элементов. Варианты `object` и `params object[]` будут требовать дополнительной конвертации `Bar` в `object`, поэтому вариант `params T[]` (который, по сути, становится `params Bar[]`) более предпочтителен.

- `Foo(new Bar(), new Bar())` —> `params T[]`

Вариант `object` не подходит по количеству элементов. Варианты `object` и `params object[]` будут требовать дополнительной конвертации `Bar` в `object`, поэтому вариант `params T[]` (который, по сути, становится `params Bar[]`) более предпочтителен.

- `Foo(new Bar(), new object())` —> `object, object` Вариант `object` не подходит по количеству элементов. Среди оставшихся вариантов существует ровно одна версия без `params`: `object, object`. Она будет более предпочтительна.

См. также:

- «Overload resolution», «Method invocations», «Applicable function member» в MSDN
- Глава «Overloading» в книге «C# in Depth»

2. Задача «Oop-OverloadResolutionOverride»

```
Bar.Quux(object)
Baz.Quux(params T[])
```

Есть такое правило: если при вызове некоторого метода в «текущем» классе находится подходящая сигнатура, то компилятор не будет даже смотреть на родительские классы. В данной задаче классы `Bar` и `Baz` имеют собственные версии метода `Quux`. Их сигнатуры подходят под передаваемый набор параметров, а значит они и буду вызваны, а перегруженный `Quux` базового класса будет проигнорирован.

См. также:

- Глава «Overloading» в книге «C# in Depth»

3. Задача «Oop-OverloadResolutionInheritance»

```
Bar.Quux(),
Bar.Quux(params object[])
Bar.Quux(),  Baz
```

Ниже представлен схематический порядок вызовов в приведённом коде:

```
Main();
  new Baz(); // Baz.ctor
      base.ctor(); // Bar.ctor
        base.ctor(); // Foo.ctor
```

```
            Quux();
              // Будет вызван Bar.Quux(),
              // т.к. Quux() имеет перегрузку в дочернем классе
              Console.WriteLine("Bar.Quux(),  " + name);
              // в name пока ничего не хранится
          name = "Bar";
      name = "Baz";
      Quux();
        // Будет вызван Bar.Quux(params object[] args),
        // т.к. в классе Bar есть подходящий метод
        Console.WriteLine("Bar.Quux(params object[])");
      ((Foo)this).Quux();
        // Будет вызван Bar.Quux(),
        // т.к. Quux() имеет перегрузку в дочернем классе
        Console.WriteLine("Bar.Quux(),  " + name);
        // name == "Baz"
```

См. также:

- Задача «OverloadResolutionOverride»

4. Задача «Oop-InheritanceNestedClass»

```
Foo.Quux()
```

Класс `Bar.Quux` имеет область видимости `private` и не может быть использован из дочернего класса. Поэтому при вызове метода `Quux` из класса `Baz` будет использован класс `Foo.Quux`.

7 LINQ (Решения)

1. Задача «Linq-EnumerableToArray»

```
GetString: Foo
EnumerableToArray: Foo
GetString: Bar
EnumerableToArray: Bar
GetString: Foo
GetString: Bar
```

LINQ-запросы являются ленивыми, т.е. реализуют отложенное исполнение. Это означает, что если сфомировать запрос и не вызвать для него явно метод вроде ToArray() или ToList(), то выполнение запроса будет отложено до того момента, пока мы явно не затребуем результатов. Таким образом, строка

```
var strings = GetStringEnumerable();
```

не выведет на консоль ничего. Далее, в цикле

```
foreach (var s in strings)
  Console.WriteLine("EnumerableToArray: " + s);
```

произойдёт выполнение запроса. Причём, сначала выполнится первый `yield` (вывод строки `GetString: Foo`), а после него выполнится тело цикла для первого значения перечисления (вывод строки `EnumerableToArray: Foo`). Далее, цикл `foreach` запросит второй элемент перечисления, будет выполнен второй `yield` (вывод строки `GetString: Bar`) и второй раз будет выполнено тело цикла для полученного элемента (вывод строки `EnumerableToArray: Bar`).

Далее следует строка

```
return strings.ToArray();
```

Тут можно наблюдать повторное исполнение LINQ-запроса, а значит мы вновь произойдёт вывод строк `GetString: Foo` и `GetString: Bar`.

2. Задача «Linq-LifeAfterYield»

```
BarBaz
```

Приведённый код превращаются в следующее (часть генерированного кода специальна удалена для лучшего понимания):

```
private sealed class FooEnumerable :
    IEnumerable<string>, IEnumerator<string>
{
  private int state;
  public string Current { get; private set; }

  object IEnumerator.Current
  {
    get { return Current; }
  }

  public FooEnumerable(int state)
  {
    this.state = state;
  }

  public IEnumerator<string> GetEnumerator()
```

```
  {
    FooEnumerable fooEnumerable;
    if (state == -2)
    {
      state = 0;
      fooEnumerable = this;
    }
    else
      fooEnumerable = new FooEnumerable(0);
    return fooEnumerable;
  }

  IEnumerator IEnumerable.GetEnumerator()
  {
    return GetEnumerator();
  }

  bool IEnumerator.MoveNext()
  {
    switch (state)
    {
      case 0:
        Current = "Bar";
        state = 1;
        return true;
      case 1:
        state = -1;
        Console.WriteLine("Baz");
        break;
    }
    return false;
  }

  void IEnumerator.Reset()
  {
    throw new NotSupportedException();
  }

  void IDisposable.Dispose()
  {
  }
}

IEnumerable<string> Foo()
{
  return new FooEnumerable(-2);
}
```

```
void Main()
{
  var enumerator = Foo().GetEnumerator();
  while (enumerator.MoveNext())
    Console.Write(enumerator.Current);
}
```

3. Задача «Linq-ClosureAndForeach»

   C# 1.0 — C# 4.0: `3 3 3`

   C# 5.0+: `1 2 3`

   В версиях C# 1.0 — C# 4.0 приведённый код превращался в следующую конструкцию:

```
public void Run()
{
  var actions = new List<Action>();
  DisplayClass c1 = new DisplayClass();
  foreach (int i in Enumerable.Range(1, 3))
  {
    c1.i = i;
    list.Add(c1.Action);
  }
  foreach (Action action in list)
    action();
}

private sealed class DisplayClass
{
  public int i;

  public void Action()
  {
    Console.WriteLine(i);
  }
}
```

   Таким образом, все три элемента списка на самом деле являются одним и тем же делегатом, поэтому в консоли мы увидим три одинаковых значения, равных последнему значению `i`.

   В C# 5.0+ произошли изменения, новый вариант кода:

```
public void Run()
{
  var actions = new List<Action>();
  foreach (int i in Enumerable.Range(1, 3))
  {
```

```
    DisplayClass c1 = new DisplayClass();
    c1.i = i;
    list.Add(c1.Action);
  }
  foreach (Action action in list)
    action();
}

private sealed class DisplayClass
{
  public int i;

  public void Action()
  {
    Console.WriteLine(i);
  }
}
```

Теперь каждый элемент списка ссылается на собственный делегат, так что все полученные значения будут разными.

4. Задача «Linq-QueryAfterRemove»

```
1
```

При вызове `list.Where(c => c.StartsWith("B"))` запрос будет только построен, но не выполнен. Реальное выполнение начнётся в момент вызов `query.Count()`. К этому времени значение `list` будет `{ "Foo", "Baz" }`, а значит, будет найден только один элемент, начинающийся с буквы `'B'`.

5. Задача «Linq-ClosureAndVariable»

```
2
```

Приведённый код примет следующий вид:

```
class DisplayClass
{
  public string startLetter;

  public bool Method1(string c)
  {
    return c.StartsWith(this.startLetter);
  }

  public bool Method2(string c)
  {
```

```
    return c.StartsWith(this.startLetter);
  }
}

void Main()
{
  DisplayClass displayClass = new DisplayClass();
  var list1 = new List<string> { "Foo", "Bar", "Baz" };
  var list2 = list1;
  displayClass.startLetter = "F";
  IEnumerable<string> source = list2.Where(displayClass.Method1);
  displayClass.startLetter = "B";
  Console.WriteLine(source.Where(displayClass.Method2).Count());
}
```

Выполнение запроса начнётся только в самой последней строчке кода. Как можно видеть, для обоих замыканий создался один и тот же вспомогательный класс. Сначала выполнится первый запрос `list2.Where(displayClass.Method1)` и вернёт `{ "Bar", "Baz" }`, т.к. `displayClass.startLetter` к моменту исполнения равна `"B"`. Далее выполнится запрос `source.Where(displayClass.Method2)`, который также вернёт `{ "Bar", "Baz" }`. Количество элементов в результате равно двум.

6. Задача «Linq-QueryWithInc»

```
Inc: 0
Inc: 1
Number: 2
Inc: 2
Inc: 3
Number: 4
```

Императивная версия кода выглядит следующим образом:

```
var numbers = new[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var takenAmount = 0;
for (int i = 0; i < numbers.Length; i++)
{
  var number = numbers[i];
  Console.WriteLine("Inc: " + number);
  var number2 = number + 1;
  if (number2 % 2 == 0)
  {
    Console.WriteLine("Number: " + number2);
    takenAmount++;
    if (takenAmount == 2)
      break;
```

```
    }
}
```

7. Задача «Linq-ExceptionYieldYield»

   При вызове `evenNumbers.FirstOrDefault()`. Строчки

```
var numbers = GetSmallNumbers();
var evenNumbers = numbers.Select(n => n * 2);
```

   только строят запрос, но не исполняют его. Логика `GetSmallNumbers()` начнёт исполняться при первом вызове метода `MoveNext()`, который соответствует вызову `evenNumbers.FirstOrDefault()`. В этот момент и произойдёт `Exception`.

8. Задача «Linq-YieldExceptionYield»

   Exception не произойдёт. Действительно, строка

```
var numbers = GetSmallNumbers();
```

   только строит запрос, но не выполняет его. Строка

```
var evenNumbers = numbers.Select(n => n * 2);
```

   также строит ещё один запрос без непосредственного выполнения. Отдельный интерес представляет последняя строка метода `Main`:

```
Console.WriteLine(evenNumbers.FirstOrDefault());
```

   Данный вызов оценит получит только первый элемент запроса (одиночные вызовы `MoveNext()` и `Current`), дальнейшее получение элементов перечисления происходить не будет. Таким образом, код отработает без исключений.

8 Математика (Решения)

1. Задача «Math-Rounding1»

```
| Number | Round | Floor | Ceiling | Truncate | Format |
|   -2.9 |    -3 |    -3 |      -2 |       -2 |     -3 |
|   -0.5 |     0 |    -1 |       0 |        0 |     -1 |
|    0.3 |     0 |     0 |       1 |        0 |      0 |
|    1.5 |     2 |     1 |       2 |        1 |      2 |
|    2.5 |     2 |     2 |       3 |        2 |      3 |
|    2.9 |     3 |     2 |       3 |        2 |      3 |
```

Если число находится ровно посередине между двумя возможными вариантами, то работают следующие правила:

Math.Round по умолчанию округляет к ближайшему чётному целому.

Math.Floor округляет вниз по направлению к отрицательной бесконечности.

Math.Ceiling округляет вверх по направлению к положительной бесконечности.

Math.Truncate округляет вниз или вверх по направлению к нулю.

String.Format округляет к числу, которое дальше от нуля (см. также Standard Numeric Format Strings, Custom Numeric Format Strings).

2. Задача «Math-Rounding2»

```
 a =  b * (a/b) + (a%b)
 7 =  3 *   2   +   1
 7 = -3 *  -2   +   1
-7 =  3 *  -2   +  -1
-7 = -3 *   2   +  -1
```

При целочисленном делении результат всегда округляется по направлению к нулю. При взятии остатка от деления должно выполняться следующее правило: `x mod y = x - (x / y) * y`.

3. Задача «Math-Eps»

```
0.1 + 0.2 != 0.3
```

Числа типа `Double` представляются согласно спецификации IEEE 754 и хранятся в 64-разрядном бинарном формате. К сожалению, многие десятичные нецелые числа невозможно представить в таком формате, что часто сказывается при выполнении арифметических операций над числами с плавающей запятой:

```
// Displays '5.5511151231257827E-17'
Console.WriteLine("{0:R}", 0.1+0.2-0.3);
```

4. Задача «Math-DivideByZero»

```
Infinity
Infinity
DivideByZeroException
```

Первые две строчки выполнятся и выведут `Infinity`. При деление произойдёт конвертация `int` к `double`, а операция `double operator /(double x, double y)` выполняется согласно IEEE 754 (ECMA-334, 14.7.2), а значит при делении положительного числа на положительный ноль должна вернуть положительную бесконечность. Операция `int operator /(int x, int y)` бросает `DivideByZeroException` в случае, если правый операнд равен нулю (ECMA-334, 14.7.2). Поэтому третья операция деление выбросит исключение, о чём будет выведено соответствующее сообщение.

5. Задача «Math-Overflow»

```
Checked   Int32   increased max: OverflowException
Checked   Double  increased max: 1,79769313486232E+308
Checked   Decimal increased max: OverflowException
Unchecked Int32   increased max: -2147483607
Unchecked Double  increased max: 1,79769313486232E+308
Unchecked Decimal increased max: OverflowException
```

Операции с переполнением `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char` выбрасывают исключение `OverflowException` в зависимости от `checked`/`unchecked`-контекста (ECMA-334, 11.1.5).

Операции с переполнением `float`, `double` не выбрасывают исключение `OverflowException` (ECMA-334, 11.1.6).

Операции с переполнением `decimal` всегда выбрасывают исключение `OverflowException` (ECMA-334, 11.1.7).

6. Задача «Math-AugmentedAssignment»

```
1
```

Конструкция

```
a += Foo();
```

развернётся в

```
a = a + Foo();
```

Сначала оценится левый операнд `a`, равный нулю. Затем оценится правый операнд, который вернёт `1`. В итоге в `a` запишется значение `1`, не смотря на то, что внутри метода `Foo` произошло переприсвоение поля `a`.

9 Значимые типы (Решения)

1. Задача «ValueTypes-Boxing»

```
Foo
Foo
Foo
Bar
Baz
Foo
```

Ключевым моментом в понимании примера является тот факт, что при вызове метода `ToString()` (который автоматически вызывается при вызове `Console.WriteLine`) для структуры `Foo` структура упаковывается. А это означает, что в управляемой куче создаётся копия структуры, метод `ToString()` отрабатывает для упакованной копии. Теперь разберём пример по строчкам:

```
var foo = new Foo();
Console.WriteLine(foo); // Displays "Foo" (value == 0)
Console.WriteLine(foo); // Displays "Foo" (value == 0)
```

Мы создали экземпляр структуры `Foo` и дважды выполнили для него `Console.WriteLine(foo)`. Дважды выполнилась копирование и упаковка структуры, `ToString()` вызвалось для копии, не тронув оригинал. Изначально `Foo.value == 0`, так что в обоих случаях выведется `"Foo"`.

```
object bar = foo;
object qux = foo;
object baz = bar;
```

Тут мы явно выполняем упаковку структуры. Объекты `bar` и `qux` указывают на разные копии структуры, т.к. мы выполнили две отдельных операции упаковки. Объект `baz` указывает на ту же копию структуры, что и `bar`, т.к. в третьей строчке мы просто выполнили копирование ссылки.

```
Console.WriteLine(baz); // Displays "Foo" (value == 0)
Console.WriteLine(bar); // Displays "Bar" (value == 1)
Console.WriteLine(baz); // Displays "Baz" (value == 2)
```

В этих трёх строчках мы работаем с одной и той же копией структуры, обращаясь к ней по ссылке (`bar` и `baz` представляют собой одну и ту же ссылку).

```
Console.WriteLine(qux); // Displays "Foo" (value == 0)
```

А в этой строчке мы работаем с копией структуры, для которой метод `ToString()` ещё ни разу не вызывался. Поэтому выведется `"Foo"`.

2. Задача «ValueTypes-MutableProperty»

```
0
```

Структуры, как мы знаем, копируются по значению. При обращении к свойству `bar.Foo` вызовется метод `bar.get_Foo()`, который вернёт нам копию структуры, для которой мы выполним метод `Change`. При этом оригинальная структура останется без изменений. В последней строчке метода `Main` мы вновь обращаемся к методу `bar.get_Foo()` и берём значение `Value` для новой копии `Foo`, равное нулю.

3. Задача «ValueTypes-Enumerator»

   Цикл зависнет, будут выводиться нули. Для понимания этого факта необходимо вспомнить, что метод `GetEnumerator` для `List<T>` возвращает структуру. А значит, при каждом обращении к методу `x.Items.MoveNext()` мы будем работать не с оригинальным енумератором, а с его копией, не меняя при этом внутреннее состояние исходного енумератора (а именно, его текущий элемент `x.Items.Current`). Таким образом, в условии цикла ничего полезного не происходит, текущий элемент на веки останется нулём.

4. Задача «ValueTypestructLayout»

```
8
8
```

   Если явно не задано, то CLR автоматически управляет размещением данных в структуре. В данном случае происходит выравнивание полей по границе 4 байт, в результате чего общий размер структуры составит 8 байт. Пользователь может явно управлять выравниванием. Например, если пометить `Foo` атрибутом `StructLayout` следующим образом:

```
[StructLayout(LayoutKind.Sequential, Pack=1)]
public struct Foo
```

   то мы заставим CLR размещать поля последовательно, выравнивания их по границе 1 байта, в результате чего получим:

```
Console.WriteLine(Marshal.SizeOf(typeof(Foo))); // Displays '5'
```

10 Строки (Решения)

1. Задача «StringslusString»

```
3A
1A2
A12
```

   Оператор + является лево-ассоциативным. Это означает, что в первом случае выполнится сначала целочисленное сложение, а затем сложение числа и строки, которое будет сопровождаться конвертированием числа в строку. Во втором и третьем случае результат первого сложения будет строкой.

```
1 + 2 + "A" = ((1 + 2) + "A") = 3 + "A"  = "3A"
1 + "A" + 2 = ((1 + "A") + 2) = "1A" + 2 = "1A2"
"A" + 1 + 2 = (("A" + 1) + 2) = "A1" + 2 = "A12"
```

2. Задача «StringslusChar»

```
68
68
68
```

3. Задача «StringstringPlusNull»

```
False
```

Фрагмент из C# Language Specification, раздел 7.8.4 Addition operator:

«String concatenation:

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

These overloads of the binary + operator perform string concatenation. **If an operand of string concatenation is null, an empty string is substituted**. Otherwise, any non-string argument is converted to its string representation by invoking the virtual ToString method inherited from type object. If ToString returns null, an empty string is substituted.»

Другими словами, при строковых операциях `null` превращается в пустую строку. Таким образом:

```
((string)null + null + null) == ("" + null + null) ==
  (("" + null) + null) == (("" + "") + null) == ("" + null) ==
  ("" + "") == ""
"" != null
```

Ссылки на исходный код:

- Microsoft Reference Source
- Mono 3.10.0 Source

4. Задача «Strings-CaseInComparison»

Нет, в некоторых локалях результат будет разный. Пример:

```
var a = "i";
var b = "I";
Thread.CurrentThread.CurrentCulture =
  CultureInfo.CreateSpecificCulture("tr-TR");
Console.WriteLine(
  string.Compare(a.ToUpper(), b)); //'1'
Console.WriteLine(
  string.Compare(a, b, StringComparison.OrdinalIgnoreCase)); //'0'
```

См. The Turkey Test.

5. Задача «Strings-CorruptedNumber»

   Задание можно выполнить с использованием пользовательской локали:

```
var culture = (CultureInfo) CultureInfo.InvariantCulture.Clone();
culture.NumberFormat.NegativeSign = "Foo";
Thread.CurrentThread.CurrentCulture = culture;
Console.WriteLine(-42); // Displays "Foo42"
```

6. Задача «Strings-CurrentCulture»

   Для форматирования даты будет использоваться `CurrentCulture`. Таким образом, дата будет выведена с использованием `en-US` локали:

```
12/31/2014 1:01:02 PM
```

   вместо локали `ru-RU`. В русской локали мы бы получили:

```
31.12.2014 13:01:02
```

7. Задача «Strings-CompareToVsEquals»

   Да, т.к. по умолчанию `CompareTo` выполняется с учётом региональных стандартов, а `Equals` — без.

   Пример: строки "ß"(Эсцет) и "ss"совпадут по `CompareTo` в немецкой локали, несмотря на то, что они разной длины.

   Выполнение оператора == совпадает с поведением метода `String.Equals`.

8. Задача «Strings-CorruptedString»

   Можно заинтернировать строку `"Hello"`, а затем через unsafe-код добраться до соответствующего участка памяти и изменить целевое значение:

```
var s = "Hello";
string.Intern(s);
unsafe
{
  fixed (char* c = s)
    for (int i = 0; i < s.Length; i++)
      c[i] = 'a';
}
Console.WriteLine("Hello"); // Displays: "aaaaa"
```

9. Задача «Strings-ExplicitlyInternment»

```
True
True
False
True
```

Первые две строчки содержат `True`, т.к. для выражений `x == y` и `x == z` будет вызвана перегрузка оператора `==` для строк, а строки совпадают. В последних двух строчках будет вызвана перегрузка оператора `==` для объектов, которая сравнит объекты по ссылке. Строка `"AB"` является заинтернированной и хранится в специальной таблице памяти. Переменная `x` указывает на интернированное значение, т.к. определена с помощью литерала. Переменная `z` указывает на это же значение, т.к. определена с помощью метода `string.Intern`. А переменная `y` будет указывать на другую область памяти, т.к. была определена через `StringBuilder`, который при вызове метода `ToString()` не учитывает интернированные значения.

10. Задача «Strings-NoStringInterning»

    Нет. Документация гласит, что в этом `NoStringInterning` лишь помечает сборку, как не требующую интернирования. Поэтому CLR только может не интернировать строки, но не обязана. Кроме того, мы всегда может заинтернировать строку через метод `String.Intern`.

11. Задача «Strings-InternmentAndMetadata»

    Один. Интернирование строк происходит во время выполнения программы и не имеет отношения к метаданным.

12. Задача «Stringsecure»

    Нужно хранить строчку с использованием класса `SecureString`.

13. Задача «StringstableSorting»

    Да, стандартная сортировка в .NET является неустойчивой, а значит, если мы, скажем, сортируем строки без учёта регистра, то строки `"AAA"` и `"aaa"` могут расположиться в любом порядке.

14. Задача «StringstringBuilderToString»

    CLR 2.0: Нет, в новая строка будет ссылаться на тот же символьный массив, что и исходный `StringBuilder`.

    CLR 4.0: Да, в этой версии платформы массив всегда копируется.

15. Задача «StringstringBuilderMemory»

    Да, если предыдущим методом был `StringBuilder.ToString()`, а используемая версия CLR меньше 4.0.

16. Задача «Strings-TextElementEnumerator»

    Да, т.к. FCL поддерживает кодировки, использующие больше 16-ти разрядов: один текстовый элемент может определяться несколькими `char`-символами.

17. Задача «Strings-EscapeSlash»

    Нужно использовать verbatim string: `@"\"`.

Библиография

Некоторые хорошие книжки, которые помогут улучшить ваши познания в платформе .NET:

1. Standard ECMA-335: Common Language Infrastructure (CLI)
2. Standard ECMA-334: C# Language Specification
3. «CLR via C#» by Jeffrey Richter
4. «C# in Depth» by Jon Skeet
5. «Pro C# 2010 and the .NET 4 Platform» by Andrew Troelsen
6. «C# 4.0: The Complete Reference» by Herbert Schildt
7. «C# Unleashed» by Joseph Mayo
8. «Essential C# 5.0» by Mark Michaelis, Eric Lippert
9. «Pro .Net Performance» by Sasha Goldshtein, Dima Zurbalev, Ido Flatow
10. «Under the Hood of .NET Memory Management» by Chris Farrell, Nick Harrison
11. «Effective C#: 50 Specific Ways to Improve Your C#» by Bill Wagner
12. «Expert .NET 2.0 IL Assembler» by by Serge Lidin
13. «Advanced .NET Debugging» by Mario Hewardt, Patrick Dessud
14. «Essential .Net Volume 1: The Common Language Runtime» by Don Box