Сборка мусора

- **Жизненный цикл объектов.**
- **Понятие сборщика мусора.**
- **Метод Finalize.**
- **Метод Dispose и интерфейс IDisposable.**
- **Класс System.GC.**
- **Понятие поколений при сборке мусора.**

Жизненный цикл любого объекта можно представить следующим образом:

1. выделение памяти для объекта;
2. инициализация выделенной памяти
   2.1. установка объекта в начальное значение;
   2.2. вызов конструктора;
3. использование объекта в программе;
4. разрушение состояния ресурса;
5. освобождение занятой памяти (GC).

Команда C# new (MSIL newobj) создает объект.

После получения этой команды CLR:

1. Подсчитывает количество байтов, необходимых для размещения полей объекта в управляемой куче (включая память, необходимую для членов данных и базовых классов).
2. Прибавляет к полученному значению количество байтов, необходимых для размещения системных полей объекта.
3. Если в управляемой куче достаточно места для объекта, ему выделяется память, начиная с адреса, на который ссылается «указатель на следующий объект», а занимаемые им байты обнуляются.
4. Вызывается конструктор типа для инициализации, а команда *new* возвращает адрес объекта в управляемой куче. При этом происходит смещение «указателя на следующий объект».

Указатель на следующий объект (NextObjPtr) настраивается так, чтобы указывать на следующий доступный участок памяти.

Разница между управляемой и неуправляемой «кучей»

В «неуправляемой» куче выделение памяти для объекта означает поиск свободного блока достаточного размера и размещение его там.

В «управляемой» куче выделение памяти для объекта означает последовательное размещение объекта и прибавление некоторого значения к указателю на следующий объект.

В «неуправляемой» куче память для объектов выделяется в любой свободной области. Поэтому вполне вероятно, что несколько последовательно созданных объектов окажутся разделенными мегабайтами адресного пространства.

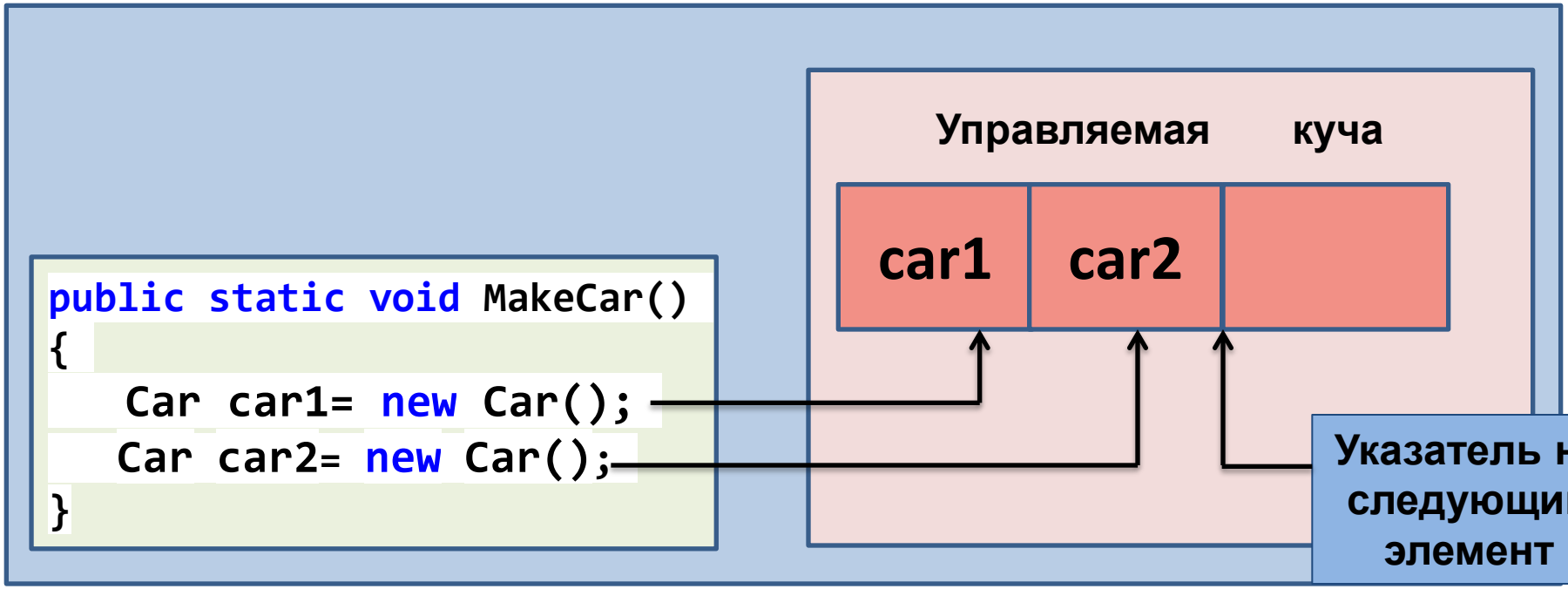
В «управляемой» куче последовательно созданные объекты гарантированно будут расположены друг за другом.

При создании приложений на C# можно смело полагать, что исполняющая среда .NET будет сама заботиться об управляющей куче без непосредственного вмешательства со стороны программиста.

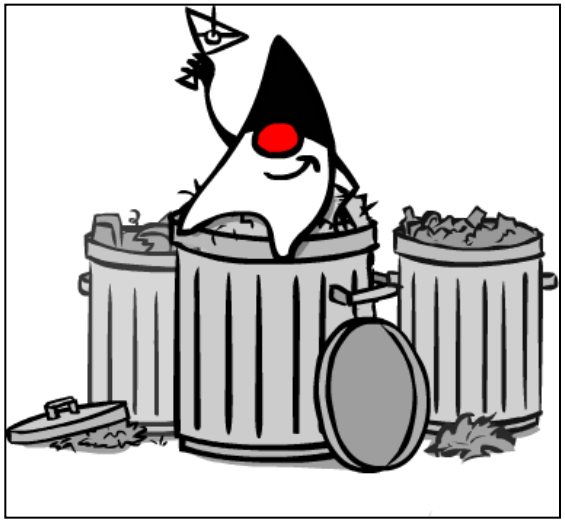
«Золотое правило» по управлению памятью :

Размещайте объект в управляющей куче с помощью ключевого слова new и забывайте об этом.

```
 public static void MakeCar()
{
  // Если car1 является единственной ссылкой на объект типа Car,
  // тогда после выполнения метода MakeCar,
  // объект myCar «может» быть уничтожен.
   Car car1= new Car();
   Car car2= new Car();
}
```



Специальный механизм, называемый ***сборщиком мусора (garbage collector)***, периодически освобождает память, удаляя объекты, которые уже не будут востребованы приложением.



Сборка мусора была впервые применена Джоном Маккарти в 1959 году в среде программирования на разработанном им функциональном языке программирования Lisp. В последствии она применялась в других системах программирования и языках, преимущественно - в функциональных и логических.

Сборщик мусора инициирует процесс сборки при:

- распределении памяти (посредством ключевого слова new) – только в случае недостатка памяти;
- при вызове сборщика мусора вручную, вызвав System.GC. Collect();
- Windows сообщает о нехватке памяти;
- после завершения работы CLR;

Общая схема сборки мусора состоит из следующих шагов:

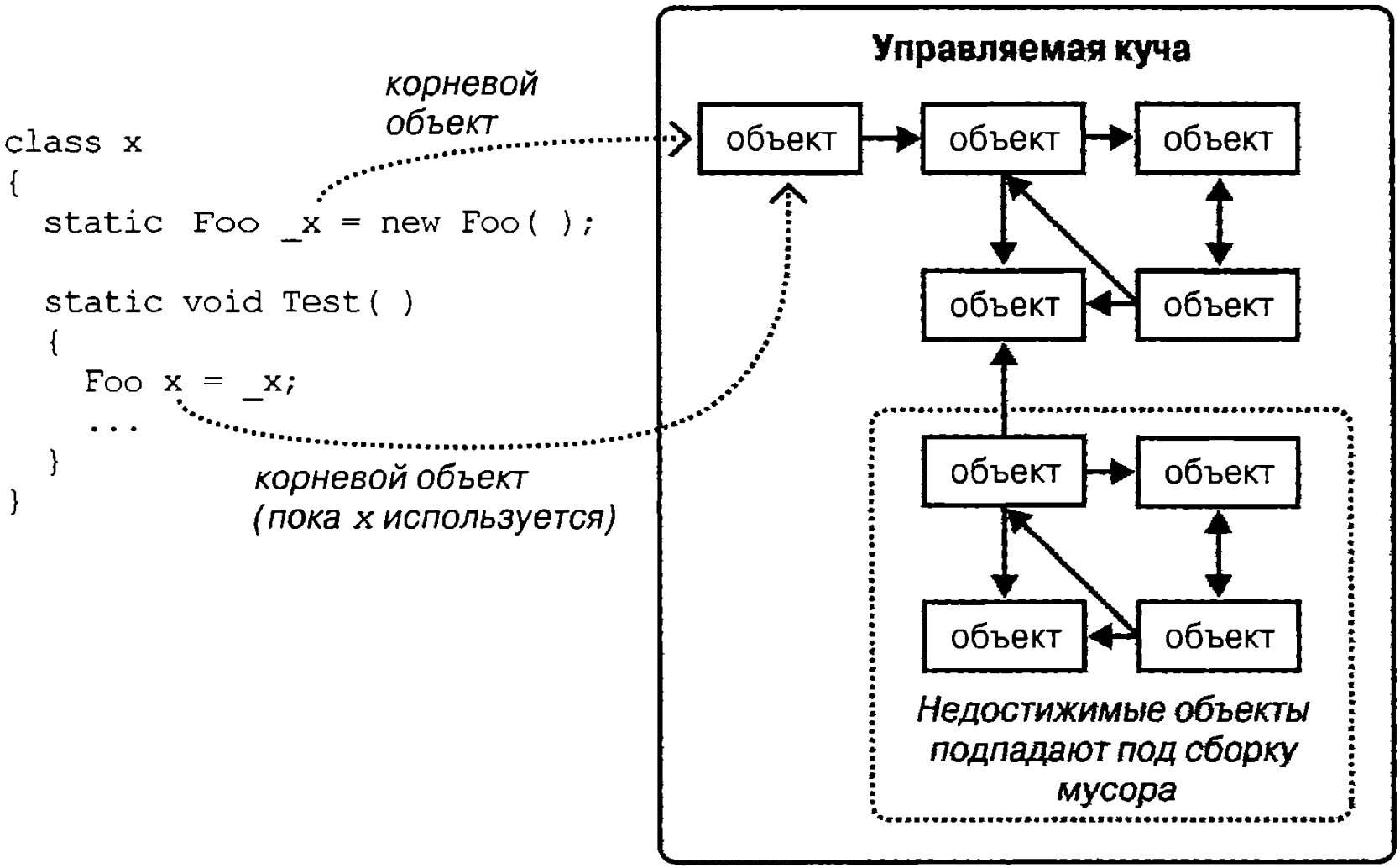
1. Сборщик мусора **осуществляет поиск объектов**, на которые есть ссылки в управляемом коде.
2. Во время сборки мусора сборщик не освобождает объект, если он еще используется приложением, т.е. **достижим**. Объект считается достижимым, если сборщик **находит хотя бы одну ссылку на него в управляемом коде**.
3. Сборщик мусора **освобождает объекты, на которые нет ссылок**, и высвобождает выделенную им память.

***Корневым элементом* (root)** называется ячейка в памяти, в которой содержится ссылка на размещающийся в куче объект.

Корнями могут быть только ссылочные типы.

К корневым элементам относятся:

- Ссылки на любые статические поля.
- Ссылки на локальные объекты.
- Ссылки на передаваемые методу параметры.
- Ссылки на объекты, ожидающие финализации.



Объекты, которые **не могут быть доступны путем следования по стрелкам (ссылкам) из корневого объекта**, являются недостижимыми и, таким образом, **подпадают под сборку мусора**

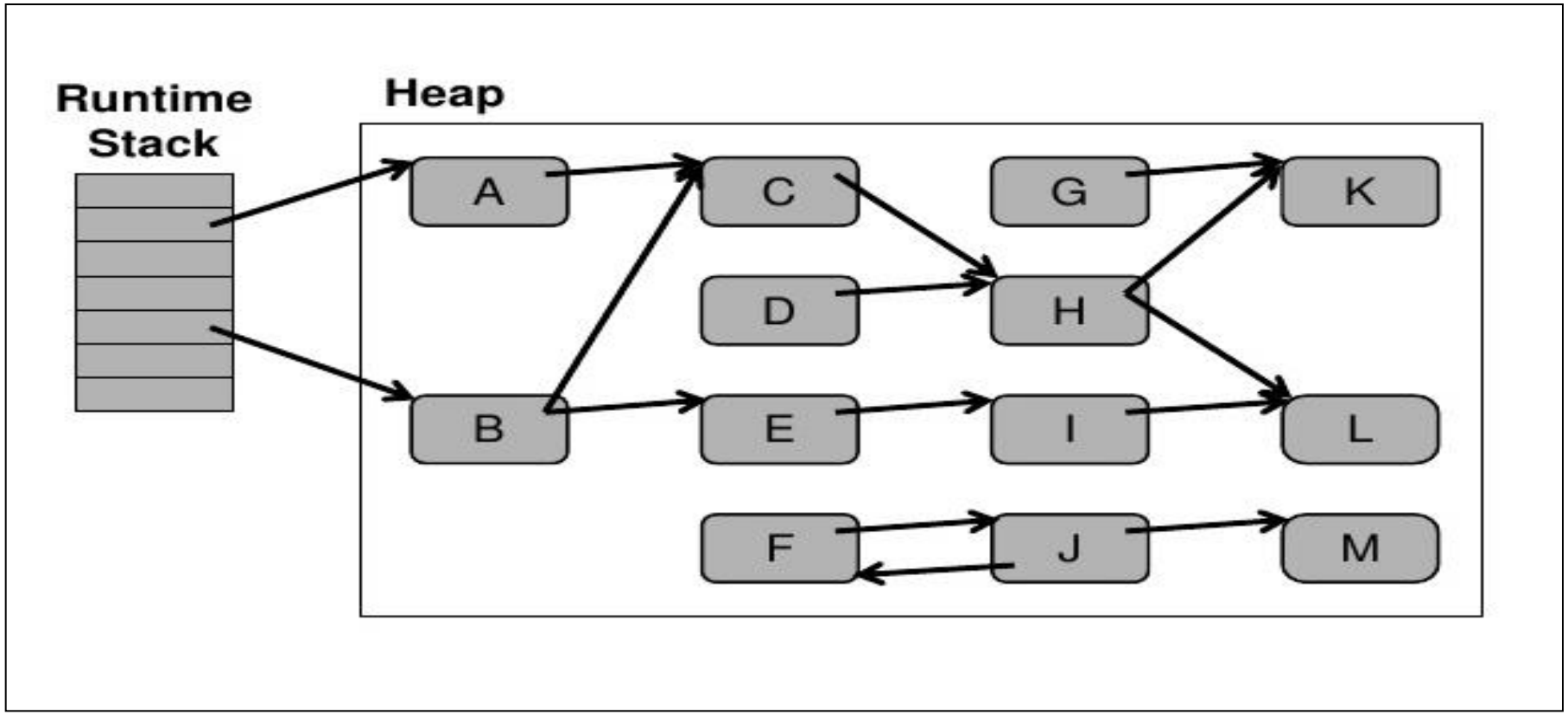
Сборщик мусора работает в режимах «**маркировки**» и «**сжатия**»

Сборщик мусора считается отслеживающим в том, что **он не вмешивается в каждый доступ к объекту**, а вместо этого:

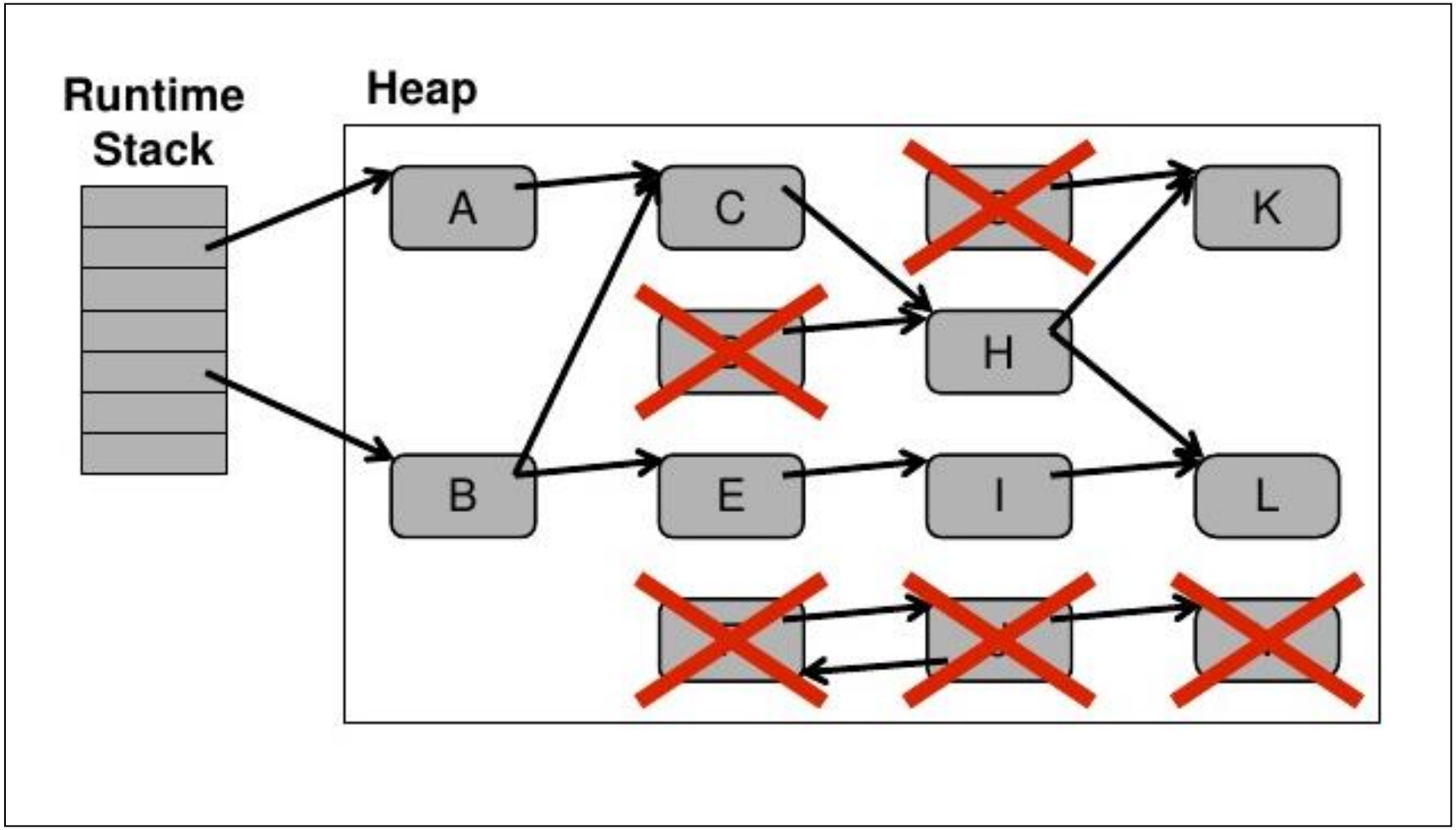
- **активизируется периодически**;
- **отслеживает граф объектов**, хранящихся в управляемой куче, с целью определения объектов, которые могут быть собраны сборщиком мусора.

Режим «маркировки».

Сборщик мусора **начинает со ссылок на корневые объекты** и проходит **по графу объектов**, помечая все затрагиваемые им объекты, как **достижимые**. Как только этот процесс завершен, все объекты, которые **не были помечены**, считаются неиспользуемыми и **подпадают под сборку мусора**.



A	B	C	D	E	F	G	H	I	J	K	L	M

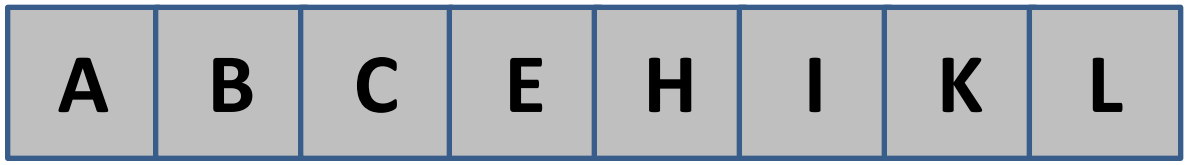


A	B	C	D	E	F	G	H	I	J	K	L	M

Режим «сжатия».

Оставшиеся **активные объекты затем сдвигаются** в начало кучи, освобождая пространство под дополнительные объекты. Такое сжатие служит двум целям:

- оно устраняет фрагментацию памяти;
- позволяет сборщику мусора применять очень простую стратегию при распределении новых объектов, для которых всегда выделяется память в конце кучи.



Для **оптимизации сборки мусора** использует :

- набор поколений;
- специальную «кучу» для массивных объектов;
- параллельную и фоновую сборки мусора;
- уведомления о сборке мусора.

Поколение 0. Идентифицирует новый размещенный объект, который еще никогда не помечался как подлежащий сборке мусора.

пороговое значение от 256 Кбайт – 4Мбайт

Поколение 1. Идентифицирует объект, который уже пережил **один** процесс сборки мусора (т.е. был помечен как подлежащий сборке мусора, но не был удален из-за наличия достаточного пространства в куче).

пороговое значение 2 Мбайт

Поколение 2. Идентифицирует объект, которому удалось пережить более одного прохода сборщика мусора.

пороговое значение около 10 Мбайт

На объекты B, C и E нет ссылок

Попытка выделить под новый объект больше памяти, чем есть в наличии

Метод	Описание
MaxGeneration	Возвращает информацию о том, сколько максимум поколений поддерживается в целевой системе. В .NET 4.0 поддерживается всего три поколения: 0, 1 и 2
GetGeneration(object ob)	Возвращает информацию о том, к какому поколению в настоящий момент относится объект
Collect()	Заставляет сборщик мусора провести сборку мусора. Должен быть перегружен так, чтобы указывать, объекты какого поколения подлежат сборке, а также какой режим сборки использовать (с помощью перечисления GCCollectionMode)
GetTotalMemory()	Возвращает информацию о том, какой объем памяти (в байтах) в настоящий момент занят в управляемой куче. Булевский параметр указывает, должен ли вызов сначала дождаться выполнения сборки мусора, прежде чем возвращать результат
CollectionCount()	Возвращает числовое значение, показывающее, сколько раз объектам данного поколения удалось переживать процесс сборки мусора
KeepAlive()	Создает ссылку на объект, защищая его от "сборки мусора". Действие этой ссылки оканчивается после выполнения метода KeepAlive()
RemoveMemoryPressure()	Задает в качестве параметра количество байтов, освобождаемых в неуправляемой области памяти
SuppressFinalize()	Позволяет устанавливать флаг, указывающий, что для данного объекта не должен вызываться его метод Finalize()

```
// Вспомогательный класс для создания мусора
class GarbageHelper
{  // Метод, создающий мусор
   public void MakeGarbage()
   {   for (int i = 0; i < 1000; i++)
        {
                Person p = new Person();
        }
}}

class Person
{    private string _name;
     private string _surname;
     private byte _age;
     public Person(string name, string surname, byte age)
      {   this._age = age;
          this._name = name;
          this._surname = surname;
      }
     public Person() : this("", "", 0)
      {
      }
}
```

```
    Console.WriteLine("Максимальное поколение: {0}", GC.MaxGeneration);
// создаем объект класс для создания "мусора"
    GarbageHelper hlp = new GarbageHelper();
// Узнаем поколение, в котором находится объект
    Console.WriteLine("Поколение объекта: {0}", GC.GetGeneration(hlp));
// Количество занятой памяти
    Console.WriteLine("Занято памяти (байт): {0}",GC.GetTotalMemory(false));
// Создаем мусор
    Console.WriteLine("Создаем мусор");  hlp.MakeGarbage();
// Количество занятой памяти
    Console.WriteLine("Занято памяти(байт):{0}", GC.GetTotalMemory(false));
// Вызываем явный сбор мусора в первом поколении
    Console.WriteLine("Вызываем сбор мусора в 1 поколении Collect(0)");
    GC.Collect(0);
// Количество занятой памяти
    Console.WriteLine("Занято памяти (байт): {0}", GC.GetTotalMemory(false));
// Узнаем поколение, в котором находится объект
    Console.WriteLine("Поколение объекта: {0}",   GC.GetGeneration(hlp));
// Вызываем явный сбор мусора во всех поколениях
    Console.WriteLine("Вызываем сбор мусора поколениях Collect()");
    GC.Collect();
// Количество занятой памяти
    Console.WriteLine("Занято памяти (байт): {0}", GC.GetTotalMemory(false));
// Узнаем поколение, в котором находится объект
    Console.WriteLine("Поколение объекта: {0}", GC.GetGeneration(hlp));
```

```
Демонстрация System.GC
Максимальное поколение: 2
Поколение объекта: 0
Занято памяти (байт): 195332
Создаем мусор
Занято памяти (байт): 211716
Вызываем явный сбор мусора в первом поколении Collect(0)
Занято памяти (байт): 97796
Поколение объекта: 1
Вызываем явный сбор мусора во всех поколениях Collect()
Занято памяти (байт): 97644
Поколение объекта: 2
```

Финализируемые объекты

- Класс System.Object – содержит виртуальный метод - Finalize();

```
public class Object
 {
  protected virtual void Finalize () {}
 }
```

- Переопределяя метод Finalize() специальном образом устанавливается специфическое место для выполнения любой необходимой данному типу логики по очистке (освобождению).

```
~FinalizeObject()
   {
    // ...
   }
```

Особенности:

- Переопределять метод Finalize() в типах структур нельзя.
- Вызов метода Finalize() будет происходить либо во время естественной активизации процесса сборки мусора, либо во время его принудительной активизации программным образом с помощью GC.Collect().

Особенности удаления сборщиком мусора объектов, имеющих финализаторы.

1. Сборщик мусора идентифицирует неиспользуемые объекты, готовые к удалению. Объекты, которые не имеют финализаторов, удаляются сразу. Объекты, которые имеют отложенные финализаторы, сохраняются в активном состоянии и помещаются в специальную очередь.
2. Сборка мусора завершена, а программа продолжает свое выполнение. Далее параллельно программе начинает выполняться поток финализаторов, выбирая объекты из этой специальной очереди и запуская их методы финализации.
3. После того, как объект извлечен из очереди, а его финализатор выполнен, объект становится висячим и будет удален при следующей сборке мусора

Финализаторы могут быть удобны, однако с ними связаны некоторые особенности:

- Финализаторы замедляют выделение и утилизацию памяти (сборщик мусора должен отслеживать, какие финализаторы были запущены).
- Финализаторы продлевают время жизни объекта и любых объектов, которые на него ссылаются (они вынуждены ожидать действительного удаления при очередной сборке мусора).
- Порядок вызова финализаторов для набора объектов предсказать невозможно.
- Имеется только ограниченный контроль над тем, когда будет вызван финализатор того или иного объекта.
- Если код в финализаторе приводит к блокировке, другие объекты не смогут выполнить финализацию.
- Финализаторы могут вообще не запуститься, если приложение не смогло быть выгружено чисто.

```
class Car
{ public string Name {get; set;}
  public Car(string name)
    {
     this.name = name;
    }
   ~ Car()  // метод Finalize()
    {
      Console.WriteLine("Вызов Finalize() Car.Name={0}", name);
    }
}
```

```
Пример использования Finalaze()
BMW
Ford
Вызов деструктора Car.Name=Ford
Вызов деструктора Car.Name=BMW
```

```
class Program
{ static void Main(string[] args)
  {
     Car car = new Car("BMW");
     Console.WriteLine(car.Name);
     Test();
     GC.Collect();  // ЯВНАЯ СБОРКА МУСОРА
  }
  static void Test()
  {   Car car = new Car("Ford");
      Console.WriteLine(car.Name);
}}
```

Создание освобождаемых объектов

В качестве альтернативы для **освобождения неуправляемых ресурсов** переопределению Finalize() классы могу реализовать **интерфейс IDisposable**

```
public interface IDisposable
{
    void Dispose();
}
```

- Использование IDisposable предполагеат, что после завершения работы с объектом метод Dispose() должен **вручную вызываться**, прежде чем объектной ссылке будет разрешено покинуть область действия.
- Интерфейс IDisposable может быть реализован в **классах и в структурах** (в отличие от метода Finalize (), который допускается переопределять только в классах.

```
class Car : IDisposable
// наследование от интерфейса IDisposable, содержащего метод Dispose
{
 public string Name { get; set; }
 public void Dispose()  // обязательная реализация метода Dispose()
 {
 Console.WriteLine("Вызов метода Dispose() для  Car.Name={0}", Name);
}}
```

```
Car car1 = new Car() { Name="BMW"};
Car car2 = new Car() { Name = "Ford" };

car1.Dispose();
// пауза вычислительного процесса на 3000 мс (3 с.)
Thread.Sleep(3000);
car2.Dispose();
```

```
Пример использования Dispose()
Вызов метода Dispose() для  Car.Name=BMW
Вызов метода Dispose() для  Car.Name=Ford
```

Различные формы реализации IDisposable

```
class MyResource : IDisposable
{
    string name;
    public MyResource(string name)
    {
        this.name = name;
    }
    public void Dispose()
    {
        // Освобождение неуправляемых ресурсов. . .
        // Только для целей тестирования.
        Console.WriteLine("***** In Dispose для {0}! ***** ", name);
    }
}
```

```
MyResource res1 = new MyResource("res1");
res1.Dispose();

MyResource res2 = new MyResource("res2");
if (res2 is IDisposable)
    res2.Dispose();

MyResource res3 = new MyResource("res3");
try
{  // Использование res3
}
finally
{   // Обеспечение вызова метод Dispose() в любом случае,
    //в том числе и при возникновении Exception.
    res3.Dispose();
}

//   использование using для реализации try/finally
using (MyResource res4 = new MyResource("res4"))
{
    // Использование res4
}
```

```
//   использование using для реализации try/finally
     //   только, если MyResource реализует IDisposable
     //   Dispose вызывается при выходе за пределы using.
using (MyResource res4 = new MyResource("res4"),
                                        res5 = new MyResource("res5"))
{
    // Использование res4 и res5.
}
```

Формализованный шаблон освобождения

```
public class MyResource : IDisposable
{  // Используется для контроля вызова метода Dispose()
    private bool disposed = false;
    public void Dispose()
    {  // true - очистку запустил пользователь объекта.
       CleanUp(true);
       GC.SuppressFinalize(this); // Подавить финализацию.
    }
    private void CleanUp(bool disposing)
    {   // Удостовериться, не выполнялось ли уже освобождение.
      if (!this.disposed)
       { // disposing равно true, освободить все управляемые ресурсы.
         if (disposing)
            {  // Освободить управляемые ресурсы.    }
         // Очистить неуправляемые ресурсы.
       }
       disposed = true;
}
~MyResource ()
{   // false -  очистку запустил сборщик мусора.
    CleanUp(false)
}}
```