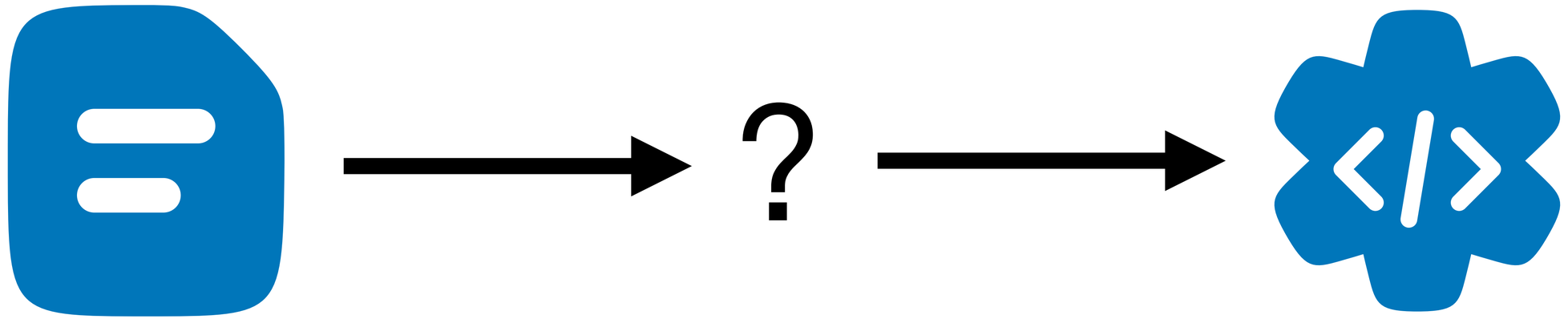
управление данными в микросервисах на C#

устройство .NET (1)

JIT компиляция, CIL, CLR, Lowering



КОМПИЛЯЦИЯ

- преобразование в машинный код до выполнения программы
- распространяются сразу бинарные пакеты
- требует отдельных сборок под различные таргеты
  - архитектуры процессоров
  - архитектуры операционных систем
  - разрядности операционных систем

интерпретация

- преобразование в машинный код происходит во время выполнения программы
- распространяется напрямую исходный код
- не требует сборки под различные таргеты (ведь сборки нет)
- пониженная производительность
  - парсинг кода
  - линковка
  - преобразование в машинный код

JIT компиляция

JIT – Just In Time

- преобразование в промежуточный формат до распространения
- преобразование в машинный код при выполнении
- во время выполнения не нужно выполнять
  - парсинг
  - линковку
- позволяет выполнять оптимизации под конкретный таргет

JIT компиляция

составляющие

- исходный код на языке программирования
- промежуточный код
- копилятор в промежуточный код
- среда выполнения промежуточного кода

JIT компиляция

составляющие

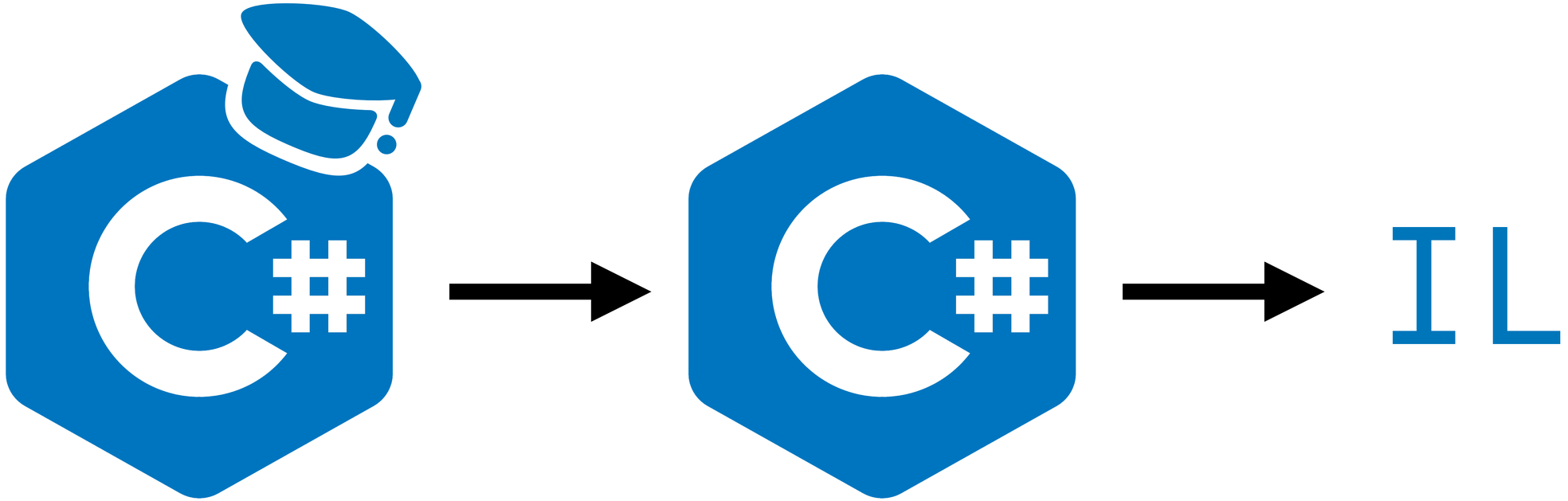
Составляющие	JVM	.NET
Язык	Java, Kotlin, Scala, …	C#, F#, VB.NET, …
Промежуточный код	JVM Byte code	CIL (Common Intermediate Language, IL)
Среда выполнения	JRE	CLR

CLR

среда выполнения

- JIT компиляция
- выделение памяти
- сборка памяти
- проверка типизаций
- загрузка методов, типов, сборок

lowering



lowering

extension methods

```
public static class MyIntegerExtensions
{
    public static int Doubled(this int value) ⇒ value * 2;
}
```

```
var value = 2;
var doubled = value.Doubled();
```

```
int value = 2;
int doubled = MyIntegerExtensions.Doubled(value);
```

lowering

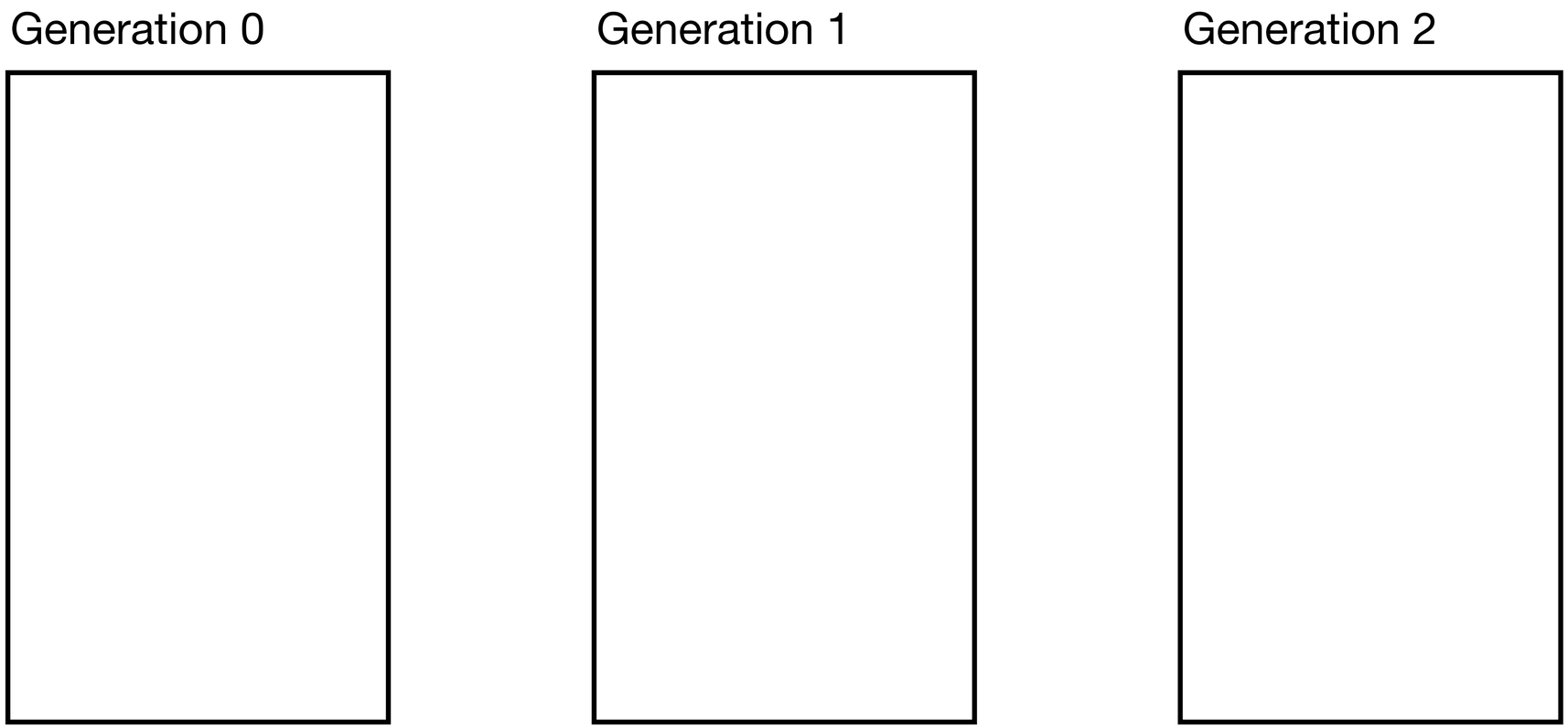
foreach

```
foreach (int value in enumerable)
{
    Console.WriteLine(value);
}
```

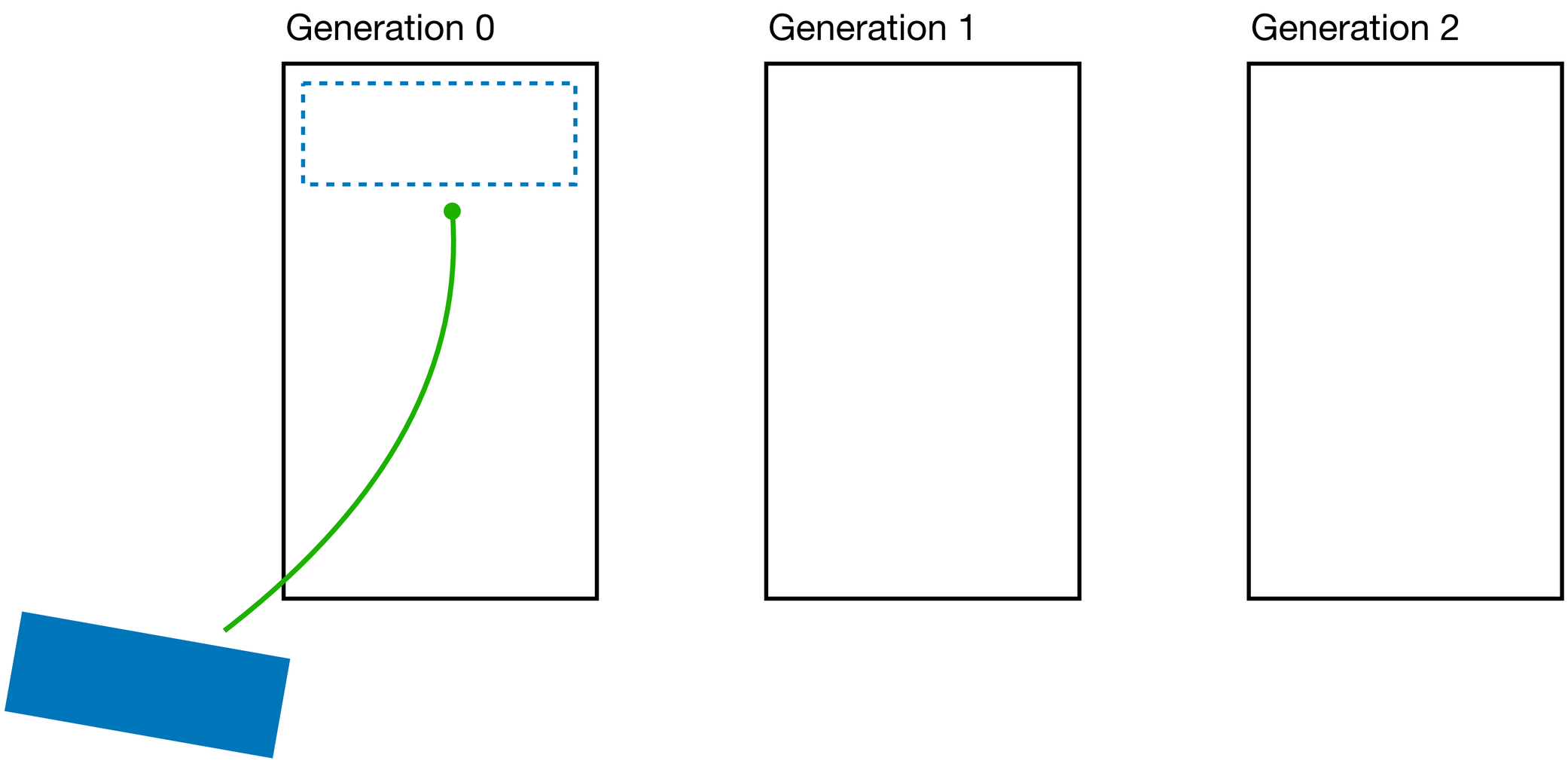
```
IEnumerator<int> enumerator = enumerable.GetEnumerator();

try
{
    while (enumerator.MoveNext())
        Console.WriteLine(enumerator.Current);
}
finally
{
    if (enumerator != null)
        enumerator.Dispose();
}
```

garbage collection

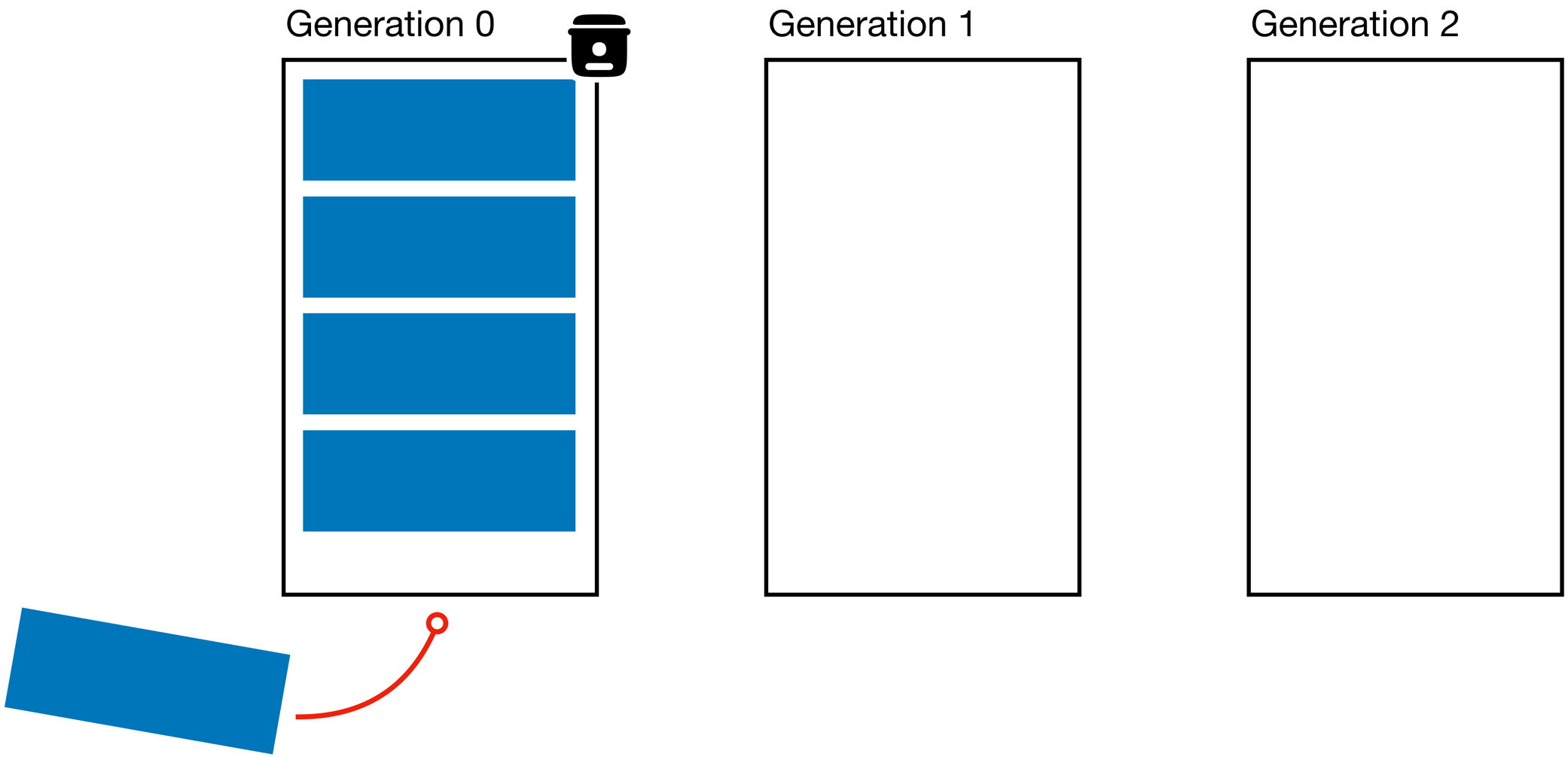
managed heap



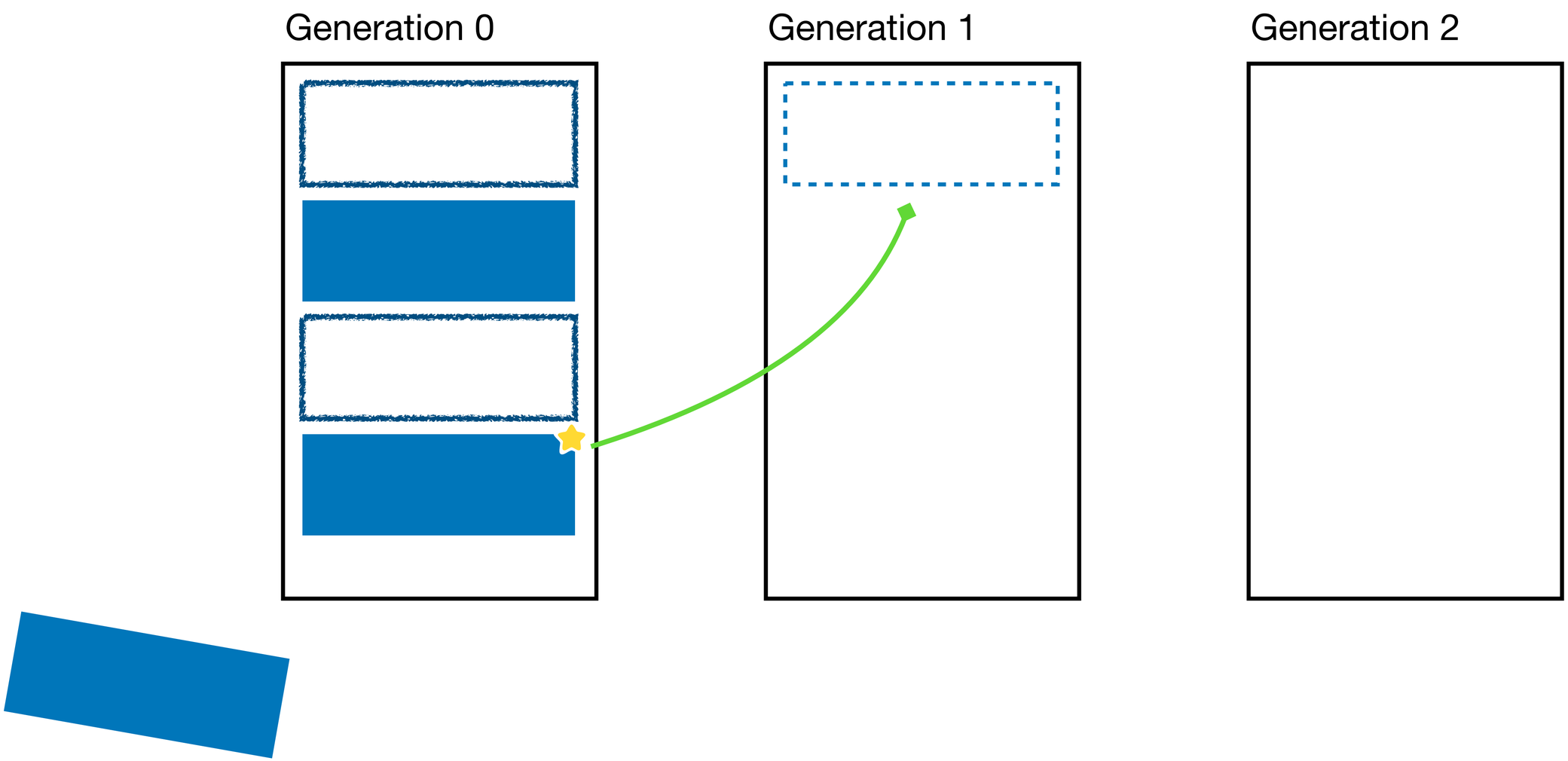
managed heap



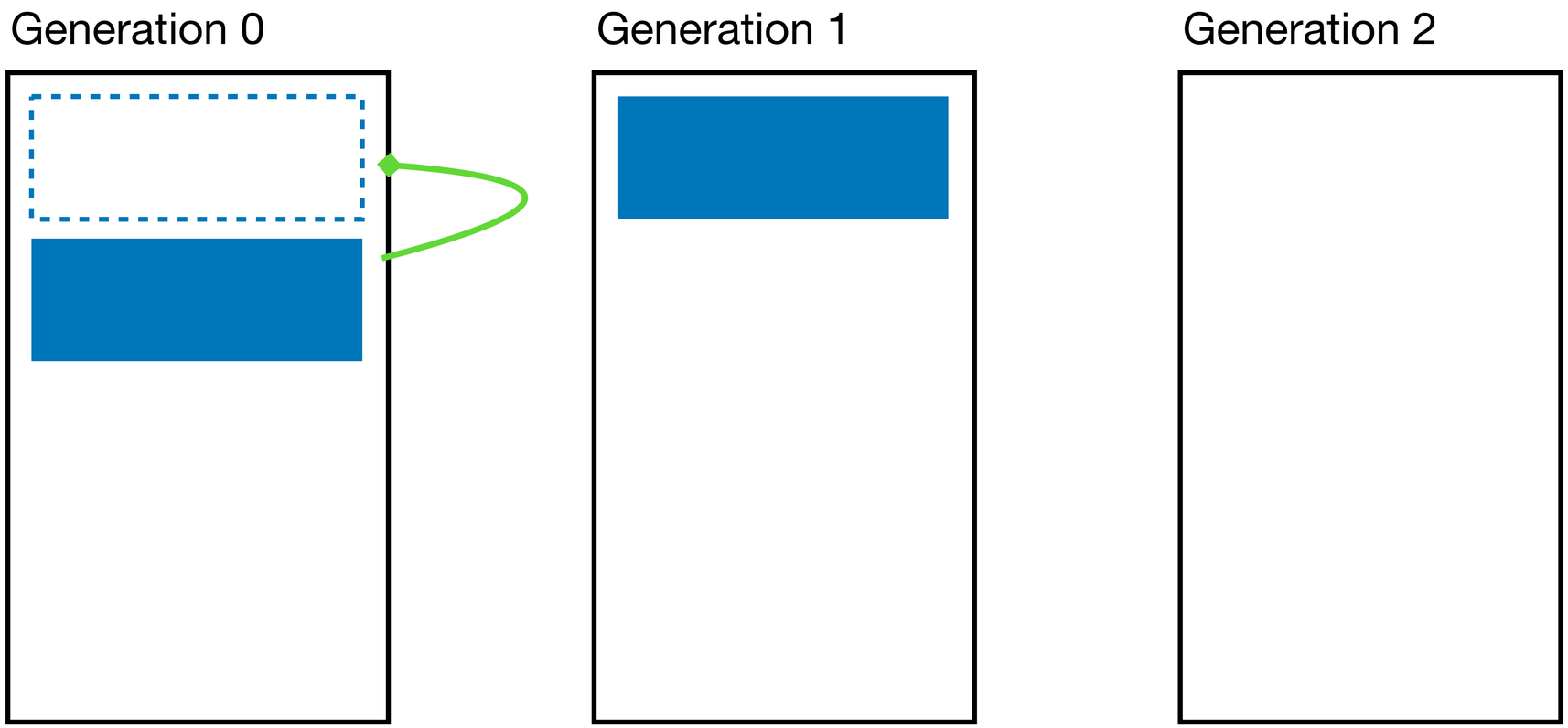
garbage collection

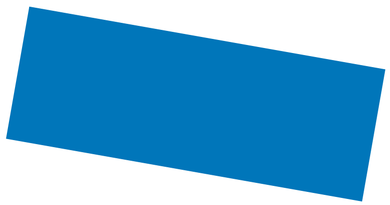


garbage collection

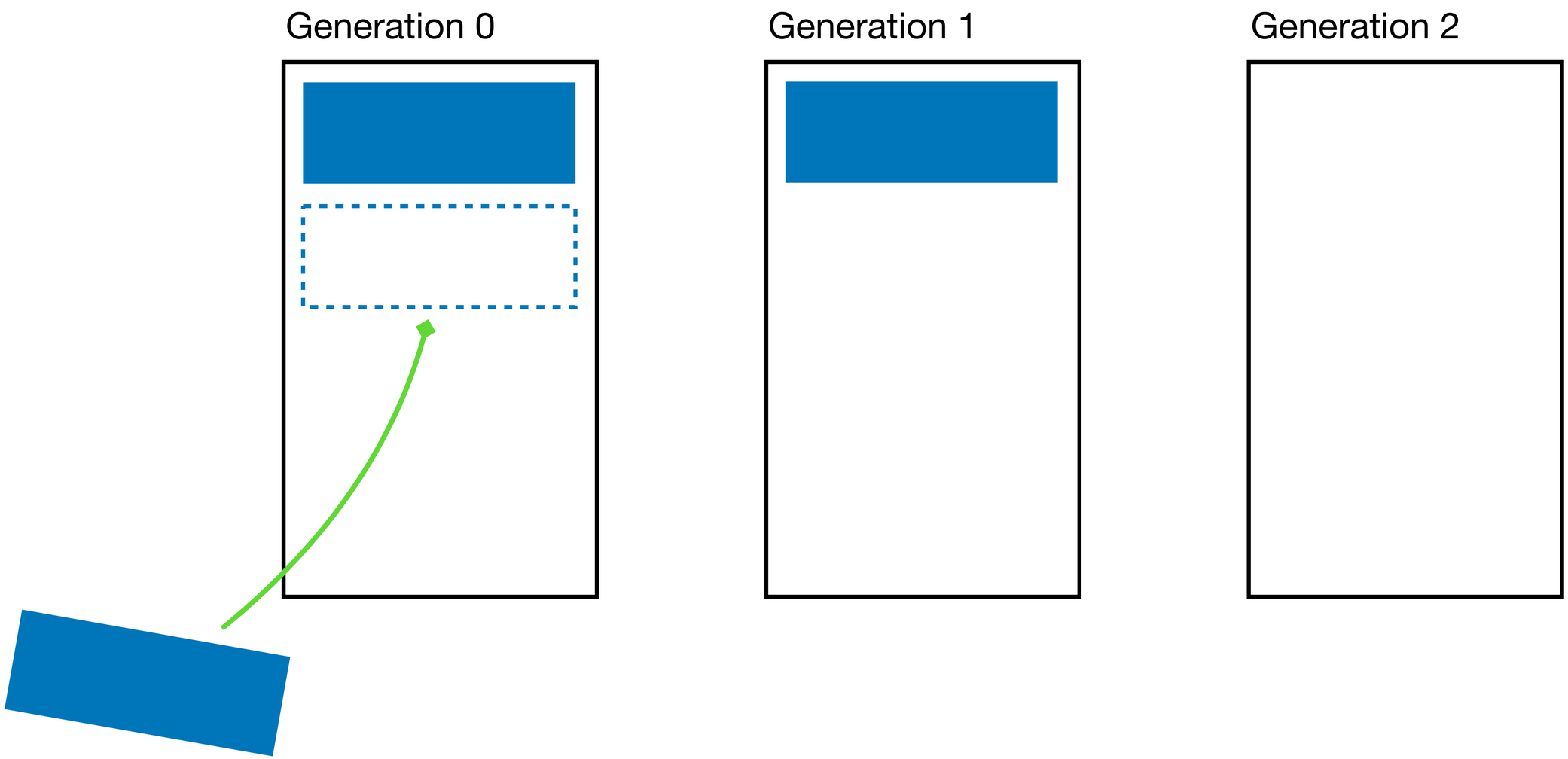


garbage collection

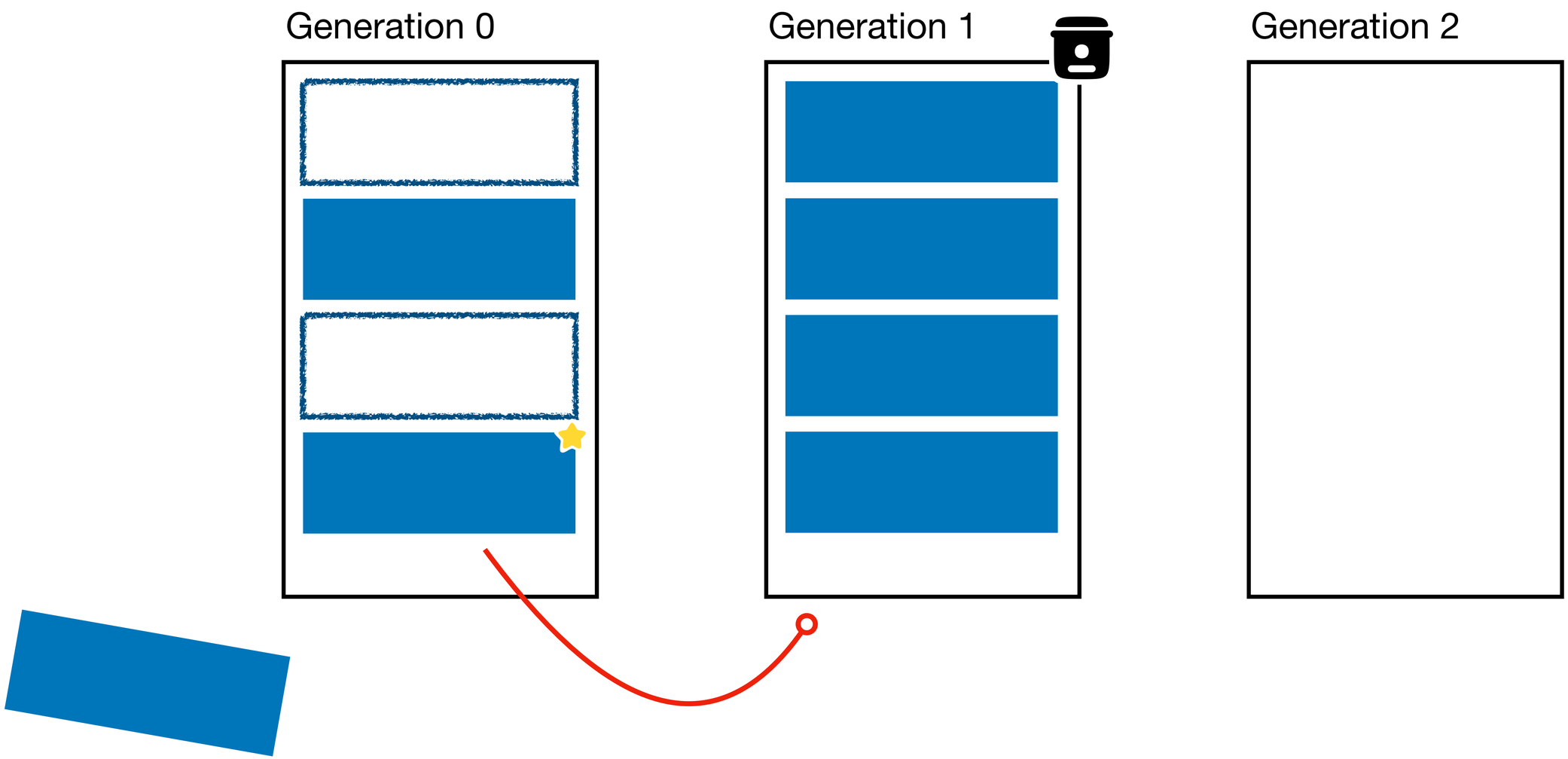




garbage collection



garbage collection



managed heap

large object heap

- отдельная куча для больших объектов
- на ней аллоцируются объекты более 85_000 байтов
- иногда называют Generation 3
- сборка происходит при сборке в Generation 2
- позволяет упростить уплотнение кучи

managed heap

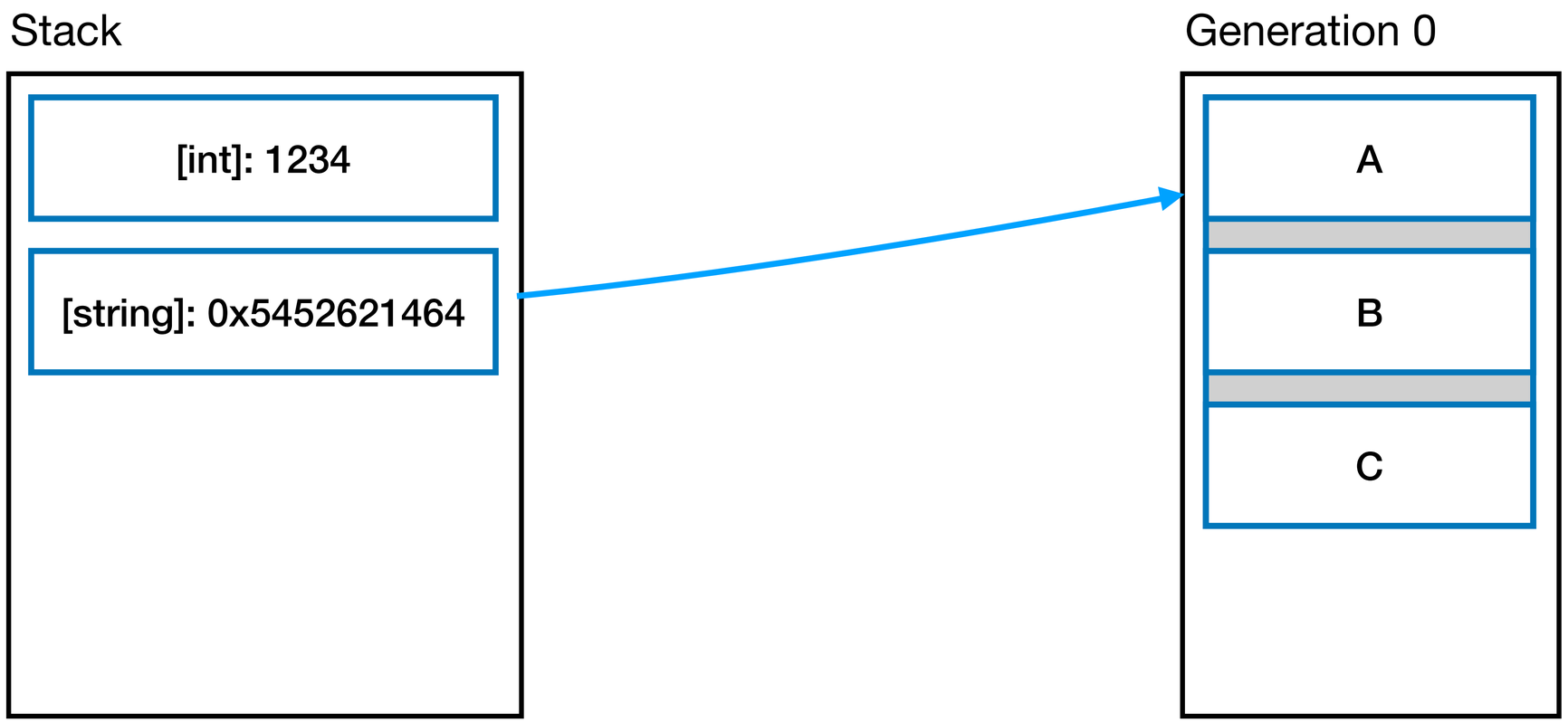
pinned object heap

- гарантировано долго живущие объекты
- интернированные строки
  - строковые константы
  - явно интернированные строки
- ключевое слово fixed
- такие объекты лежат отдельно чтобы не мешать сборке и уплотнению в других поколениях

boxing

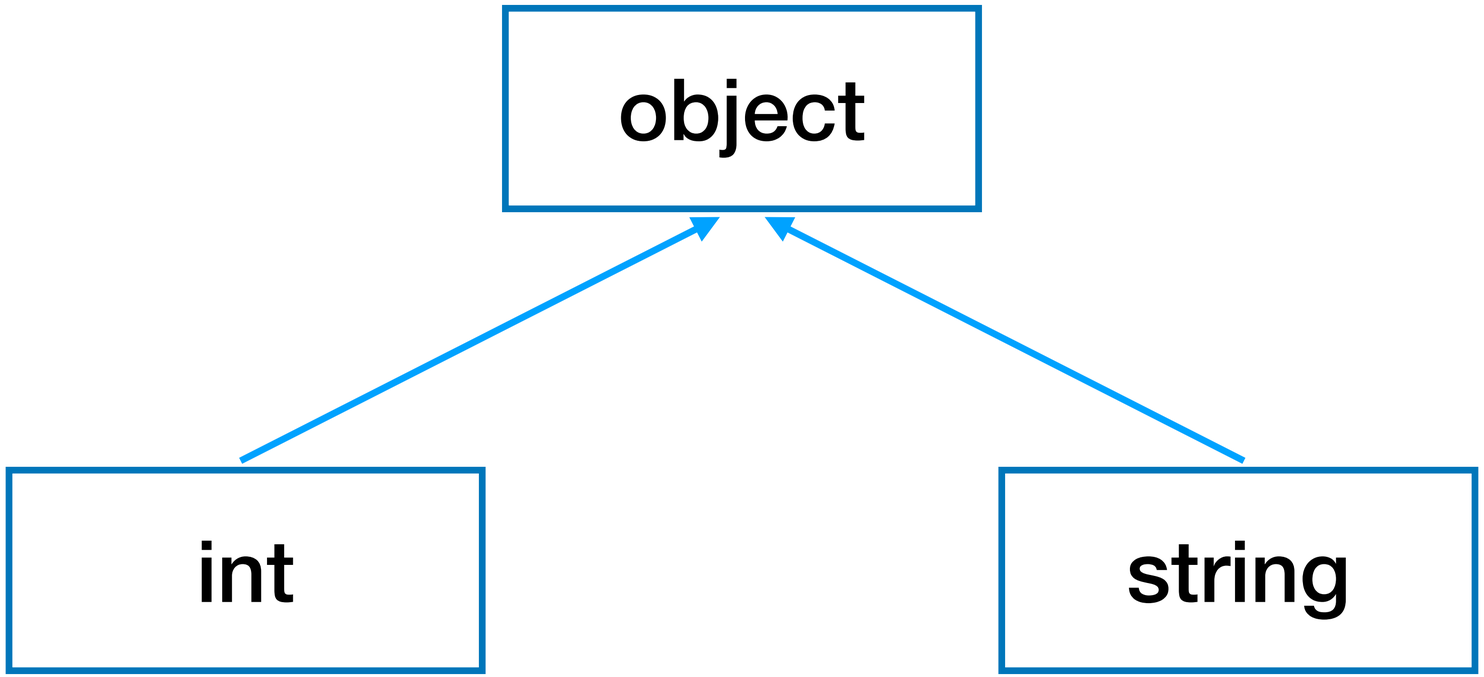
boxing

value vs reference types



boxing

type hierarchy



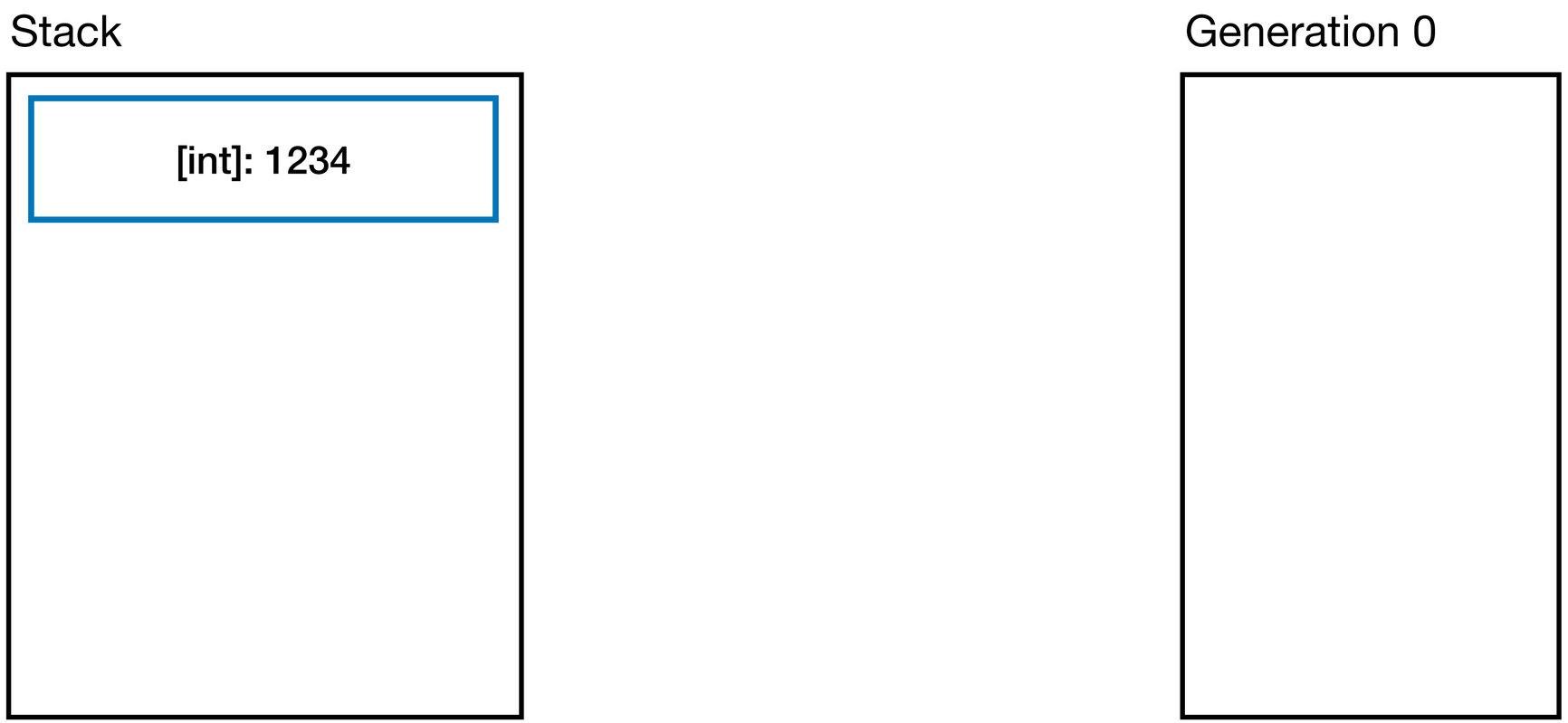
boxing

type hierarchy

```
Print(123);
Print("123");

static void Print(object obj)
{
    Console.WriteLine(obj);
}
```

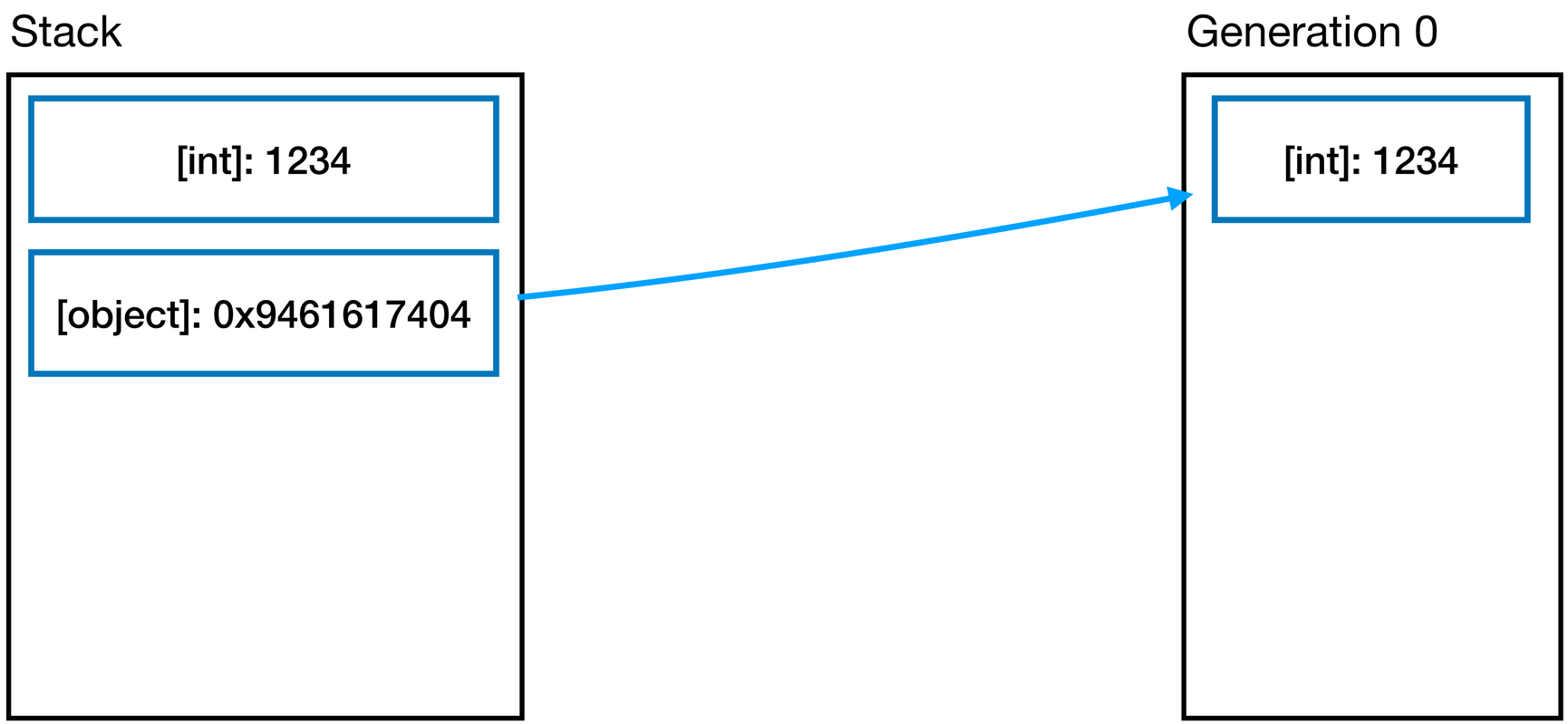
boxing

boxing



boxing

boxing



boxing

unboxing

```
int value = 123;
object boxed = Box(value);
int unboxed = (int)boxed;

static object Box<T>(T value)
    where T : struct
{
    return value;
}
```

генераторы, IEnumerable, LINQ, ленивые вычисления

IEnumerable

- представляет собой какой-то перечисляемый поток объектов
- позволяет получить однонаправленный итератор
- является базовым типов всех коллекций

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

LINQ

language integrated query

- методы расширения для работы с коллекциями
- методы LINQ, возвращающие IEnumerable<>, работают “лениво”

- материализованные коллекции – полностью хранятся в памяти
- не материализованные коллекции – в памяти хранится информация для их создания

Ленивые вычисления не всегда оптимальней по памяти

Некоторые методы LINQ под капотом могут аллоцировать коллекции

генераторы

IEnumerable

```
public IEnumerable<int> Range(int start, int count)
{
    for (int i = 0; i < count; i++)
    {
        yield return start + i;
    }
}
```

генераторы

IEnumerator

```
public interface IEnumerator<out T> : IDisposable
{
    T Current { get; }

    bool MoveNext();
}
```

IEnumerable

генераторы

```
foreach (int value in Range(0, 10))
{
    Console.WriteLine(value);
}
```

```
public IEnumerable<int> Range(int start, int count)
{
    for (int i = 0; i < count; i++)
    {
        yield return start + i;
    }
}
```

IEnumerable

генераторы

```
foreach (int value in MyGenerators.Range( start: 0,  count: 10))
{
    Console.WriteLine(value);
}
```

```
5       public static IEnumerable<int> Range(int start, int count)
4       {
3           for (var i = 0; i < count; i++)
2           {
1               yield return start + i;
10          }
1       }
```

IEnumerable

генераторы

```
foreach (int value in MyGenerators.Range( start: 0,  count: 10))
{
    Console.WriteLine(value);
}
```

```
1       public static IEnumerable<int> Range(int start, int count)
6 ➜     {  ≤ 3 ms elapsed
1           for (var i = 0; i < count; i++)
2           {
3               yield return start + i;
4           }
5       }
```

IEnumerable

генераторы

```
foreach (int value in MyGenerators.Range( start: 0,  count: 10))
{
    Console.WriteLine(value);
}
```

```
4       public static IEnumerable<int> Range(int start, int count)
3       {
2           for (var i = 0; i < count; i++)   i: 0    count: 10
1           {
9               yield return start + i;   ≤ 1 ms elapsed   i: 0    start
1           }
2       }
```

IEnumerable

генераторы

```
foreach (int value in MyGenerators.Range( start: 0,  count: 10))
{
    Console.WriteLine(value);  ≤ 1 ms elapsed
}
```

```
17  public static IEnumerable<int> Range(int start, int count)
16  {
15      for (var i = 0; i < count; i++)
14      {
13          yield return start + i;
12      }
11  }
```

IEnumerable

генераторы

```
foreach (int value in MyGenerators.Range( start: 0,  count: 10))
{
    Console.WriteLine(value);
}
```

```
20      public static IEnumerable<int> Range(int start, int count)
19      {
18          for (var i = 0; i < count; i++) ‹ +0 +1 ›   i: 0    coun
17          {
16              yield return start + i;   start: 0    i: 0
15          } ≤ 2 ms elapsed
14      }
```

IEnumerable

генераторы

```
foreach (int value in MyGenerators.Range( start: 0,  count: 10))
{
    Console.WriteLine(value);
}
```

```
18      public static IEnumerable<int> Range(int start, int count)
17      {
16          for (var i = 0; i < count; i++)  ≤ 1 ms elapsed  i: 0    c
15          {
14              yield return start + i;
13          }
12      }
```

IEnumerable

генераторы

```
foreach (int value in MyGenerators.Range( start: 0,  count: 10))
{
    Console.WriteLine(value);
}
```

```
18      public static IEnumerable<int> Range(int start, int count)
17      {
16          for (var i = 0; i < count; i++)   i: 1    count: 10
15          {
14              yield return start + i;  ≤ 1 ms elapsed  i: 1    start
13          }
12      }
```

IEnumerable

генераторы

```
foreach (int value in MyGenerators.Range( start: 0,  count: 10))
{
    Console.WriteLine(value);  ≤ 1 ms elapsed
}
```

```
21  public static IEnumerable<int> Range(int start, int count)
20  {
19      for (var i = 0; i < count; i++)
18      {
17          yield return start + i;
16      }
15  }
```

многопоточка в .NET

многопоточка в .NET

потоки

- представлены классом Thread
- привязаны к потокам операционной системы

```
var thread = new Thread(() ⇒ Console.WriteLine("Hello from another thread!"));
thread.Start();

thread.Join();
```

многопоточка в .NET

класс Parallel

- Предоставляет методы для параллельного выполнения кода
- Parallel.For
- Parallel.ForEach
- Parallel.Invoke

класс Parallel

For и ForEach

```
Parallel.For(0, 10, i ⇒ Console.WriteLine(i));
Parallel.ForEach(Enumerable.Range(0, 10), i ⇒ Console.WriteLine(i));
```

класс Parallel

Invoke

```
Parallel.Invoke(
    () ⇒ Console.WriteLine(1),
    () ⇒ Console.WriteLine(2),
    () ⇒ Console.WriteLine(3));
```

класс Parallel

ParallelOptions

```
var options = new ParallelOptions
{
    CancellationToken = cancellationToken,
    MaxDegreeOfParallelism = 10,
};
```

класс Parallel

ParallelLoopResult

```
var options = new ParallelOptions { CancellationToken = cancellationToken };
ParallelLoopResult result = Parallel.For(0, 10, options, i ⇒ Console.WriteLine(i));

if (result.IsCompleted is false)
{
    Console.WriteLine("Failed to complete parallel operation");
}
```

многопоточка в .NET

PLINQ

- позволяет преобразовать обычный IEnumerable в параллельный
- дублирует большинство методов LINQ
- выполняет операции лениво в несколько потоков

```
ParallelQuery<TSource> AsParallel<TSource>(this IEnumerable<TSource> source)
```

```
int[] hashes = Enumerable
    .Range(0, 100)
    .AsParallel()
    .Select(x ⇒ ComputeHashExpensive(x))
    .ToArray();
```

примитивы синхронизации в .NET

примитивы синхронизации в .NET

lock

- встроен в функционал языка
- аналог в других языках – mutex
- реализует блокирование на какой-либо объект
- записывает идентификатор потока в sync-block объекта

примитивы синхронизации в .NET

lock

```
var lockKey = new object();

lock (lockKey)
{
    Console.WriteLine("Hello from locked code!");
}
```

```
object obj = new object();
bool lockTaken = false;

try
{
    Monitor.Enter(obj, ref lockTaken);
    Console.WriteLine("Hello from locked code!");
}
finally
{
    if (lockTaken)
        Monitor.Exit(obj);
}
```

примитивы синхронизации в .NET

lock

```
var lockKey = 123;

lock (lockKey)
{
    Conso
}
```

'int' is not a reference type as required by the lock statement

local variable int **lockKey**

примитивы синхронизации в .NET

lock

```
lock ("string literal lock key")
{
    Console.WriteLine("Hello from locked code!");
}
```

примитивы синхронизации в .NET

Semaphore и SemaphoreSlim

- реализованы через счётчик пропущенных потоков
- позволяют допускать к критической секции более одного потока
- нет завязки на идентификатор потока
- Slim версия не уходит в блокировку потока сразу,
  а выполняет SpinWait какое-то время

примитивы синхронизации в .NET

Semaphore и SemaphoreSlim

```
var semaphore = new SemaphoreSlim(initialCount: 1, maxCount: 1);

semaphore.Wait();

try
{
    Console.WriteLine("Hello from semaphore locked code");
}
finally
{
    semaphore.Release();
}
```

потокобезопасные коллекции в .NET

потокобезопасные коллекции в .NET

System.Collections.Concurrent

- ConcurrentDictonary<TKey, TValue>
- ConcurrentBag<T>
- ConcurrentQueue<T>
- ConcurrentStack<T>