



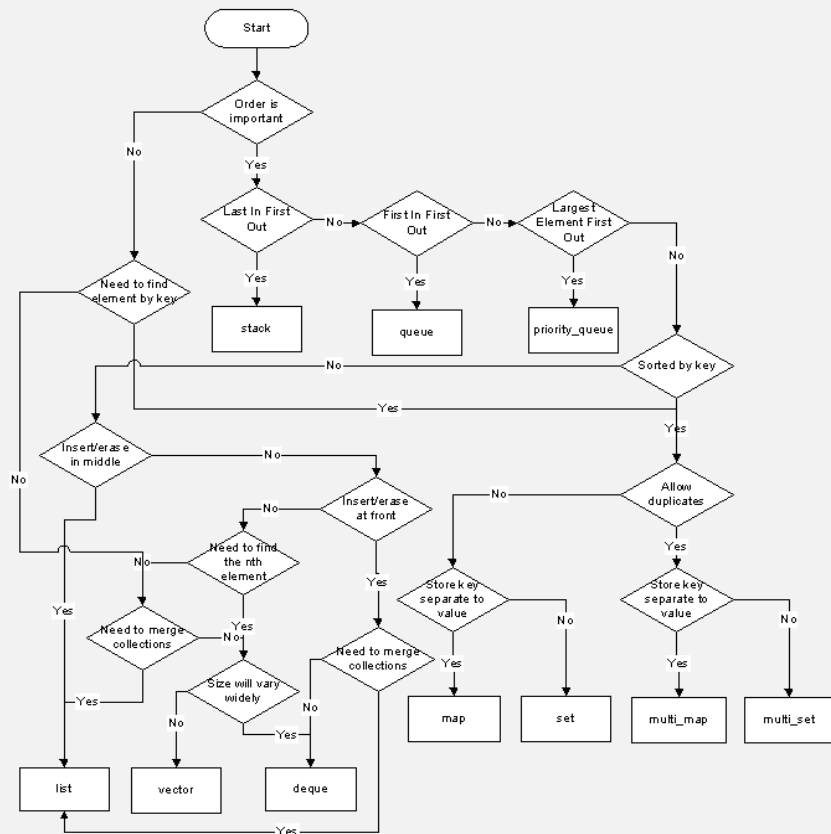
# Коллекционируем данные в .NET

Игорь Лабутин, [ilabutin@gmail.com](mailto:ilabutin@gmail.com), 19.12.2017

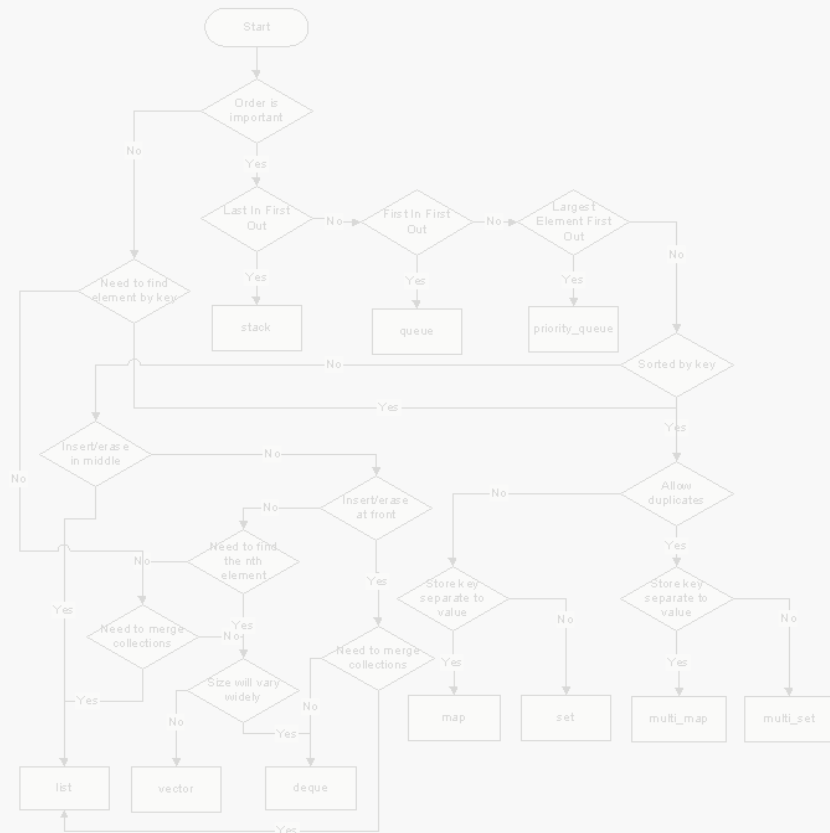
# О себе

- ▶ 16 лет в разработке ПО
  - ▶ C/C++, .NET (C#)
- ▶ Архитектор
  - ▶ Читаю, пишу код
  - ▶ Комментирую код и раздаю советы

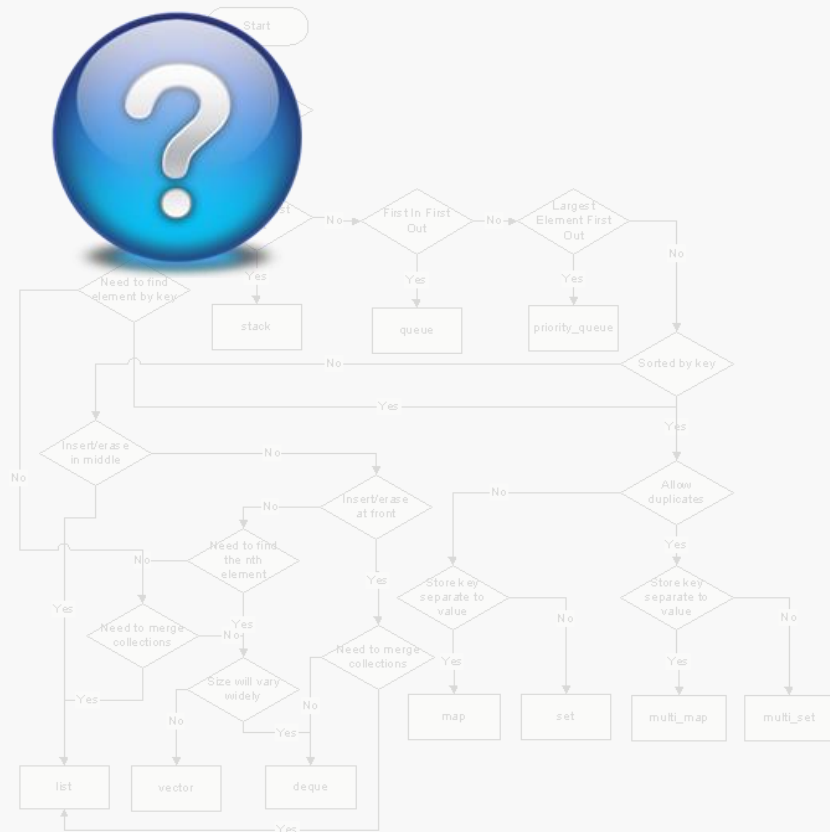
# Выбор коллекции



# Выбор коллекции



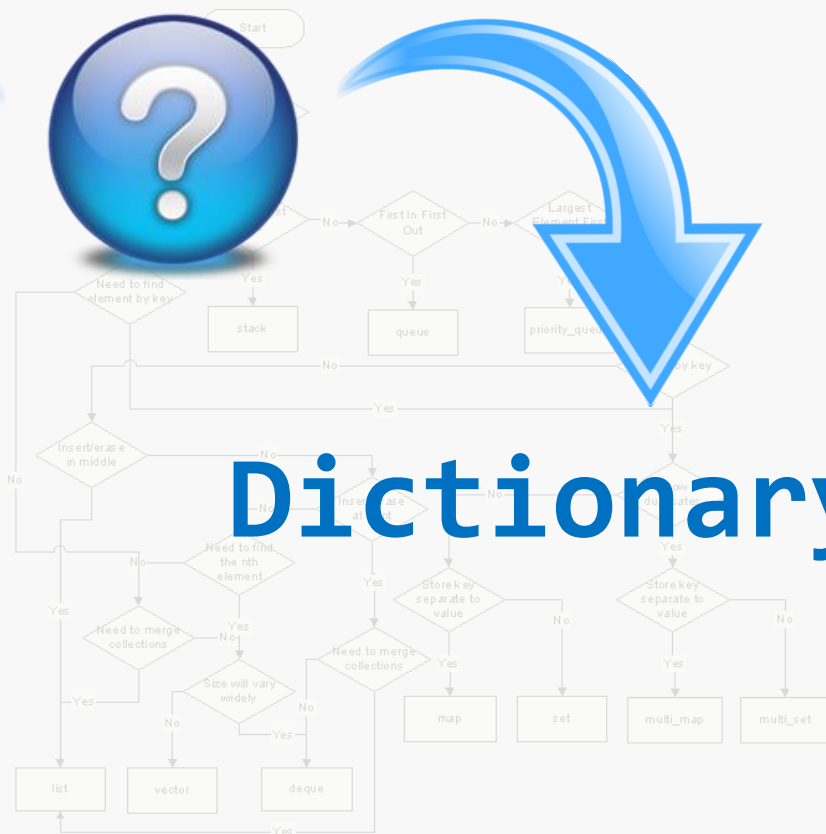
# Выбор коллекции



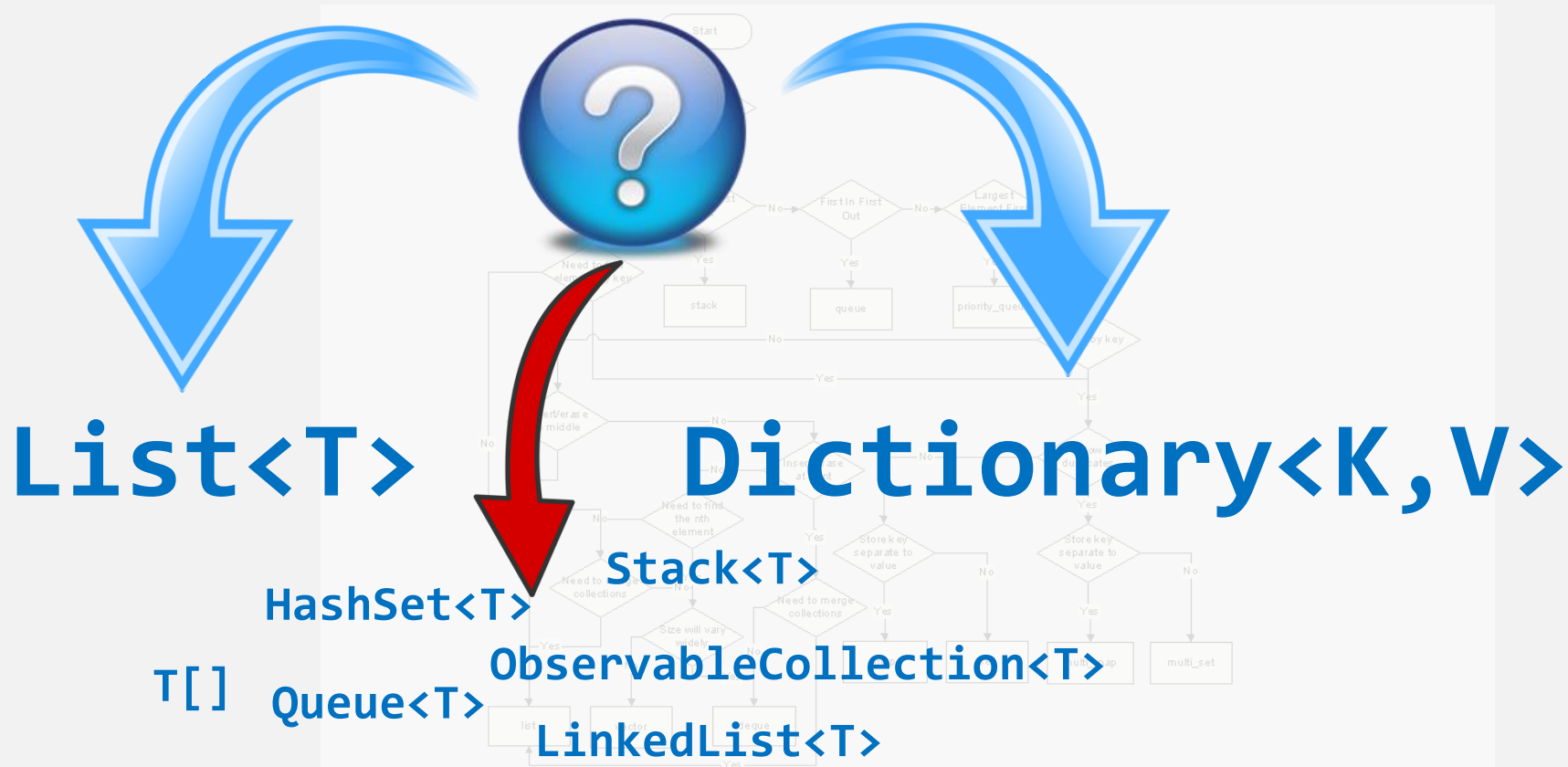
# Выбор коллекции

**List<T>**

**Dictionary<K, V>**



# Выбор коллекции



# План

- ▶ Классические коллекции
- ▶ Потокбезопасность
- ▶ Неизменяемость
- ▶ Собственные коллекции
- ▶ Коллекции в API



# Классические коллекции

- ▶ Массивы
- ▶ Простые коллекции
  - ▶ `List<T>`, `Queue<T>`, `Stack<T>`
- ▶ Сложные коллекции
  - ▶ `Dictionary<K,V>`, `HashSet<T>`

# Простые коллекции

# Простые коллекции

- ▶ Внутри – массив!



# Простые коллекции



- ▶ Внутри – массив!
- ▶ Компактное хранение, но платим при росте

# Простые коллекции



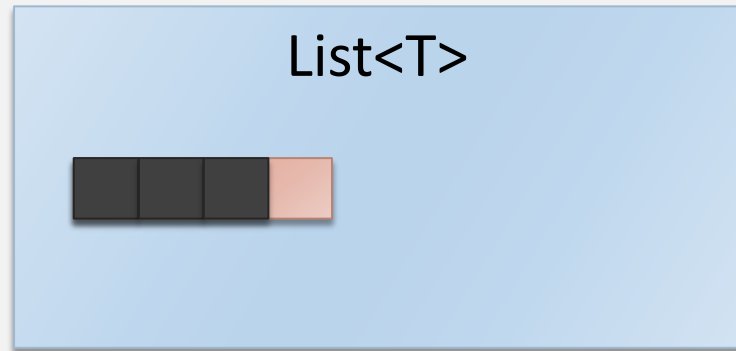
- ▶ Внутри – массив!
- ▶ Компактное хранение, но платим при росте

# Простые коллекции



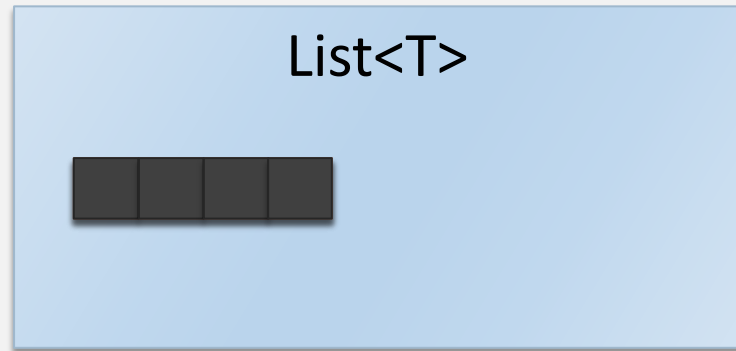
- ▶ Внутри – массив!
- ▶ Компактное хранение, но платим при росте

# Простые коллекции



- ▶ Внутри – массив!
- ▶ Компактное хранение, но платим при росте

# Простые коллекции

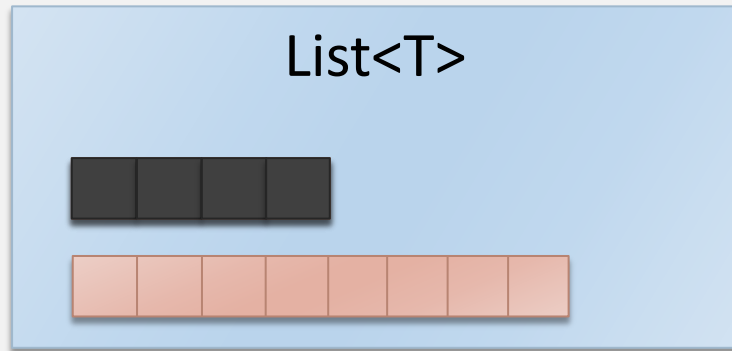


- ▶ Внутри – массив!
- ▶ Компактное хранение, но платим при росте



# Простые коллекции

- ▶ Внутри – массив!
- ▶ Компактное хранение, но платим при росте



# Простые коллекции

- ▶ Внутри – массив!
- ▶ Компактное хранение, но платим при росте



# Простые коллекции

- ▶ Внутри – массив!
- ▶ Компактное хранение, но платим при росте



# Простые коллекции

- ▶ Внутри – массив!
- ▶ Компактное хранение, но платим при росте
- ▶ Растущий массив попадает в LОН
  - ▶ Дорогая сборка мусора
  - ▶ Фрагментация памяти



# А много ли памяти?

```
for (int i = 0; i < N; i++)  
{  
    list.Add(i);  
}
```

# А много ли памяти?

```
for (int i = 0; i < N; i++)  
{  
    list.Add(i);  
}
```

N	1 К	10 К	100 К
Данных в списке	4 Кб	40 Кб	400 Кб
Использовано памяти	8 Кб	128 Кб	1026 Кб

# А много ли памяти?

```
for (int i = 0; i < N; i++)  
{  
    list.Add(i);  
}
```

N	1 K	10 K	100 K
Данных в списке	4 Kб	40 Kб	400 Kб
Использовано памяти	8 Kб	128 Kб	1026 Kб

# А много ли памяти?

```
for (int i = 0; i < N; i++)  
{  
    list.Add(i);  
}
```

N	1 K	10 K	100 K
Данных в списке	4 Кб	40 Кб	400 Кб
Использовано памяти	8 Кб	128 Кб	1026 Кб
Gen0 сборок (на 1000 запусков)	2.6	42	498
Gen1 сборок (на 1000 запусков)	-	-	254
Gen2 сборок (на 1000 запусков)	-	-	244



# А много ли памяти?

```
for (int i = 0; i < N; i++)  
{  
    list.Add(i);  
}
```

N	1 K	10 K	100 K
Данных в списке	4 Кб	40 Кб	400 Кб
Использовано памяти	8 Кб	128 Кб	1026 Кб
Gen0 сборок (на 1000 запусков)	2.6	42	498
Gen1 сборок (на 1000 запусков)	-	-	254
Gen2 сборок (на 1000 запусков)	-	-	244

# Что делать?

# Что делать?

- ▶ `new List<T>(int capacity)`

# Что делать?

- ▶ `new List<T>(int capacity)`

<code>List&lt;T&gt;()</code>	2000
<code>List&lt;T&gt;(int capacity)</code>	
<code>List&lt;T&gt;(IEnumerable&lt;T&gt;)</code>	

# Что делать?

- ▶ `new List<T>(int capacity)`

<code>List&lt;T&gt;()</code>	2000
<code>List&lt;T&gt;(int capacity)</code>	7
<code>List&lt;T&gt;(IEnumerable&lt;T&gt;)</code>	42

# Что делать?

- ▶ `new List<T>(int capacity)`
- ▶ Пишем свой список
  - ▶ Список из «коротких» массивов из пула

<code>List&lt;T&gt;()</code>	2000
<code>List&lt;T&gt;(int capacity)</code>	7
<code>List&lt;T&gt;(IEnumerable&lt;T&gt;)</code>	42

# Что делать?

- ▶ `new List<T>(int capacity)`
- ▶ Пишем свой список
  - ▶ Список из «коротких» массивов из пула
- ▶ Готовый пул: `ArrayPool<T>`
  - ▶ Требуется .NET Core 1.1, в .NET Fx встроенного нет

<code>List&lt;T&gt;()</code>	2000
<code>List&lt;T&gt;(int capacity)</code>	7
<code>List&lt;T&gt;(IEnumerable&lt;T&gt;)</code>	42

# Что делать?

- ▶ `new List<T>(int capacity)`
- ▶ Пишем свой список
  - ▶ Список из «коротких» массивов из пула
- ▶ Готовый пул: `ArrayPool<T>`
  - ▶ Требуется .NET Core 1.1, в .NET Fx встроенного нет
- ▶ Преимущества
  - ▶ Много коротких массивов – либо быстро умирают, либо переиспользуются
  - ▶ Нет копирования данных из короткого массива в длинный

<code>List&lt;T&gt;()</code>	2000
<code>List&lt;T&gt;(int capacity)</code>	7
<code>List&lt;T&gt;(IEnumerable&lt;T&gt;)</code>	42



# Много ли памяти для array pool

```
private List<int[]> chunks = new List<int[]>();  
public void Add(int item)  
{  
    if (length % ChunkSize == 0) chunks.Add(pool.TakeChunk());  
    chunks[length / ChunkSize][length % ChunkSize] = item;  
    length++;  
}
```

# Много ли памяти для array pool

```
private List<int[]> chunks = new List<int[]>();  
public void Add(int item)  
{  
    if (length % ChunkSize == 0) chunks.Add(pool.TakeChunk());  
    chunks[length / ChunkSize][length % ChunkSize] = item;  
    length++;  
}
```

N	1 K	10 K	100 K
Данных в списке	4 Кб	40 Кб	400 Кб
Использовано памяти	8 Кб / 78 Кб	128 Кб / 78 Кб	1026 Кб / 470 Кб

# Много ли памяти для array pool

```
private List<int[]> chunks = new List<int[]>();  
public void Add(int item)  
{  
    if (length % ChunkSize == 0) chunks.Add(pool.TakeChunk());  
    chunks[length / ChunkSize][length % ChunkSize] = item;  
    length++;  
}
```

N	1 K	10 K	100 K
Данных в списке	4 Кб	40 Кб	400 Кб
Использовано памяти	8 Кб / 78 Кб	128 Кб / 78 Кб	1026 Кб / 470 Кб
Gen0 сборок (на 1000 запусков)	2.5 / 25	42 / 25	498 / 90
Gen1 сборок (на 1000 запусков)	-	-	254 / 30
Gen2 сборок (на 1000 запусков)	-	-	244 / -

# Много ли памяти для array pool

```
private List<int[]> chunks = new List<int[]>();  
public void Add(int item)  
{  
    if (length % ChunkSize == 0) chunks.Add(pool.TakeChunk());  
    chunks[length / ChunkSize][length % ChunkSize] = item;  
    length++;  
}
```

N	1 K	10 K	100 K
Данных в списке	4 Кб	40 Кб	400 Кб
Использовано памяти	8 Кб / 78 Кб	128 Кб / 78 Кб	1026 Кб / 470 Кб
Gen0 сборок (на 1000 запусков)	2.5 / 25	42 / 25	498 / 90
Gen1 сборок (на 1000 запусков)	-	-	254 / 30
Gen2 сборок (на 1000 запусков)	-	-	244 / -

# Много ли памяти для array pool

```
private List<int[]> chunks = new List<int[]>();  
public void Add(int item)  
{  
    if (length % ChunkSize == 0) chunks.Add(pool.TakeChunk());  
    chunks[length / ChunkSize][length % ChunkSize] = item;  
    length++;  
}
```

N	1 K	10 K	100 K
Данных в списке	4 Кб	40 Кб	400 Кб
Использовано памяти	8 Кб / 78 Кб	128 Кб / 78 Кб	1026 Кб / 470 Кб
Gen0 сборок (на 1000 запусков)	2.5 / 25	42 / 25	498 / 90
Gen1 сборок (на 1000 запусков)	-	-	254 / 30
Gen2 сборок (на 1000 запусков)	-	-	244 / -

# Сложные коллекции

- ▶ Внутри – опять массив
  - ▶ Или несколько
- ▶ Дополнительные расходы на подсчет хэш-функций
  - ▶ Могут быть слишком велики для небольших коллекций

# Хэш – это дешево

# Хэш – это дешево

- ▶ «Правильная» хэш-функция



# Хэш – это дешево

- ▶ «Правильная» хэш-функция
- ▶ Мало элементов в словаре

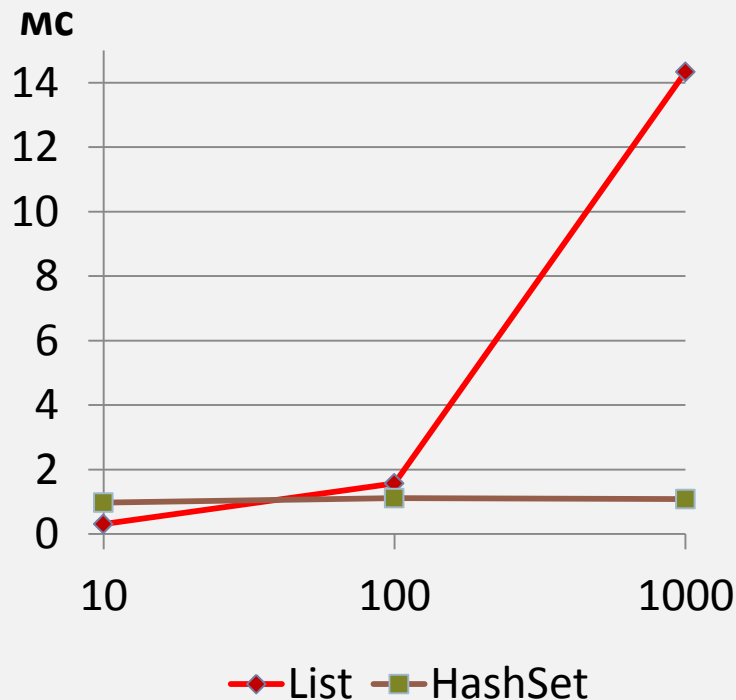
# Хэш – это дешево

- ▶ «Правильная» хэш-функция
- ▶ Мало элементов в словаре
  - ▶ Подсчет хэша vs линейный поиск в списке

# Хэш – это дешево

- ▶ «Правильная» хэш-функция
- ▶ Мало элементов в словаре
  - ▶ Подсчет хэша vs линейный поиск в списке

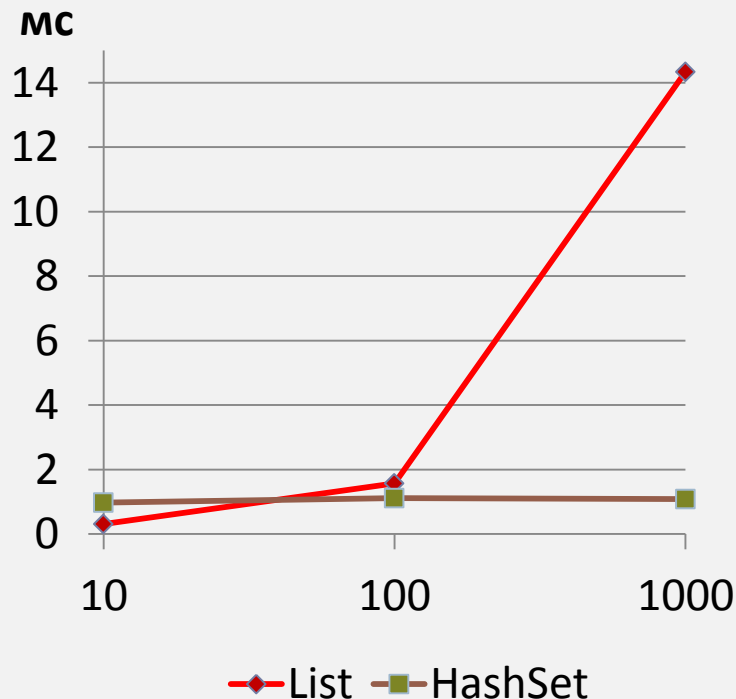
Время поиска 10 000 раз в коллекции строк размером N



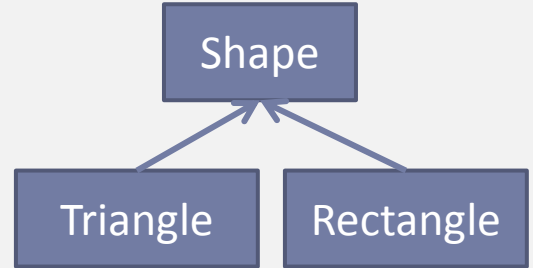
# Хэш – это дешево

- ▶ «Правильная» хэш-функция
- ▶ Мало элементов в словаре
  - ▶ Подсчет хэша vs линейный поиск в списке
- ▶ Roslyn: Collection of imported PE Names (см. <http://bit.ly/2wYE9lss>)

Время поиска 10 000 раз в коллекции строк размером N



# Проблемы продолжают



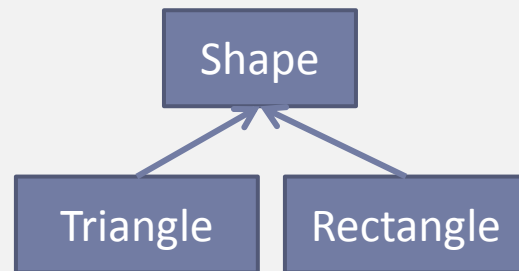
# Проблемы продолжают

## ► Массивы ковариантны

```
Triangle[] tr = ...;
```

```
Shape[] shapes = tr;
```

```
public void DoSomething(Shape[] t)
{
    t[0] = new Rectangle();
}
```



# Проблемы продолжают

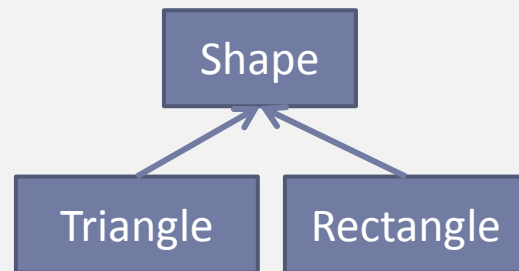
## ► Массивы ковариантны

```
Triangle[] tr = ...;  
Shape[] shapes = tr;
```

```
public void DoSomething(Shape[] t)  
{  
    t[0] = new Rectangle();  
}
```



**ArrayTypeMismatchException**



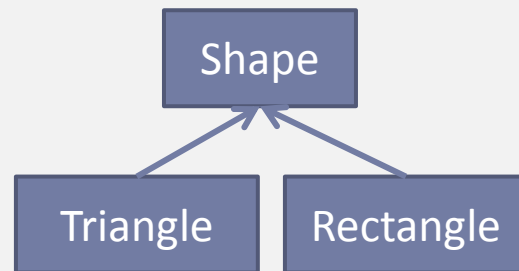
# Проблемы продолжают

## ► Массивы ковариантны

```
Triangle[] tr = ...;  
Shape[] shapes = tr;
```

```
public void DoSomething(Shape[] t)  
{  
    t[0] = new Rectangle();  
}
```

← **ArrayTypeMismatchException**



## ► Проверки типов в рантайме



# Проблемы продолжаются

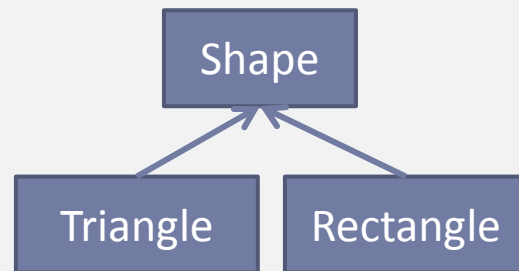
- ▶ **Массивы ковариантны**

```
Triangle[] tr = ...;  
Shape[] shapes = tr;
```

```
public void DoSomething(Shape[] t)  
{  
    t[0] = new Rectangle();  
}
```



**ArrayTypeMismatchException**



- ▶ Проверки типов в рантайме
- ▶ Почему так?

# Проблемы продолжают

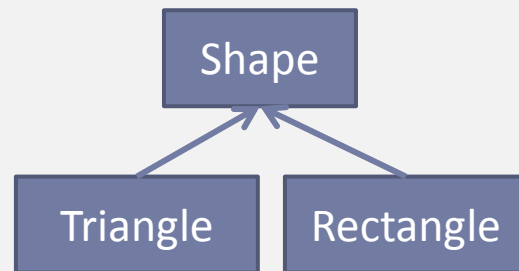
## ► Массивы ковариантны

```
Triangle[] tr = ...;  
Shape[] shapes = tr;
```

```
public void DoSomething(Shape[] t)  
{  
    t[0] = new Rectangle();  
}
```



**ArrayTypeMismatchException**



## ► Проверки типов в рантайме

## ► Почему так?

- “It was added to the CLR because Java requires it”

© Eric Lippert (см. <http://bit.ly/2wZetW3>)

# Вариантность в наши дни

- ▶ Ковариантность и контравариантность
  - ▶ Для интерфейсов и делегатов
  - ▶ Проверки на этапе компиляции
- ▶ Ключевые слова `in/out`

# Вариантность в наши дни

- ▶ Ковариантность и контравариантность
  - ▶ Для интерфейсов и делегатов
  - ▶ Проверки на этапе компиляции
- ▶ Ключевые слова `in/out`

```
interface IEnumerable<out T>
```

```
IEnumerable<Circle> GetCircles();  
IEnumerable<Shape> x = GetCircles();
```

# Вариантность в наши дни

- ▶ Ковариантность и контравариантность
  - ▶ Для интерфейсов и делегатов
  - ▶ Проверки на этапе компиляции
- ▶ Ключевые слова `in/out`

```
interface IEnumerable<out T>
```

```
delegate void Action<in T>
```

```
IEnumerable<Circle> GetCircles();  
IEnumerable<Shape> x = GetCircles();
```

```
void ProcessShapes(Action<Circle> action);
```

```
Action<Shape> shapeAction = shape => shape.Draw();  
ProcessShapes(shapeAction);
```

# Вариантность в наши дни

- ▶ Ковариантность и контравариантность
  - ▶ Для интерфейсов и делегатов
  - ▶ Проверки на этапе компиляции
- ▶ Ключевые слова `in/out`

```
interface IEnumerable<out T>
```

```
IEnumerable<Circle> GetCircles();  
IEnumerable<Shape> x = GetCircles();
```

```
delegate void Action<in T>
```

```
void ProcessShapes(Action<Circle> action);
```

```
delegate TResult  
    Func<in T1, out TResult>(T1 arg1);
```

```
Action<Shape> shapeAction = shape => shape.Draw();  
ProcessShapes(shapeAction);
```

# Классические коллекции - выводы

- ▶ Следить за потреблением памяти
  - ▶ Явно использовать конструкторы с указанием размера
  - ▶ Писать свои коллекции при необходимости
- ▶ Для словарей с малым количеством данных рассмотреть взамен списки

# Оффтопик: MemoryStream

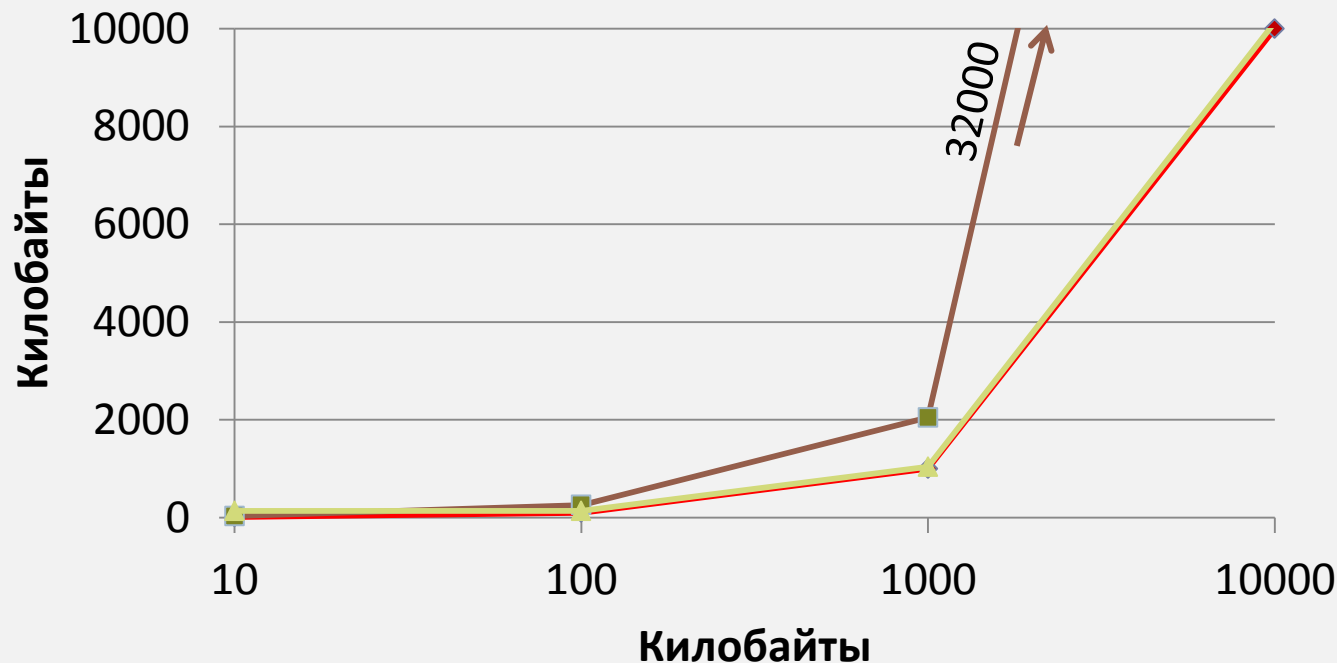
- ▶ Не совсем коллекция, но ведет себя похоже
  - ▶ Массив внутри
  - ▶ Отсюда проблемы с LOH и потреблением памяти



# MemoryStream - решение

- ▶ Microsoft.IO.RecyclableMemoryStream

- ▶ Доступен в .NET Standard 1.6, .NET 4.0+



Потребление памяти  
при добавлении  
 $N \cdot 1024$  байт

- ◆ Данные
- MemoryStream
- ▲ RecyclableMS

# План

- ▶ Классические коллекции
- ▶ **Потокобезопасность**
- ▶ Неизменяемость
- ▶ Собственные коллекции
- ▶ Коллекции в API

# Многопоточность

- ▶ Все стандартные коллекции не потокобезопасны
  - ▶ При наличии операций записи
  - ▶ Нужны внешние блокировки
- ▶ Блокировки
  - ▶ `lock (...) {}`
  - ▶ `ReaderWriterLockSlim`
- ▶ Нужно передавать 2 объекта

# Решение

- ▶ `BlockingCollection<T>`
  - ▶ Сценарий producer-consumer
  - ▶ Поддерживает синхронные и асинхронные операции
- ▶ `System.Collections.Concurrent`
  - ▶ Все привычные операции как правило имеют префикс Try..
  - ▶ Отсутствует `ConcurrentList<T>`

# Потокобезопасность

## ► Получение значения

```
if (dictionary.ContainsKey(key))  
{  
    var value = dictionary[key];  
    // Do something with 'value'  
}
```

# Потокобезопасность

## ► Получение значения

```
if (dictionary.ContainsKey(key))  
{  
    var value = dictionary[key];  
    // Do something with 'value'  
}
```

```
if (concurrentDictionary.TryGetValue(key, out int value))  
{  
    // Do something with 'value'  
}
```

# Потокобезопасность

## ► Получение значения

```
if (dictionary.ContainsKey(key))  
{  
    var value = dictionary[key];  
    // Do something with 'value'  
}
```

```
if (concurrentDictionary.TryGetValue(key, out int value))  
{  
    // Do something with 'value'  
}
```

## ► Обновление значения

```
if (dictionary[key] == 10)  
{  
    dictionary[key] = 20;  
}
```

# Потокобезопасность

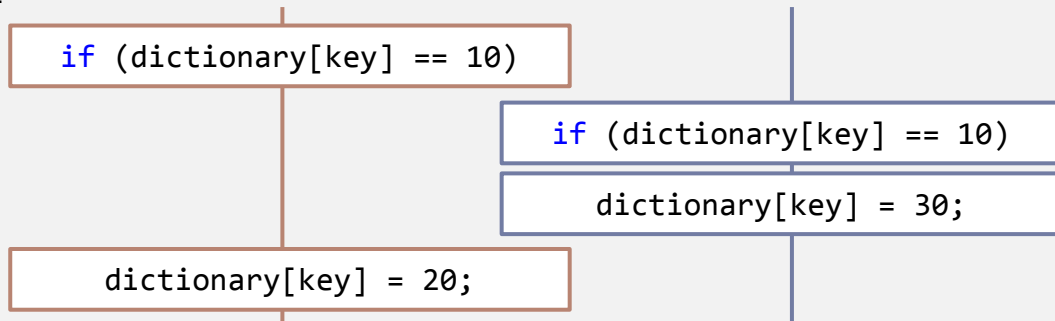
## ► Получение значения

```
if (dictionary.ContainsKey(key))  
{  
    var value = dictionary[key];  
    // Do something with 'value'  
}
```

```
if (concurrentDictionary.TryGetValue(key, out int value))  
{  
    // Do something with 'value'  
}
```

## ► Обновление значения

```
if (dictionary[key] == 10)  
{  
    dictionary[key] = 20;  
}
```





# Потокобезопасность

## ► Получение значения

```
if (dictionary.ContainsKey(key))  
{  
    var value = dictionary[key];  
    // Do something with 'value'  
}
```

```
if (concurrentDictionary.TryGetValue(key, out int value))  
{  
    // Do something with 'value'  
}
```

## ► Обновление значения

```
if (dictionary[key] == 10)  
{  
    dictionary[key] = 20;  
}
```

```
if (!concurrentDictionary.TryUpdate(key, 20, 10))  
{  
    // Handle somehow !?!  
}
```

if (dictionary[key] == 10)

if (dictionary[key] == 10)

dictionary[key] = 30;

dictionary[key] = 20;

# Потокобезопасность

## ► Получение значения

```
if (dictionary.ContainsKey(key))  
{  
    var value = dictionary[key];  
    // Do something with 'value'  
}
```

```
if (concurrentDictionary.TryGetValue(key, out int value))  
{  
    // Do something with 'value'  
}
```

## ► Обновление значения

```
if (dictionary[key] == 10)  
{  
    dictionary[key] = 20;  
}
```

if (dictionary[key] == 10)

dictionary[key] = 20;

```
if (!concurrentDictionary.TryUpdate(key, 20, 10))  
{  
    // Handle somehow !?!  
}
```

if (dictionary[key] == 10)

dictionary[key] = 30;

# Производительность

- ▶ По возможности используется lock-free подход
  - ▶ Обычные блокировки тоже есть
- ▶ Заточены под разные задачи
  - ▶ ConcurrentDictionary – lock-free чтение
  - ▶ ConcurrentBag – читатель/писатель
  - ▶ Queue/Stack – полностью lock-free
- ▶ Сравнение от Stephen Toub - <http://bit.ly/2vyv5DG>

# Потокобезопасность - выводы

- ▶ Рассмотреть ConcurrentCollections с оглядкой на производительность
- ▶ Практика: часто List<T> с lock() вполне хватает
- ▶ Для словарей которые только читают – использовать обычный Dictionary<K, V>

# Коллекции с async/await

- ▶ TPL DataFlow: `BufferBlock<T>`
- ▶ AsyncEx (<https://github.com/StephenCleary/AsyncEx>)

# План

- ▶ Классические коллекции
- ▶ Потокобезопасность
- ▶ **Неизменяемость**
- ▶ Собственные коллекции
- ▶ Коллекции в API

# Неизменяемость

- ▶ Изменение объекта посредством создания нового
- ▶ Гарантированная потокобезопасность
- ▶ Защита от изменений любым API

# Неизменяемость

## ► Всё просто:

```
var list = new List<int>(100);  
var readOnlyList = list.AsReadOnly();  
readOnlyList.Add(5); // throws NotSupportedException
```



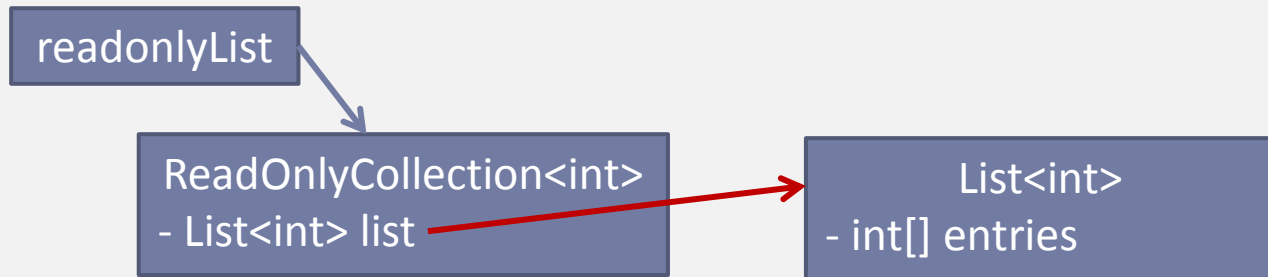
# Неизменяемость

## ▶ Всё просто:

```
var list = new List<int>(100);  
var readOnlyList = list.AsReadOnly();  
readOnlyList.Add(5); // throws NotSupportedException
```

## ▶ На самом деле нет!

### ▶ ReadOnly это не Immutable



# Неизменяемость

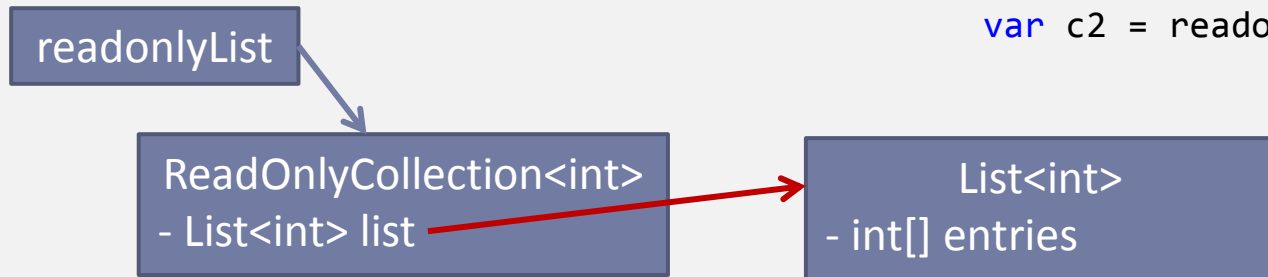
## ► Всё просто:

```
var list = new List<int>(100);  
var readOnlyList = list.AsReadOnly();  
readOnlyList.Add(5); // throws NotSupportedException
```

## ► На самом деле нет!

### ► Readonly это не Immutable

```
var c1 = readOnlyList.Count; // 100  
list.Add(42);  
var c2 = readOnlyList.Count; // 101
```



# Неизменяемость - решение

- ▶ `System.Collections.Immutable`
- ▶ Поддержаны
  - ▶ .NET 4.5+
  - ▶ .NET Standard 1.6+
- ▶ Узнайте характеристики по производительности!

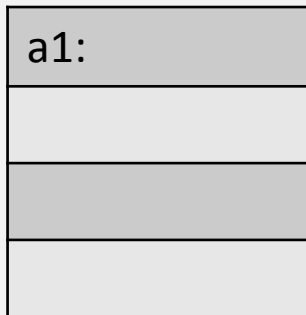
# Неизменяемые Array и List - память

```
var a1 = new int[8];
```

```
var a2 = ImmutableArray.Create(a1);
```

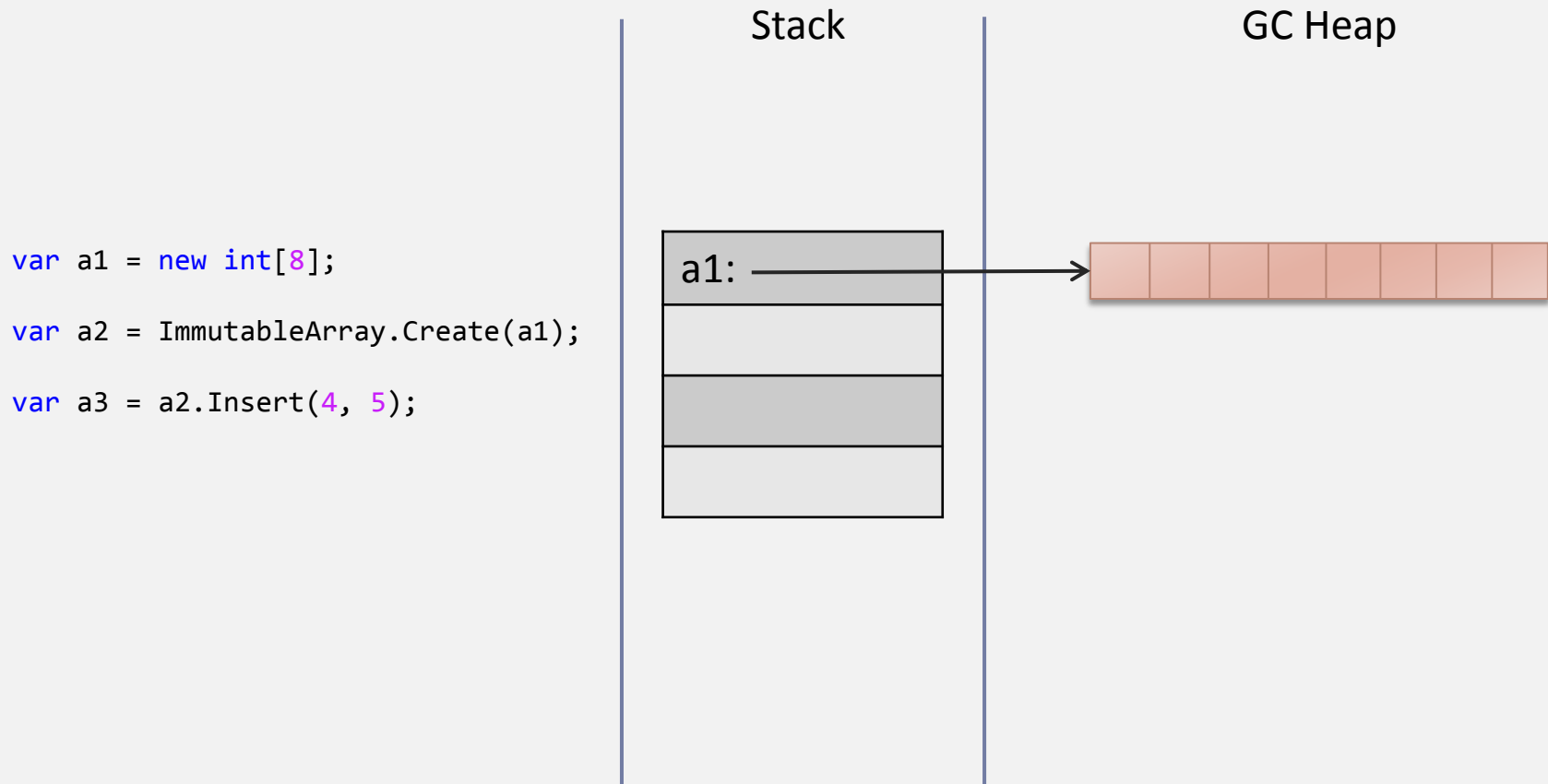
```
var a3 = a2.Insert(4, 5);
```

Stack



GC Heap

# Неизменяемые Array и List - память



# Неизменяемые Array и List - память

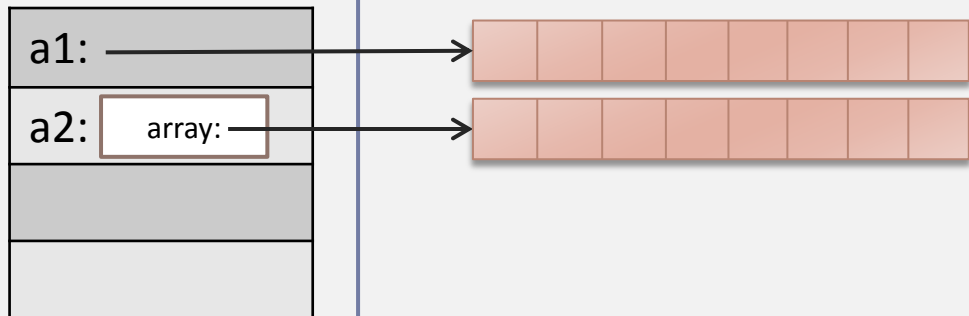
```
var a1 = new int[8];
```

```
var a2 = ImmutableArray.Create(a1);
```

```
var a3 = a2.Insert(4, 5);
```

Stack

GC Heap

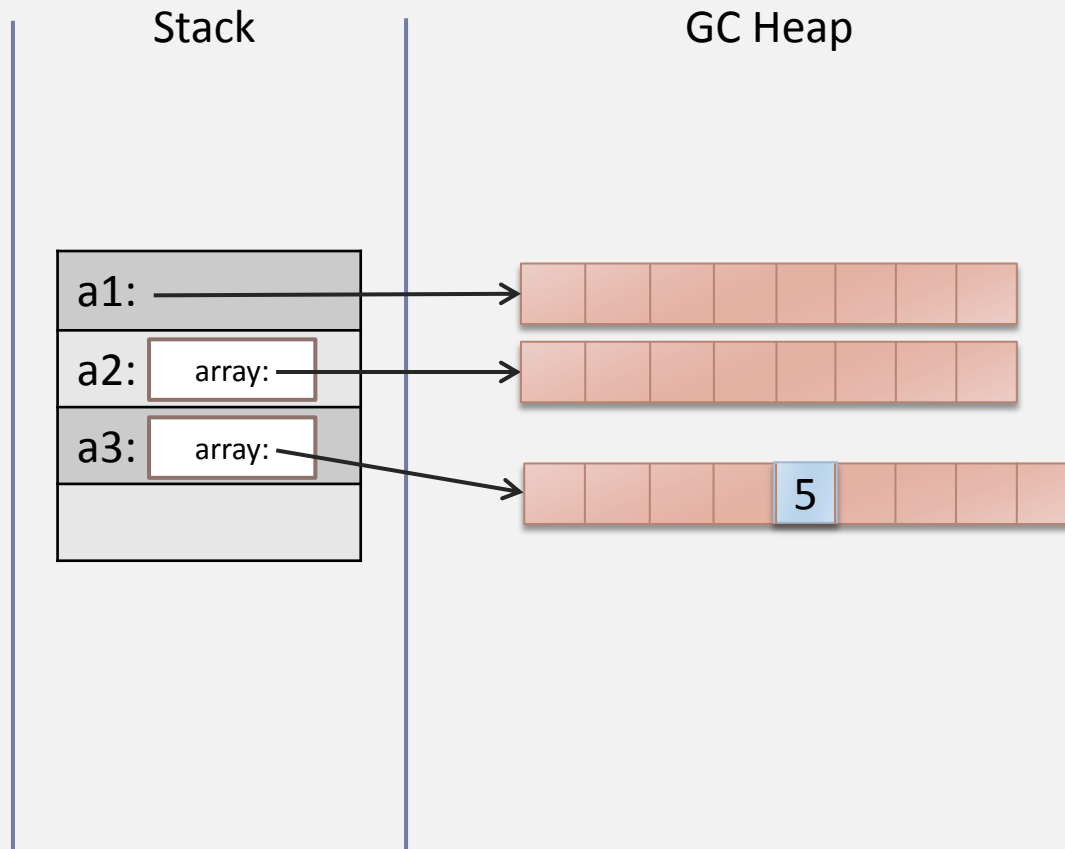


# Неизменяемые Array и List - память

```
var a1 = new int[8];
```

```
var a2 = ImmutableArray.Create(a1);
```

```
var a3 = a2.Insert(4, 5);
```



# Неизменяемые Array и List - память

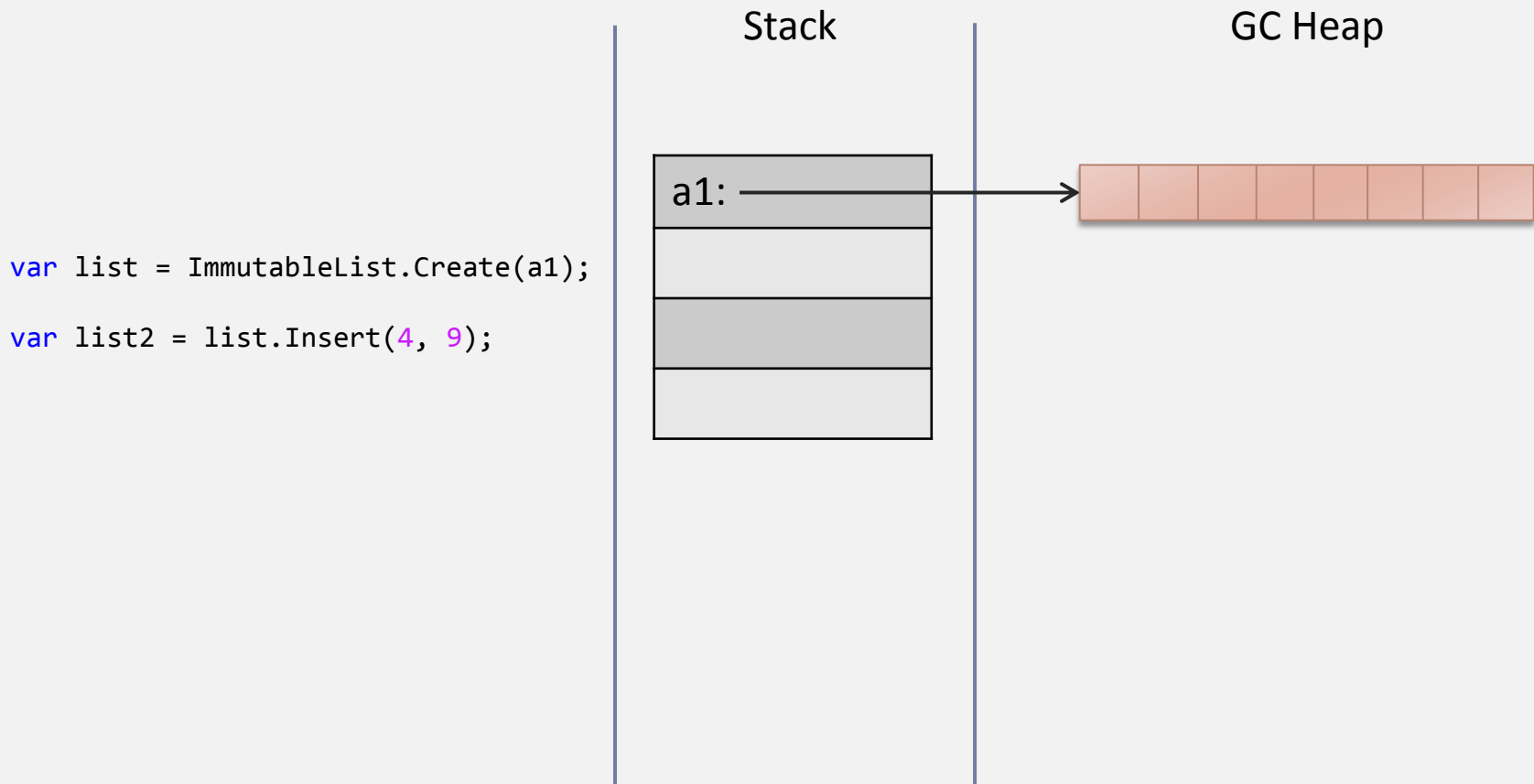
Stack

GC Heap

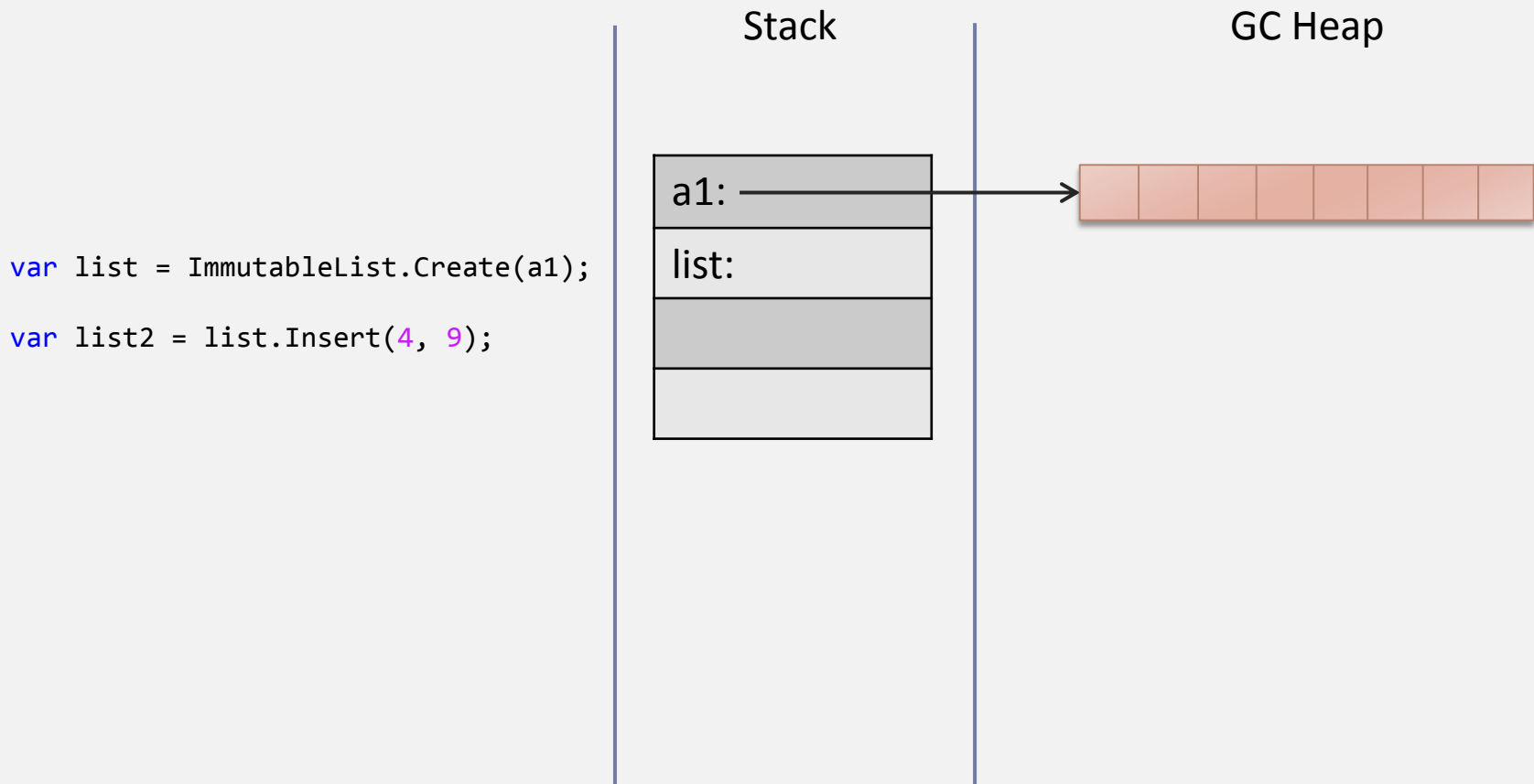
```
var list = ImmutableList.Create(a1);  
var list2 = list.Insert(4, 9);
```



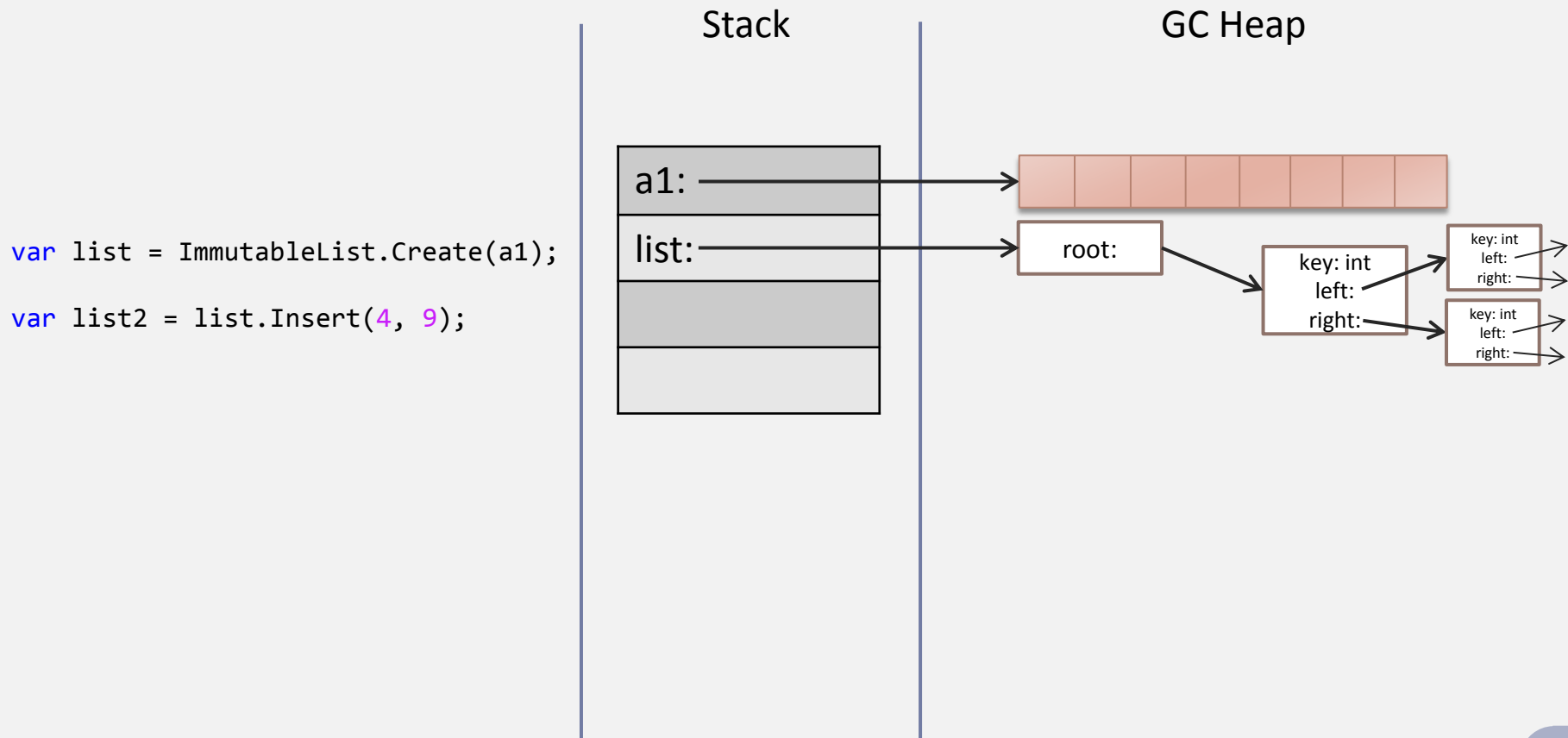
# Неизменяемые Array и List - память



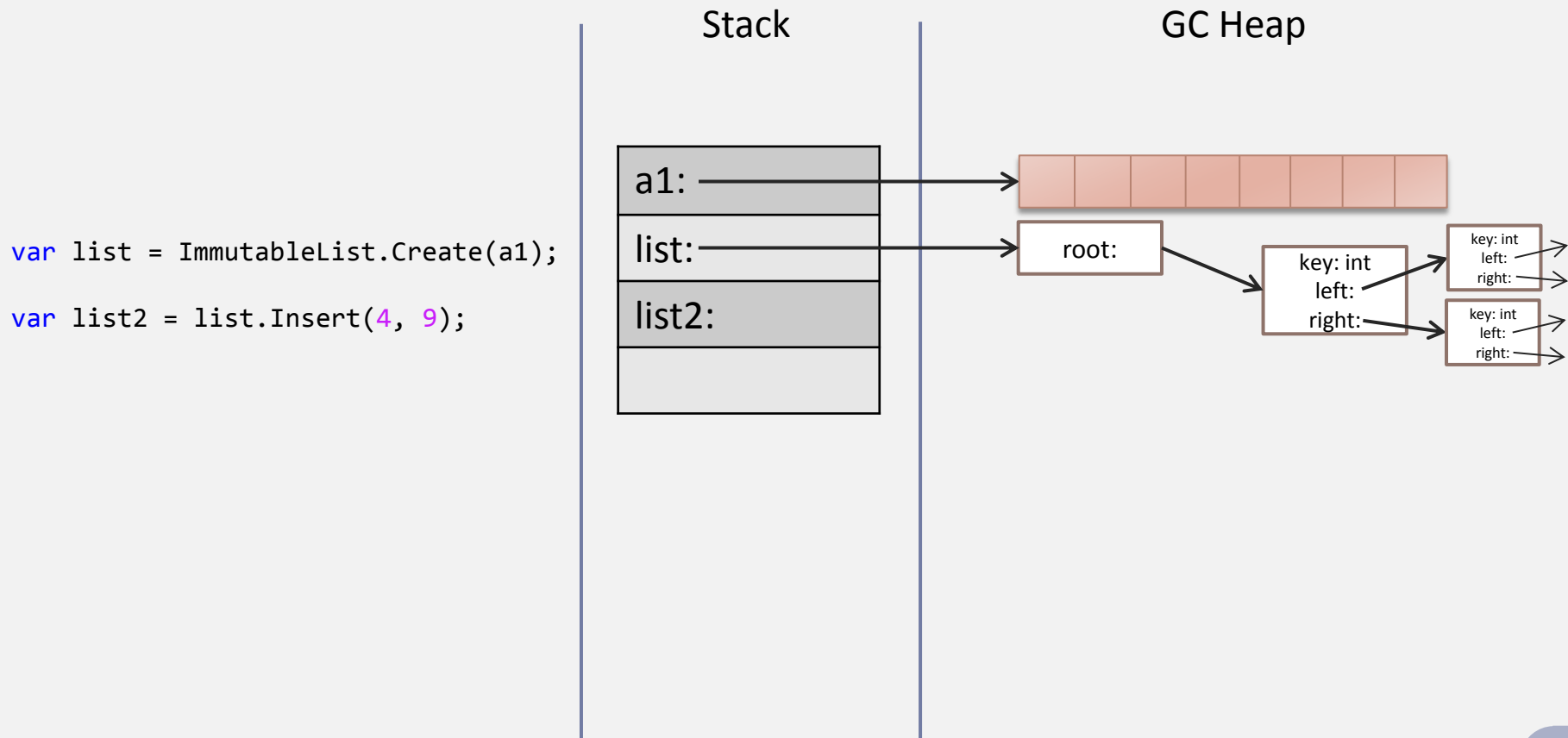
# Неизменяемые Array и List - память



# Неизменяемые Array и List - память

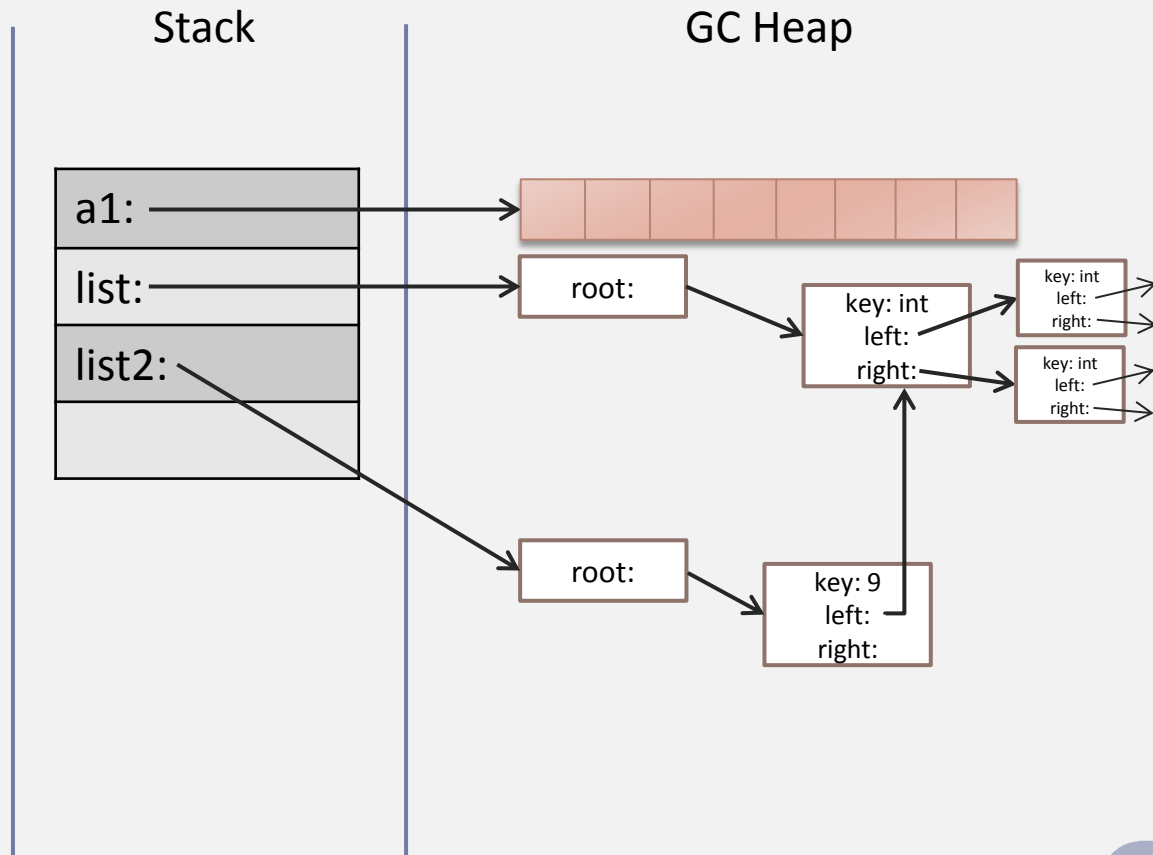


# Неизменяемые Array и List - память



# Неизменяемые Array и List - память

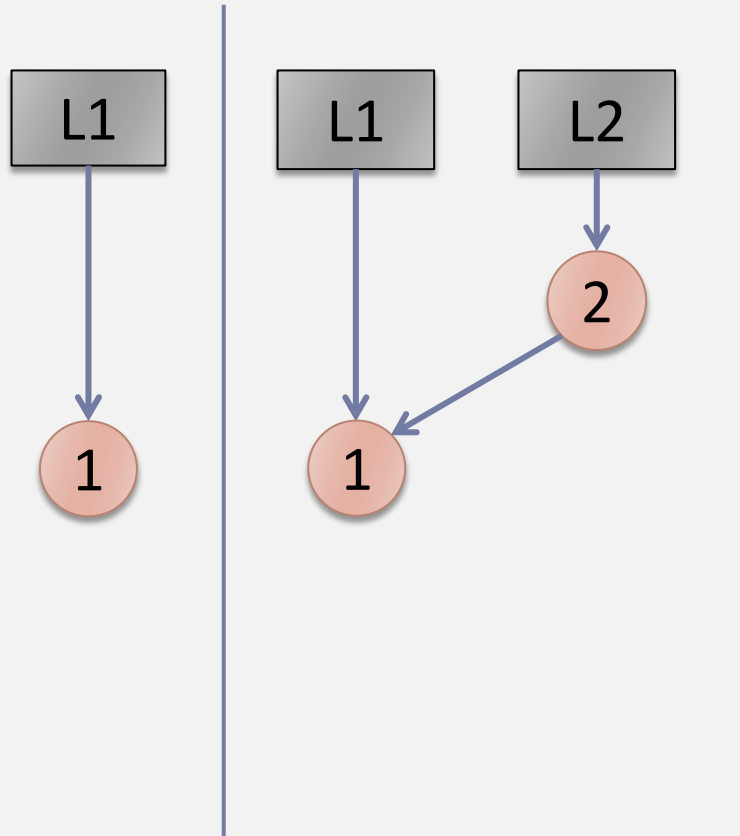
```
var list = ImmutableList.Create(a1);  
var list2 = list.Insert(4, 9);
```



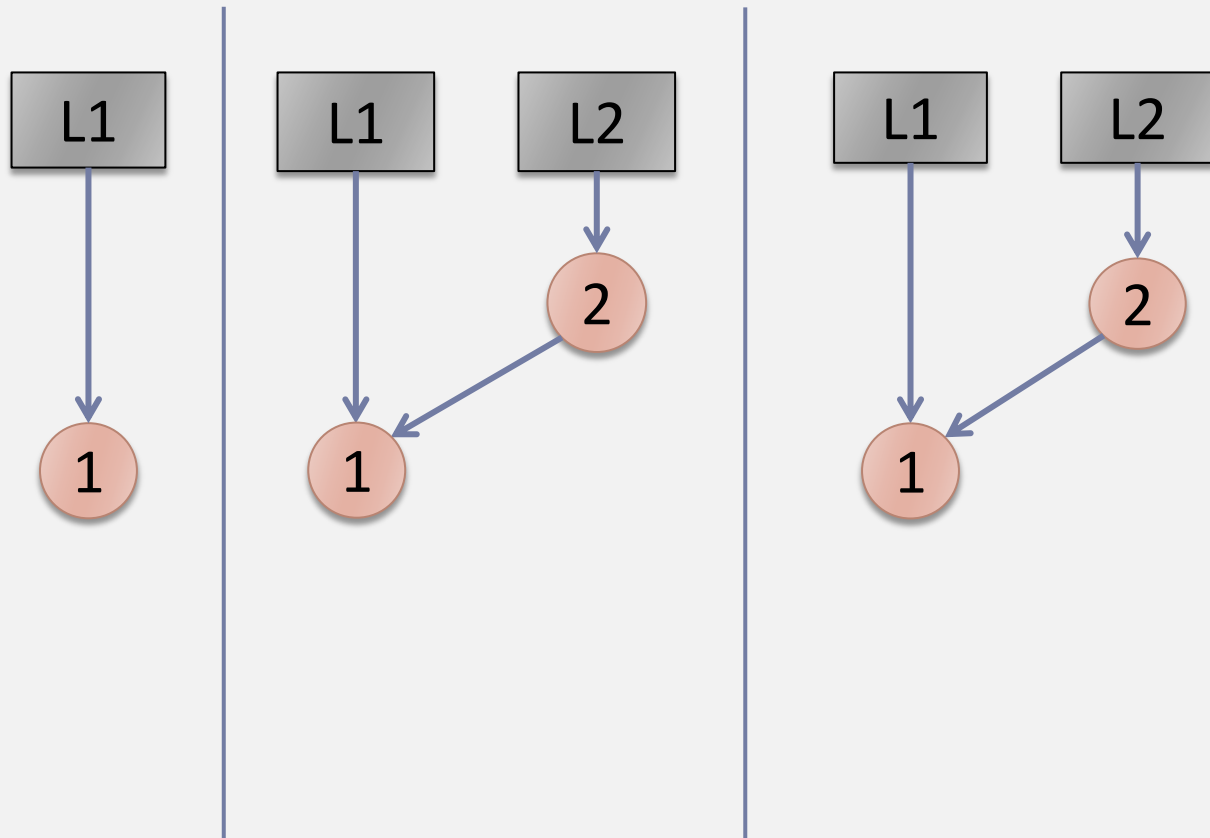
# Неизменяемые Array и List - память



# Неизменяемые Array и List - память

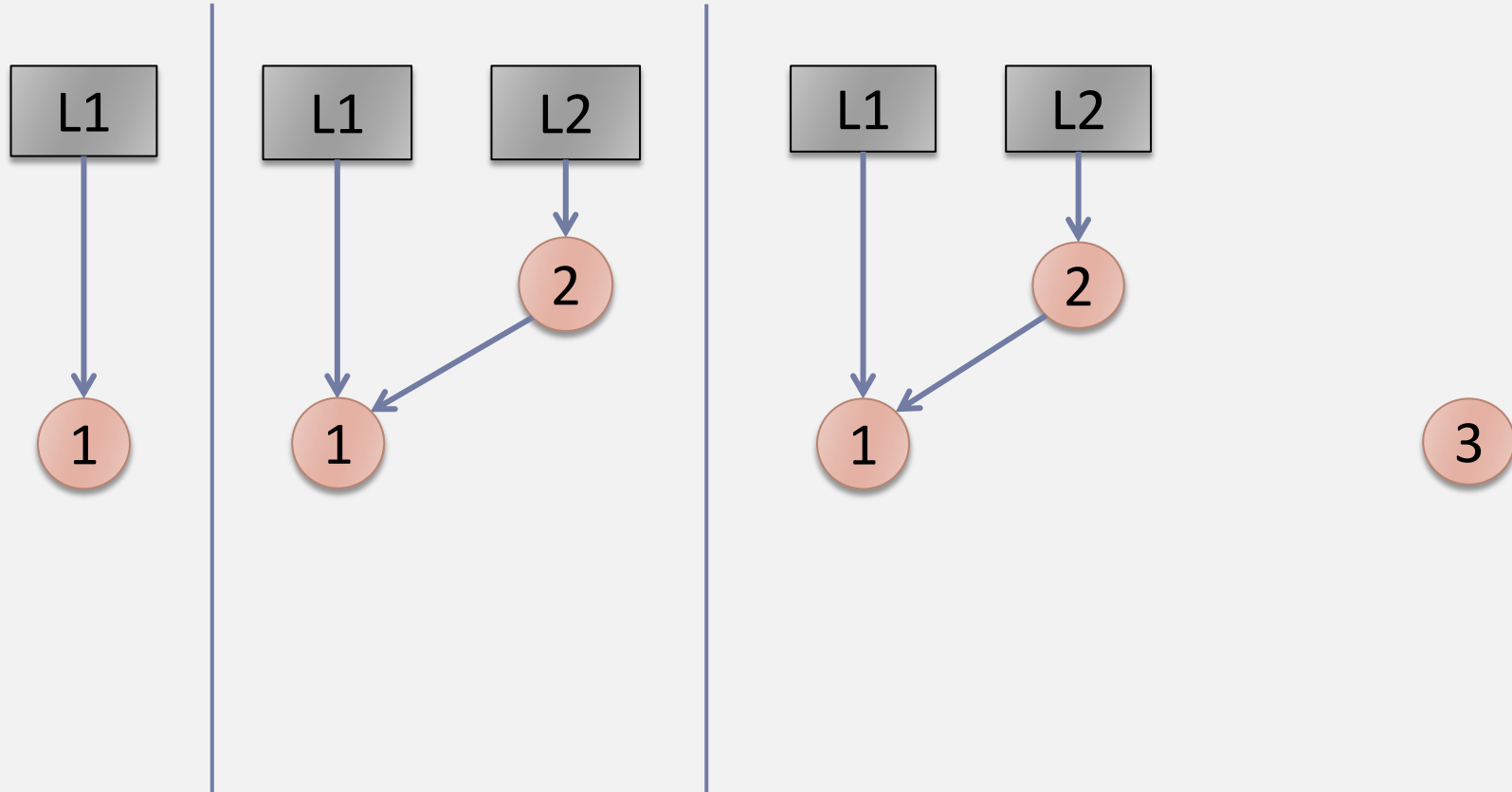


# Неизменяемые Array и List - память

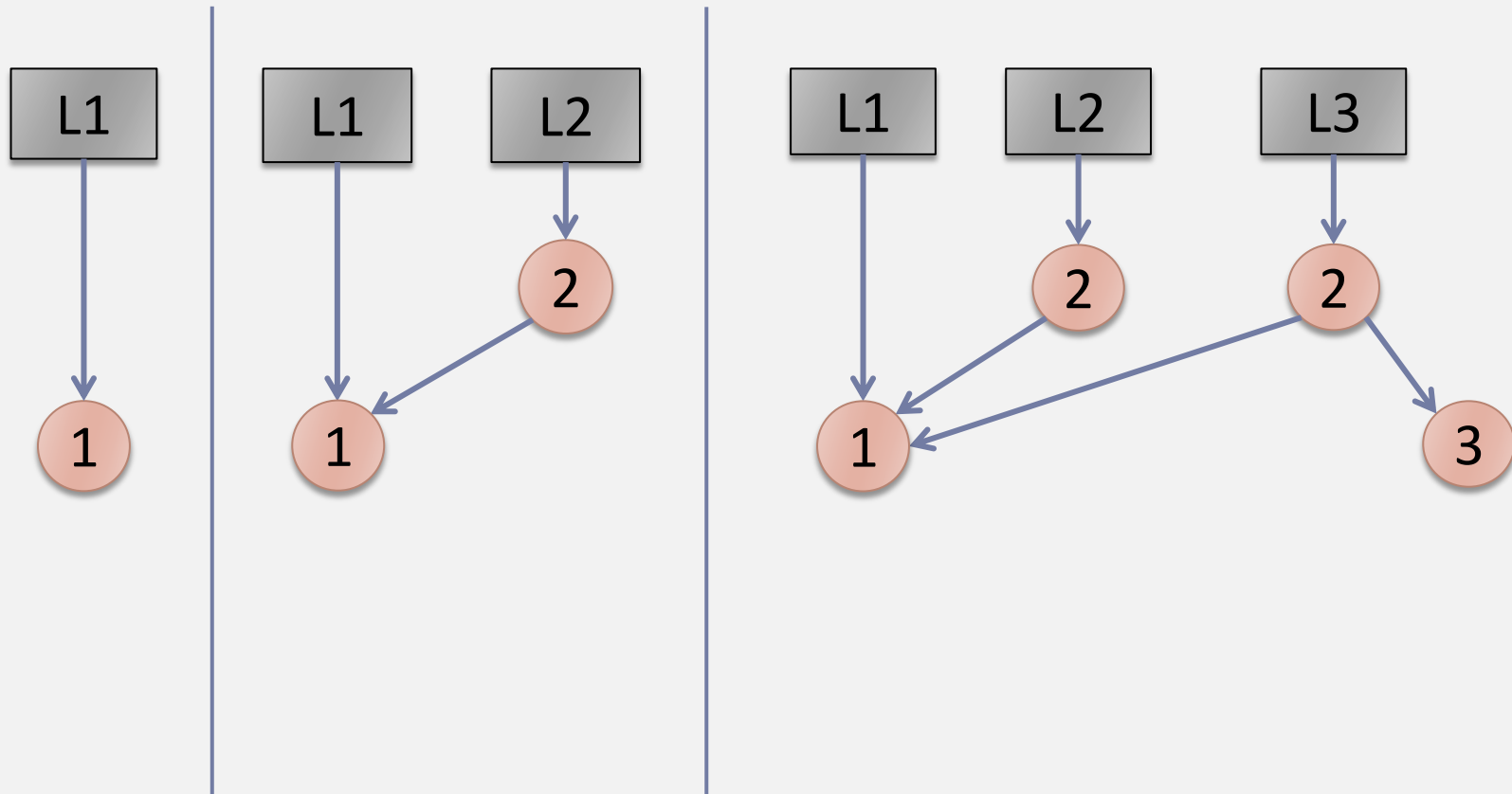




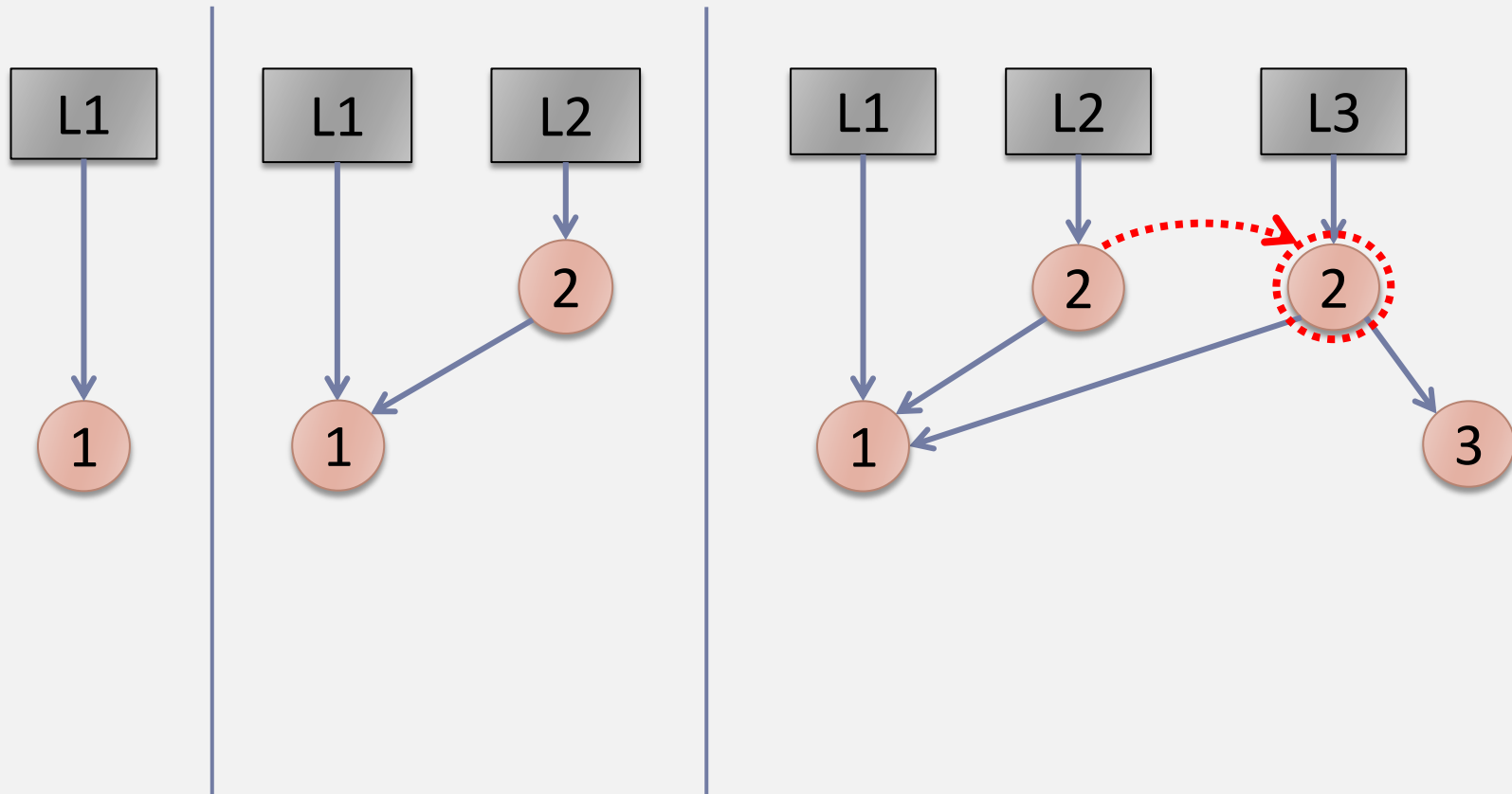
# Неизменяемые Array и List - память



# Неизменяемые Array и List - память



# Неизменяемые Array и List - память



# Неизменяемость - выводы

- ▶ Создание новых объектов – нагрузка на GC

# Неизменяемость - выводы

- ▶ Создание новых объектов – нагрузка на GC
- ▶ Использование неизменяемых объектов – архитектурное решение

# План

- ▶ Классические коллекции
- ▶ Потокобезопасность
- ▶ Неизменяемость
- ▶ **Собственные коллекции**
- ▶ Коллекции в API

# Специализированные коллекции

# Специализированные коллекции

- ▶ `System.Collections.Specialized`



# Специализированные коллекции

- ▶ `System.Collections.Specialized`
  - ▶ Не надо использовать (не типизированы)

# Специализированные коллекции

- ▶ `System.Collections.Specialized`
  - ▶ Не надо использовать (не типизированы)
- ▶ `System.Collections.ObjectModel`
  - ▶ `Collection<T>`, `ReadOnlyCollection<T>`
    - ▶ Упрощенный и расширяемый `List<T>`

# Специализированные коллекции

- ▶ `System.Collections.Specialized`
  - ▶ Не надо использовать (не типизированы)
- ▶ `System.Collections.ObjectModel`
  - ▶ `Collection<T>`, `ReadOnlyCollection<T>`
    - ▶ Упрощенный и расширяемый `List<T>`
  - ▶ `ObservableCollection<T>`
    - ▶ Извещение об изменениях коллекции
    - ▶ Широко используется в WPF

# Специализированные коллекции

- ▶ `System.Collections.Specialized`
  - ▶ Не надо использовать (не типизированы)
- ▶ `System.Collections.ObjectModel`
  - ▶ `Collection<T>`, `ReadOnlyCollection<T>`
    - ▶ Упрощенный и расширяемый `List<T>`
  - ▶ `ObservableCollection<T>`
    - ▶ Извещение об изменениях коллекции
    - ▶ Широко используется в WPF
  - ▶ `abstract KeyedCollection<K, V>`
    - ▶ До определенного момента – `Collection<V>`  
потом – еще и `Dictionary<K, V>`

# Собственные коллекции

# Собственные коллекции

- ▶ Реализовывать необходимые интерфейсы
  - ▶ Или наследовать от встроенных коллекций

# Собственные коллекции

- ▶ Реализовывать необходимые интерфейсы
  - ▶ Или наследовать от встроенных коллекций
- ▶ Собственный итератор – структура

# Собственные коллекции

- ▶ Реализовывать необходимые интерфейсы
  - ▶ Или наследовать от встроенных коллекций
- ▶ Собственный итератор – структура

```
interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator();
}
```



# Собственные коллекции

- ▶ Реализовывать необходимые интерфейсы
  - ▶ Или наследовать от встроенных коллекций
- ▶ Собственный итератор – структура

```
interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator();
}
```

```
class Collection<T> : IEnumerable<T>
{
    IEnumerator<T> GetEnumerator() { ... }

}
```

# Собственные коллекции

- ▶ Реализовывать необходимые интерфейсы
  - ▶ Или наследовать от встроенных коллекций
- ▶ Собственный итератор – структура

```
interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator();
}
```

```
class Collection<T> : IEnumerable<T>
{
    IEnumerator<T> GetEnumerator() { ... }

    public struct Enumerator<T>
        : IEnumerator<T>
    {
        bool MoveNext() { ... }
        T Current { get { ... } }
    }
}
```

# Итератор-структура

```
var list = new List<int>();  
...  
foreach (var item in list)  
{  
    // do something  
}
```

---

# Итератор-структура

```
var list = new List<int>();  
...  
foreach (var item in list)  
{  
    // do something  
}
```

```
var list = new List<int>();  
...  
List<int>.Enumerator e = list.GetEnumerator();  
while (e.MoveNext())  
{  
    int item = e.Current;  
    // do something  
}
```

# Итератор-структура

```
var list = new List<int>();  
...  
foreach (var item in list)  
{  
    // do something  
}
```

```
var list = new List<int>();  
...  
List<int>.Enumerator e = list.GetEnumerator();  
while (e.MoveNext())  
{  
    int item = e.Current;  
    // do something  
}
```

---

```
var xList = (IList<int>)list;  
foreach (var xItem in xList)  
{  
    // do something  
}
```

# Итератор-структура

```
var list = new List<int>();  
...  
foreach (var item in list)  
{  
    // do something  
}
```

```
var list = new List<int>();  
...  
List<int>.Enumerator e = list.GetEnumerator();  
while (e.MoveNext())  
{  
    int item = e.Current;  
    // do something  
}
```

---

```
var xList = (IList<int>)list;  
foreach (var xItem in xList)  
{  
    // do something  
}
```

```
var xList = (IList<int>)list;  
IEnumerator<int> x = xList.GetEnumerator();  
while (x.MoveNext())  
{  
    int xItem = x.Current;  
    // do something  
}
```

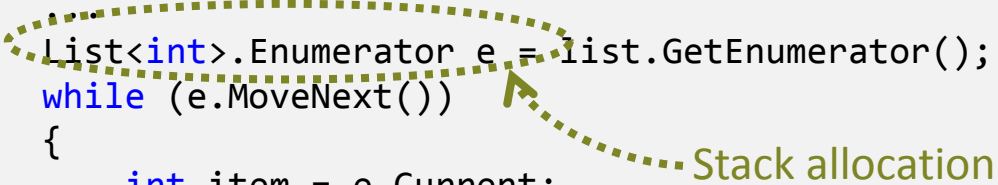
# Итератор-структура

```
var list = new List<int>();  
...  
foreach (var item in list)  
{  
    // do something  
}
```

---

```
var xList = (IList<int>)list;  
foreach (var xItem in xList)  
{  
    // do something  
}
```

```
var list = new List<int>();  
List<int>.Enumerator e = list.GetEnumerator();  
while (e.MoveNext())  
{  
    int item = e.Current;  
    // do something  
}
```



A dashed green oval highlights the line `List<int>.Enumerator e = list.GetEnumerator();`. A green arrow points from the text "Stack allocation" to the variable `e` within this line.

---

```
var xList = (IList<int>)list;  
IEnumerator<int> x = xList.GetEnumerator();  
while (x.MoveNext())  
{  
    int xItem = x.Current;  
    // do something  
}
```

# Итератор-структура

```
var list = new List<int>();  
...  
foreach (var item in list)  
{  
    // do something  
}
```

```
var list = new List<int>();  
List<int>.Enumerator e = list.GetEnumerator();  
while (e.MoveNext())  
{  
    int item = e.Current;  
    // do something  
}
```

Stack allocation

```
var xList = (IList<int>)list;  
foreach (var xItem in xList)  
{  
    // do something  
}
```

```
var xList = (IList<int>)list;  
IEnumerator<int> x = xList.GetEnumerator();  
while (x.MoveNext())  
{  
    int xItem = x.Current;  
    // do something  
}
```

Boxing!



# Итератор-структура

```
var list = new List<int>();  
...  
foreach (var item in list)  
{  
    // do something  
}
```

```
var list = new List<int>();  
List<int>.Enumerator e = list.GetEnumerator();  
while (e.MoveNext())  
{  
    int item = e.Current;  
    // do something  
}
```

Stack allocation

```
var xList = (IList<int>)list;  
foreach (var xItem in xList)  
{  
    // do something  
}
```

```
var xList = (IList<int>)list;  
IEnumerator<int> x = xList.GetEnumerator();  
while (x.MoveNext())  
{  
    int xItem = x.Current;  
    // do something  
}
```

Boxing!



Why do BCL Collections use struct enumerators, not classes? <http://bit.ly/2ADagls>

# План

- ▶ Классические коллекции
- ▶ Потокобезопасность
- ▶ Неизменяемость
- ▶ Собственные коллекции
- ▶ Коллекции в API

# Коллекции в API

- ▶ Для внутренних нужд классов в 99% случаев подойдет `List<T>` и другие обычные обобщенные коллекции
- ▶ Для внешнего API все иначе
  - ▶ `public members` – это тоже API
  - ▶ Позаботьтесь о пользователях!

# Принимаем коллекции

- ▶ `IEnumerable<T>` в аргументах

# Принимаем коллекции

- IEnumerable<T> в аргументах

```
private void Process(List<int> l)
{
    int s = 0;
    foreach (var x in l)
    {
        s += x.GetHashCode();
    }
}
```

(class) System.Collections.Generic.List<int>  
Represents a strongly typed list of objects that can be accessed individually by index.  
Parameter can be of type 'IEnumerable<int>'

# Принимаем коллекции

- ▶ IEnumerable<T> в аргументах

```
private void Process(List<int> l)
{
    int s = 0;
    foreach (var x in l)
    {
        s += x.GetHashCode();
    }
}
```

(class) System.Collections.Generic.List<int>  
Represents a strongly typed list of objects that c  
Parameter can be of type 'IEnumerable<int>'

- ▶ «Guidelines for Collections» от Microsoft тоже советуют
  - ▶ Но еще советуют использовать 'is' для проверки «а не Collection<T> ли это». См. <http://bit.ly/2xEbfEq>

# IEnumerable - проблемы

# IEnumerable - проблемы

- ▶ Несколько итераций

```
private void Process(IEnumerable<int> list)
{
    int sum = list.Sum();
    int max = list.Max();
}
```

(parameter) IEnumerable<int> list

Possible multiple enumeration of IEnumerable



# IEnumerable - проблемы

- ▶ Несколько итераций

```
private void Process(IEnumerable<int> list)
{
    int sum = list.Sum();
    int max = list.Max();
}
```

(parameter) IEnumerable<int> list

Possible multiple enumeration of IEnumerable

```
private void Process(IEnumerable<int> list)
{
    IEnumerable<int> enumerable = list as IList<int> ?? list.ToList();
    int sum = enumerable.Sum();
    int max = enumerable.Max();
}
```

# IEnumerable - проблемы

- ▶ Несколько итераций

```
private void Process(IEnumerable<int> list)
{
    int sum = list.Sum();
    int max = list.Max();
}
```

(parameter) IEnumerable<int> list

Possible multiple enumeration of IEnumerable

```
private void Process(IEnumerable<int> list)
{
    IEnumerable<int> enumerable = list as IList<int> ?? list.ToList();
    int sum = enumerable.Sum();
    int max = enumerable.Max();
}
```

- ▶ Защитная материализация

```
private void Process(IEnumerable<int> list)
{
    var l = list.ToList();
    int sum = l.Sum();
    int max = l.Max();
}
```

# ToList<T>() и ToArray<T>()

# ToList<T>() и ToArray<T>()

- ▶ Что выбрать?

# ToList<T>() и ToArray<T>()

- ▶ Что выбрать?
- ▶ Если не нужен конкретно массив, то однозначно ToList<T>()
- ▶ Причины
  - ▶ Хранится внутри всё равно массив
  - ▶ ToArray<T>() требует дополнительную аллокацию для возврата массива точного размера

# Принимаем коллекции - варианты

- ▶ Если не нужно изменять:
  - ▶ `IEnumerable<T>` - только итерирование

# Принимаем коллекции - варианты

- ▶ Если не нужно изменять:
  - ▶ `IEnumerable<T>` - только итерирование
  - ▶ `ReadOnlyCollection<T>` - итерирование + количество

# Принимаем коллекции - варианты

- ▶ Если не нужно изменять:
  - ▶ `IEnumerable<T>` - только итерирование
  - ▶ `ReadOnlyCollection<T>` - итерирование + количество
  - ▶ `ReadOnlyList<T>` - произвольный доступ



# Принимаем коллекции - варианты

- ▶ Если не нужно изменять:
  - ▶ `IEnumerable<T>` - только итерирование
  - ▶ `ReadOnlyCollection<T>` - итерирование + количество
  - ▶ `ReadOnlyList<T>` - произвольный доступ
- ▶ Если нужно изменять:
  - ▶ `IList<T>`

# Возвращаем коллекции

- ▶ Возвращать `IEnumerable<T>`

# Возвращаем коллекции

- ▶ Возвращать `IEnumerable<T>`
  - ▶ Одноразово-итерируемая структура

# Возвращаем коллекции

- ▶ Возвращать `IEnumerable<T>`
  - ▶ Одноразово-итерируемая структура
  - ▶ Iterator blocks ('yield')

# Возвращаем коллекции

- ▶ Возвращать IEnumerable<T>
  - ▶ Одноразово-итерируемая структура
  - ▶ Iterator blocks ('yield')
- ▶ Иначе – более специфическую коллекцию
  - ▶ ReadOnlyCollection<T>
  - ▶ Collection<T>
    - ▶ Или своего наследника

# Коллекции в API - выводы

- ▶ Принимаем интерфейсы, возвращаем конкретные коллекции
- ▶ Можно возвращать собственные коллекции-наследники
- ▶ `.ToList()` вместо `.ToArray()` если возможно

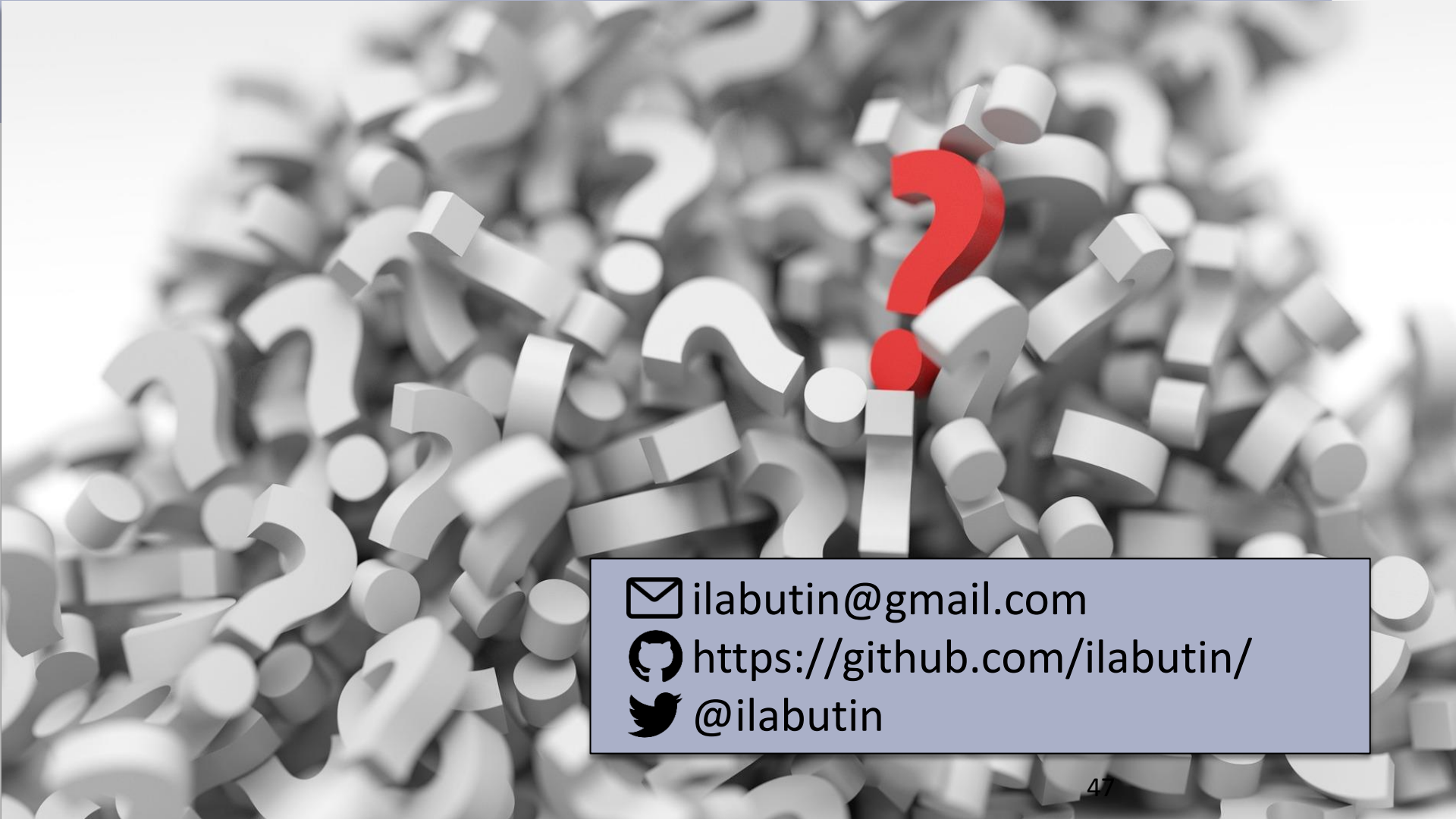
# Повторим

- ▶ Классические коллекции
- ▶ Потокобезопасность
- ▶ Неизменяемость
- ▶ Собственные коллекции
- ▶ Коллекции в API

# Ссылки

- ▶ Специализированные коллекции в Roslyn: <http://bit.ly/2wYE9ls>
- ▶ Рекомендации Microsoft по работе с коллекциями: <http://bit.ly/2xEbfEq>
- ▶ Сравнение производительности ConcurrentCollections и явных блокировок (PDF): <http://bit.ly/2vyv5DG>
- ▶ Серия статей от Eric Lippert про историю развития ко- и контравариантности в C#: <http://bit.ly/2xEiqMO>
- ▶ Why do BCL Collections use struct enumerators, not classes? <http://bit.ly/2ADagls>
- ▶ Примеры и бенчмарки из доклада: <https://github.com/ilabutin/spbdotnet2017/>





ilabutin@gmail.com



<https://github.com/ilabutin/>



@ilabutin