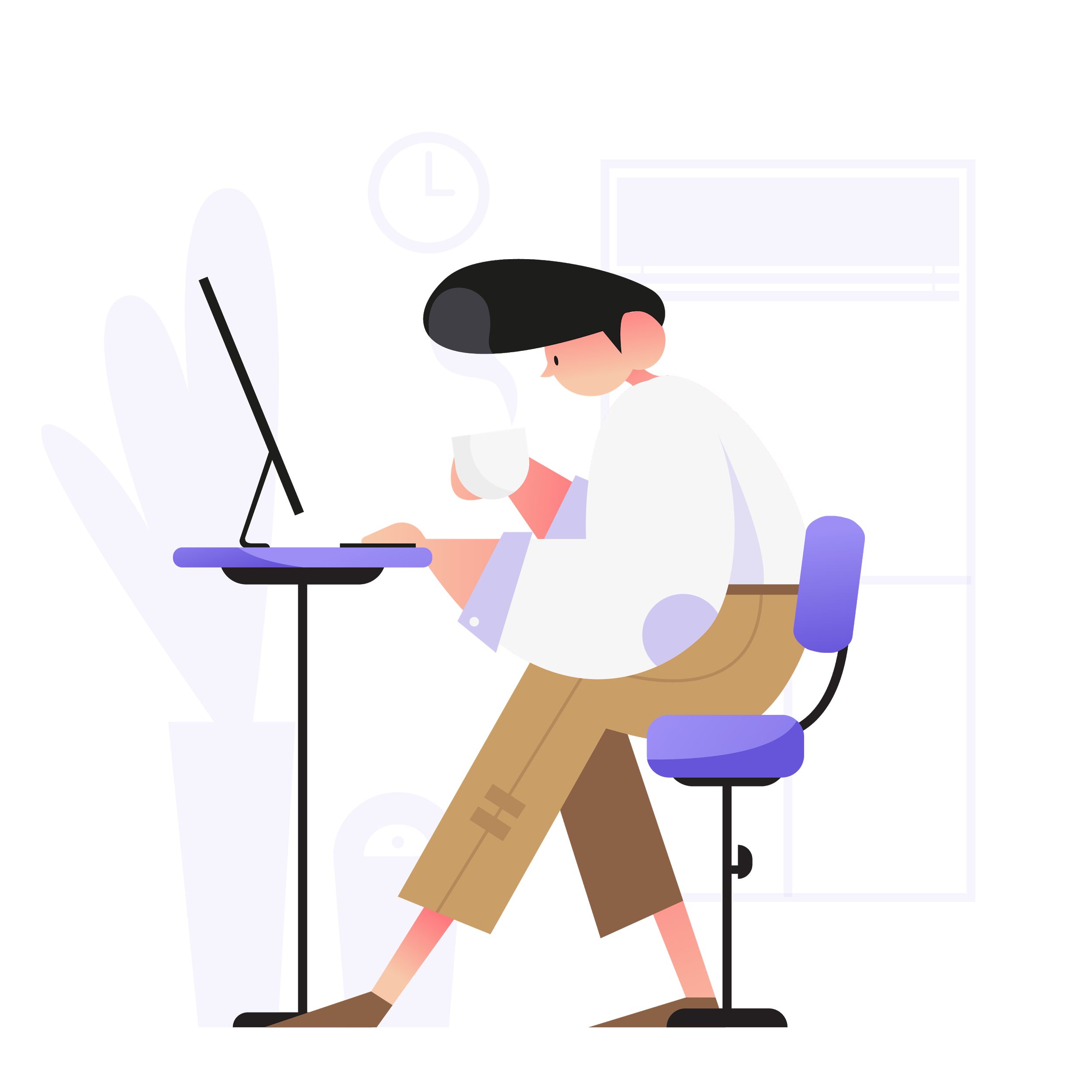
Алгоритмы и структуры данных на С#

Класс, структура и дистанция

Netcore 3.1



**На этом уроке**

1. Разберём понятия Struct и Class.
2. Познакомимся со ссылочным и значимым типами.
3. Узнаем, от чего зависит выбор ссылочного или значимого типа.
4. Узнаем, что такое Boxing.
5. Освоим методы вычисления дистанции между точками.
6. Научимся подсчитывать время выполнения.

**Оглавление**

[Введение](#_lbtu0vk2ba8h)

[Struct и Class](#_p5gikuffpqid)

[Что это](#_wz371bvslc14)

[Ссылочный тип](#_395h8pud2c5n)

[Значимый тип](#_dypjqr9uqsb2)

[Когда это нужно](#_ctfh9mcyg7l7)

[Boxing](#_pj3pu8mb9rui)

[Дистанция](#_8p26d8v13u6b)

[Методы расчёта](#_slbti4nvr524)

[Подсчёт времени выполнения](#_90h2otcqenj8)

[Заключение](#_gn2pj9fbechk)

[Практическое задание](#_uvngr3pebly3)

[Дополнительные материалы](#_2jxsxqh)

[Используемые источники](#_7c9jrnh1eqkx)

# 

# Введение

Класс и структура — одни из ключевых вещей в C#. Понимание, как они устроены в памяти, важно для корректной реализации алгоритмов и структур данных. Иногда алгоритм или задачу можно реализовать разными способами. Мы рассмотрим, на что это может влиять, на примере расчёта дистанции.

# Struct и Class

## Что это

Ключевые слова structи classв C# на первый взгляд означают одно и то же — описание некоторой сущности, например точки в пространстве. На самом деле между этими понятиями есть большое различие.

Пример класса и структуры:

|  |
| --- |
| public class PointClass {  public int X;  public int Y; }  public struct PointStruct {  public int X;  public int Y; } |

В структуре нельзя наследоваться от класса, но можно реализовывать интерфейсы.

Главное различие состоит в том, как происходит обращение к структуре и классу и как они хранятся. Структура — это значимый тип (value type), а класс — это ссылочный тип (reference type).

К значимым типам относятся:

* структуры;
* byte;
* short/ushort;
* int/uint;
* long/ulong;
* float;
* double.

К ссылочным типам относятся:

* классы;
* массивы;
* string;
* Object.

## 

## Ссылочный тип

Ссылочный тип означает, что при создании нового экземпляра этого типа в переменной, поле, свойстве или элементе массива мы получаем ссылку на созданный элемент, который располагается в куче (heap). В примере ниже создаётся объект типа object, ссылка на который кладётся в переменную x.

|  |
| --- |
| static void Main(string[] args) {  object x = new object(); } |

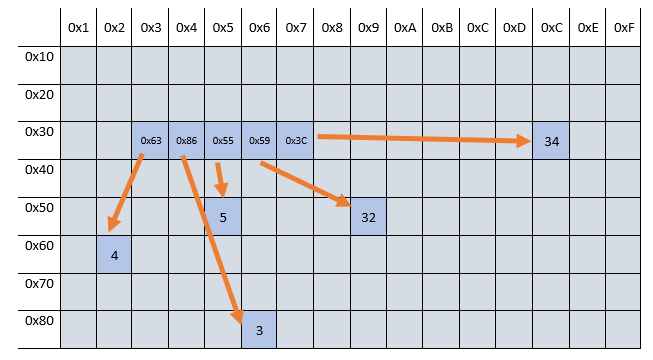
Одна из особенностей ссылочного типа — при передаче в метод или в другую переменную мы также оперируем ссылками, то есть передаётся не сам объект, а только ссылка на него. Если мы изменим где-то этот объект, то все участки кода, которые содержат эту ссылку, будут видеть изменения. Пример ниже иллюстрирует эту особенность:

|  |
| --- |
| static void Main(string[] args) {  var point = new PointClass() { X = 42, Y = 42 };  var array = new PointClass[1];  array[0] = point;   PrintPoint(point, "1. point local variable");  PrintPoint(array[0], "2. point in array");   ChangePoint(point);   PrintPoint(point, "3. point local variable");  PrintPoint(array[0], "4. point in array");   point = new PointClass() { X = 7, Y = 7 };   PrintPoint(point, "5. point local variable");  PrintPoint(array[0], "6. point in array"); }  public static void ChangePoint(PointClass pointClass) {  pointClass.X = 13;  pointClass.Y = 13; }  public static void PrintPoint(PointClass pointClass, string tag) {  Console.WriteLine($"{tag}\t X:{pointClass.X}, Y:{pointClass.Y}"); } |

Вывод работы кода будет такой:

|  |
| --- |
| 1. point local variable X:42, Y:42 2. point in array X:42, Y:42 3. point local variable X:13, Y:13 4. point in array X:13, Y:13 5. point local variable X:7, Y:7 6. point in array X:13, Y:13 |

Ещё, как пример, можно показать, как будет выглядеть массив ссылочных типов в памяти:



## 

## Значимый тип

Значимый тип ссылками не оперирует, а сразу хранится в той структуре или переменной, где он создан. Например, при передаче в функцию в качестве аргумента значимый тип будет передаваться не по ссылке и сразу целиком. Фактически у функции будет копия переданного значения, и если изменить значение аргумента, то вызывающий код не увидит эти изменения. Значимый тип можно передать в функцию по ссылке, если использовать слова ref или out.

Переменная метода находится на стеке. Если она значимого типа, то все значения переменной тоже будут на стеке. Если же она ссылочного типа, то на стеке будет только ссылка, а сам экземпляр ссылочного типа будет в heap. Мы не будем разбирать в этом курсе, что такое heap и стек вызова в среде .NET, но для справки стоит уточнить, что heap и стек не разные типы памяти. Они оба находятся в оперативной памяти в контексте одного процесса, но у них разное предназначение и методы работы с ними различаются.

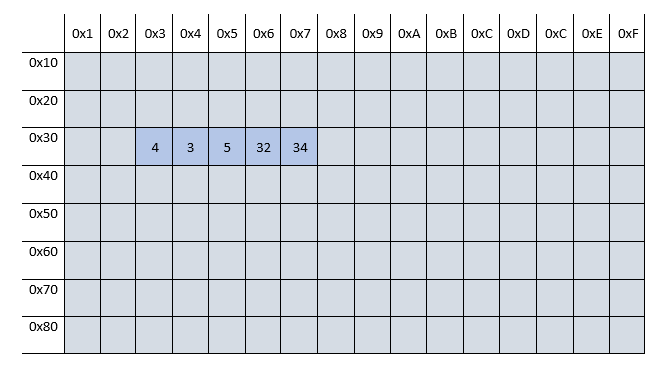
Аналогичный код из примера со ссылочным типом, но теперь уже со структурой:

|  |
| --- |
| static void Main(string[] args) {  var point = new PointStruct { X = 42, Y = 42 };  var array = new PointStruct[1];   array[0] = point;  array[0].Y = 33;   PrintPoint(point, "1. point local variable");  PrintPoint(array[0], "2. point in array");   ChangePoint(point);   PrintPoint(point, "3. point local variable");  PrintPoint(array[0], "4. point in array");   point = new PointStruct() { X = 7, Y = 7 };   PrintPoint(point, "5. point local variable");  PrintPoint(array[0], "6. point in array"); }  public static void ChangePoint(PointStruct pointClass) {  pointClass.X = 13;  pointClass.Y = 13; }  public static void PrintPoint(PointStruct pointClass, string tag) {  Console.WriteLine($"{tag}\t X:{pointClass.X}, Y:{pointClass.Y}"); } |

Вывод работы кода будет такой:

|  |
| --- |
| 1. point local variable X:42, Y:42 2. point in array X:42, Y:33 3. point local variable X:42, Y:42 4. point in array X:42, Y:33 5. point local variable X:7, Y:7 6. point in array X:42, Y:33 |

В качестве примера можно показать, как будет выглядеть массив значимых типов в памяти:



## Когда это нужно

Выбор ссылочного или значимого типа зависит от конкретной задачи. Когда требуется оперировать большим объектом, например массивом на 1 000 000 элементов, лучше его передавать по ссылке, чтобы не копировать везде значение. Когда же требуется хранить массив однородных данных и часто его обрабатывать, может пригодиться значимый тип внутри массива.

Также использование значимого типа помогает уменьшить трафик памяти и нагрузку на GC (Garbage Collector). При одинаковых полях класса и структуры экземпляр класса всегда будет занимать больше места в памяти из-за ссылки на него и хранения дополнительной информации в каждом объекте: ссылки на тип и блока синхронизации.

## Boxing

Boxing — одна из особенностей среды .NET, которая неявно может упаковывать значимые типы в ссылочные. Например, когда вы приводите значимый тип к интерфейсу или типу object. Эта особенность может показаться безобидной, но если разобраться, то можно обнаружить очень значимые её последствия, когда речь идёт не о единичных операциях. Обратная boxing операция называется unboxing.

При упаковке создаётся копия элемента в heap и возвращается ссылка на него. Само по себе создание экземпляра ссылочного типа включает в себя большой набор операций: выделение памяти, поиск подходящего места в куче, создание объекта.

Кроме того, что это увеличивает потребление и трафик памяти, также возрастает нагрузка на GC, потому что сборщику мусора потребуется убрать объект ссылочного типа, когда он будет не нужен. Если мы говорим о единичной операции, это не так страшно, но когда речь пойдёт о больших объёмах данных, например обработке большого массива, boxing может заметно замедлить работу приложения.

|  |
| --- |
| static void Main(string[] args) {  int x = 42;  object ox = x; *// boxing*  int y = (int)ox; *// unboxing*   Console.WriteLine($"x : {x}, ox : {ox}, y : {y}"); } |

# Дистанция

## Методы расчёта

С точки зрения математики, ничего не стоит вычислить расстояние между двумя точками, если известны их координаты. Но как только мы оказываемся на машинном уровне, может обнаружиться множество особенностей: от точности вычислений до их производительности. На примере расчёта расстояния между точками мы посмотрим, как одна и та же задача может решаться очень похожими, но разными способами и какие особенности у них могут быть.

В первую очередь стоит ответить на вопрос, нужен ли нам весь результат вычисления расстояния между точками или будет достаточно его части. Например, если нам требуется сравнить два расстояния, мы можем не вычислять корень числа на последнем шаге.

Пример расчёта дистанции:

|  |
| --- |
| public static float PointDistance(PointStruct pointOne, PointStruct pointTwo) {  float x = pointOne.X - pointTwo.X;  float y = pointOne.Y - pointTwo.Y;  return MathF.Sqrt((x \* x) + (y \* y)); } |

Тот же пример, но уже без вычисления квадратного корня в конце:

|  |
| --- |
| public static float PointDistanceShort(PointStruct pointOne, PointStruct pointTwo) {  float x = pointOne.X - pointTwo.X;  float y = pointOne.Y - pointTwo.Y;  return (x \* x) + (y \* y); } |

Другой вопрос — насколько точные значения нам требуются. В числах с плавающей запятой есть типы float и double: float занимает 4 байта в памяти, а double — 8 байт. Double обладает удвоенной точностью, в отличие от float, но требует больше памяти. При единичных операциях можно не заметить разницы (особенно в х64-системах), но когда начинается обработка больших массивов, процессор может использовать векторные операции, где за раз может обработать несколько идущих подряд чисел. Размер регистра, в который эти числа помещаются, ограничен, поэтому чем меньше тип данных, тем больше его значений поместится в этот регистр. Аналогично может быть и с кешированием: меньше размер типа — меньше нужно будет переносить данных из оперативной памяти в кеш.

Также стоит отметить, что есть тип decimal, который занимает 16 байт в памяти, и BigInteger, который имеет произвольную точность и может быть использован в финансовых расчётах или в точных научных.

Пример одного и того же кода, но с разными типами данных:

|  |
| --- |
| public static double PointDistanceDouble(PointStruct pointOne, PointStruct pointTwo) {  double x = pointOne.X - pointTwo.X;  double y = pointOne.Y - pointTwo.Y;  return Math.Sqrt((x \* x) + (y \* y)); }  public static float PointDistance(PointStruct pointOne, PointStruct pointTwo) {  float x = pointOne.X - pointTwo.X;  float y = pointOne.Y - pointTwo.Y;  return MathF.Sqrt((x \* x) + (y \* y)); } |

Вместо точного вычисления какой-то функции можно обойтись её приближёнными значениями, использовав упрощённый (с точки зрения компьютера) алгоритм вычисления. Например, за счёт различных хаков и особенностей машинной математики и просто математики. Пример ниже показывает, как можно, не используя Math.Sqrt,вычислить квадратный корень:

|  |
| --- |
| [StructLayout(LayoutKind.Explicit, Pack = 1)] public struct FloatIntUnion {  [FieldOffset(0)]  public int i;   [FieldOffset(0)]  public float f; }   public static float fsrt(float z) {  if (z == 0) return 0;  FloatIntUnion u;  u.i = 0;  u.f = z;  u.i -= 1 << 23; */\* Subtract 2^m. \*/*  u.i >>= 1; */\* Divide by 2. \*/*  u.i += 1 << 29; */\* Add ((b + 1) / 2) \* 2^m. \*/*  return u.f; } |

# Подсчёт времени выполнения

Иногда одного знания асимптотической сложности алгоритма недостаточно. Могут быть неочевидные с первого взгляда участки кода, которые способны подпортить производительность алгоритма, например частый boxing и unboxing. Или просто бывает сложно прикинуть, насколько быстро выполняется определённый участок кода или какой из методов быстрее.

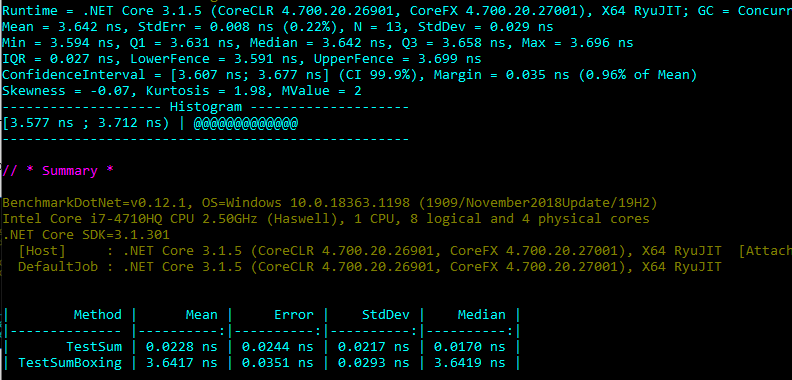
Можно измерить реальное время выполнения. Самый простой способ — взять время перед вызовом метода и после, вычесть одно из другого. Так мы получим время выполнения, но этот способ может дать обманчивую информацию, так как один и тот же метод может выполняться за разное время при каждом вызове. Это может происходить за счёт кеширования или из-за того, что операционная система в какой-то раз решила дать коду чуть больше процессорного времени.

Чтобы уменьшить влияние этих факторов на финальный результат замера, можно выполнить несколько раз этот участок кода и взять среднее время. Чтобы не писать такой код самостоятельно каждый раз под такую задачу, существует библиотека BenchmarkDotNet. Ниже показан пример работы с ней.

Перед запуском кода требуется установить NuGet-пакет BenchmarkDotNet и запускать код из release-сборки. Во время запуска потребуется выбрать, какие именно тесты запускать, после выбора — просто ждать результата.

|  |
| --- |
| using BenchmarkDotNet.Attributes; using BenchmarkDotNet.Running;  namespace GeekBrainsAlgos {  class Program  {  static void Main(string[] args)  {  BenchmarkSwitcher.FromAssembly(typeof(Program).Assembly).Run(args);  }  }   public class BechmarkClass  {  public int SumValueType(int value)  {  return 9 + value;  }   public int SumRefType(object value)  {  return 9 + (int)value;  }   [Benchmark]  public void TestSum()  {  SumValueType(99);  }   [Benchmark]  public void TestSumBoxing()  {  object x = 99;  SumRefType(x);  }  } } |

После небольшого ожидания можно получить вот такой результат. Числа могут различаться в зависимости от машины.



# Заключение

Мы узнали, что структура и класс различаются по методу хранения в памяти: класс хранится по ссылке, структура — по значению. Также изучили метод вычисления дистанции и поняли, как можно решить одну и ту же задачу в зависимости от требований. Подсчитать время выполнения кода можно не руками, а специализированной библиотекой.

# 

# Практическое задание

**Протестируйте разные методы расчёта дистанции**

Напишите тесты производительности для расчёта дистанции между точками с помощью BenchmarkDotNet. Рекомендуем сгенерировать заранее массив данных, чтобы расчёт шёл с различными значениями, но сам код генерации должен происходить вне участка кода, время которого будет тестироваться.

Для каких методов потребуется написать тест:

1. Обычный метод расчёта дистанции со ссылочным типом (PointClass — координаты типа float).
2. Обычный метод расчёта дистанции со значимым типом (PointStruct — координаты типа float).
3. Обычный метод расчёта дистанции со значимым типом (PointStruct — координаты типа double).
4. Метод расчёта дистанции без квадратного корня со значимым типом (PointStruct — координаты типа float).

Результаты можно оформить в виде списка или таблицы, в которой наглядно можно будет увидеть время выполнения того или иного метода.

# Дополнительные материалы

1. [Быстрый обратный квадратный корень](https://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D1%8B%D0%B9_%D0%BE%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D1%8B%D0%B9_%D0%BA%D0%B2%D0%B0%D0%B4%D1%80%D0%B0%D1%82%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D1%80%D0%B5%D0%BD%D1%8C).
2. [Heap, stack C#](https://www.c-sharpcorner.com/article/C-Sharp-heaping-vs-stacking-in-net-part-i/) .
3. [BenchmarkDotNet](https://github.com/dotnet/BenchmarkDotNet).

# Используемые источники

1. Р. Стивенс. Алгоритмы. Теория и практическое применение. М.: Издательство «Э», 2016.
2. Н. Вирт. Алгоритмы и структуры данных. Новая версия для Оберона. М.: ДМК-пресс, 2010.
3. Д. Рихтер CLR via C# М.: Питер, 2019.