Визуальное программирование

ЛЕКЦИЯ 10

Содержание лекции

01 Жизненный цикл объекта

02 Сборка мусора в .NET

03 Финализация

04 Неуправляемые ресурсы

ЖИЗНЕННЫЙ ЦИКЛ ОБЪЕКТА

Жизненный цикл объекта

Объект проходит несколько различных этапов жизненного цикла, начиная с его создания и заканчивая разрушением.

- Выделяется блок памяти.
- Блок памяти конвертируется в объект. Объект инициализируется.

Можно контролировать только второй из этих двух шагов, превращающий блок памяти в объект. Этот шаг контролируется реализацией конструктора.

CLR выполняет распределение памяти для управляемых объектов, однако, если вызываются неуправляемые библиотеки, возможно, потребуется вручную выделять память для создания объектов.

Жизненный цикл объекта

Когда работа с объектом завершена, он может быть уничтожен.

- Объект очищается, например, путем освобождения любых неуправляемых ресурсов, используемых приложением.
- Память, используемая объектом, возвращается.

Контролировать можно только первый из этих этапов – очистку объекта и освобождения ресурсов.

CLR управляет освобождением памяти, используемой управляемыми объектами, однако, если используются неуправляемые объекты, может потребоваться вручную высвобождать память.

СБОРКА МУСОРА В .NET

Сборка мусора

Сборка мусора представляет собой непрерывный процесс слежения за состоянием памяти. Однако, активную фазу принимает в основном в момент острого недостатка памяти для продолжения работы приложения.

Кроме того, сборщик мусора всегда запускается в отдельном потоке и в момент активной работы его все остальные потоки приложения приостанавливаются.

Преимущества

- Разработчикам не нужно освобождать память вручную.
- Эффективно выделяет память для объектов в управляемой куче.
- Уничтожает объекты, которые больше не используются, очищает их память и сохраняет память доступной для будущих распределений. Управляемые объекты автоматически получают чистое содержимое, поэтому конструкторам не нужно инициализировать каждое поле данных.
- Обеспечивает безопасность памяти, гарантируя, что объект не сможет использовать содержимое другого объекта.

Когда выполняется

- Недостаточно физической памяти в системе. Это можно определить по уведомлению операционной системы о нехватке памяти или по сообщению узла о нехватке памяти.
- Объем памяти, используемой объектами, выделенными в управляемой куче, превышает допустимый порог. Этот порог непрерывно корректируется во время выполнения процесса.
- Вызывается метод GC.Collect . Практически во всех случаях вызов этого метода не потребуется, так как сборщик мусора работает непрерывно. Этот метод в основном используется для уникальных ситуаций и тестирования.

Сборка мусора

Важной функцией сборщика мусора .NET Framework является наблюдение за объектом в куче и определение, когда последняя ссылка на этот объект исчезнет, тогда объект может быть безопасно уничтожен.

Определение момента, когда объект не имеет ссылки, может быть трудоемкой и дорогостоящей операцией, поэтому сборщик мусора выполняет эту задачу только тогда, когда это необходимо, как правило, когда количество доступной памяти в куче падает ниже некоторого порога.

Сборка мусора

Второй функцией сборщика мусора является дефрагментация кучи.

Если приложение пытается создать объект, для которого в настоящее время недостаточно смежного пустого места в куче, сборщик мусора будет пытаться переместить некоторые существующие объекты и сжать результирующее свободное пространство в достаточно большой кусок памяти, чтобы сохранить новый объект.

Этапы сборки мусора

- Сборщик мусора отмечает недостижимые объекты.
- Он начинает с объектов, на которые есть ссылки в стеке, и отмечает используемые объекты как достижимые. Он выполняет это рекурсивно, если объект, который уже отмечен как достижимый ссылается на другой объект, этот объект также отмечен как достижимый.

Этапы сборки мусора

- Сборщик мусора проверяет, имеются ли какие-либо объекты, которые были помечены как недостижимые, деструкторы, которых должны быть выполнены (финализация).

  Объекты, требующие финализацию, перемещаются в структуру данных, поддерживаемую сборщиком мусора - очередь объектов, доступных для финализации (freachable queue).

  Эта очередь хранит указатели на объекты, требующие завершения до восстановления их ресурсов.

Этапы сборки мусора

- Объекты, добавленные в очередь, отмечены как достижимые, потому что существуют действительные ссылки на них; деструктор должен быть запущен перед тем, как их память может быть освобождена.
- Объекты, отмеченные как достижимые, перемещаются вниз кучи для формирования непрерывного блока, дефрагментируя кучу. Ссылки на объекты, перемещенные сборщиком мусора, обновляются.

Этапы сборки мусора

- Другие потоки возобновляются.
- В отдельном потоке объекты, добавленные в freachable очередь, завершаются. После завершения объекта указатель на этот объект удаляется из freachable очереди. Объекты не удаляются из памяти до следующего раза работы сборщика мусора.

Корневые элементы

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

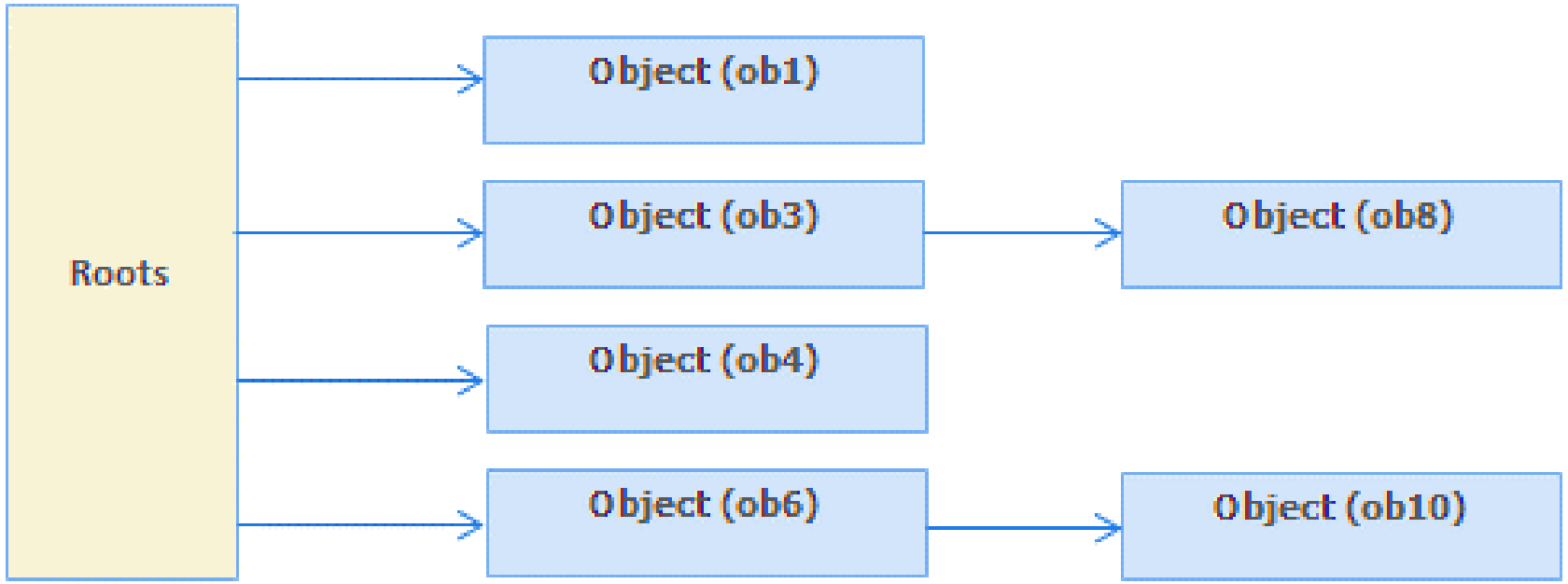
К корневым элементам относятся:

- Ссылки на любые статические объекты или статические поля.
- Ссылки на локальные объекты в пределах кодовой базы приложения.
- Ссылки на передаваемые методу параметры объектов.
- Ссылки на объекты, ожидающие финализации.

Корневые элементы

Во время процесса сборки мусора исполняющая среда будет исследовать объекты в управляемой куче, чтобы определить, являются ли они по-прежнему достижимыми (т.е. корневыми) для приложения.

Для этого среда CLR будет создавать графы объектов, представляющие все достижимые для приложения объекты в куче. При этом сборщик мусора никогда не будет создавать граф для одного и того же объекта дважды, избегая необходимости выполнения подсчета циклических ссылок.

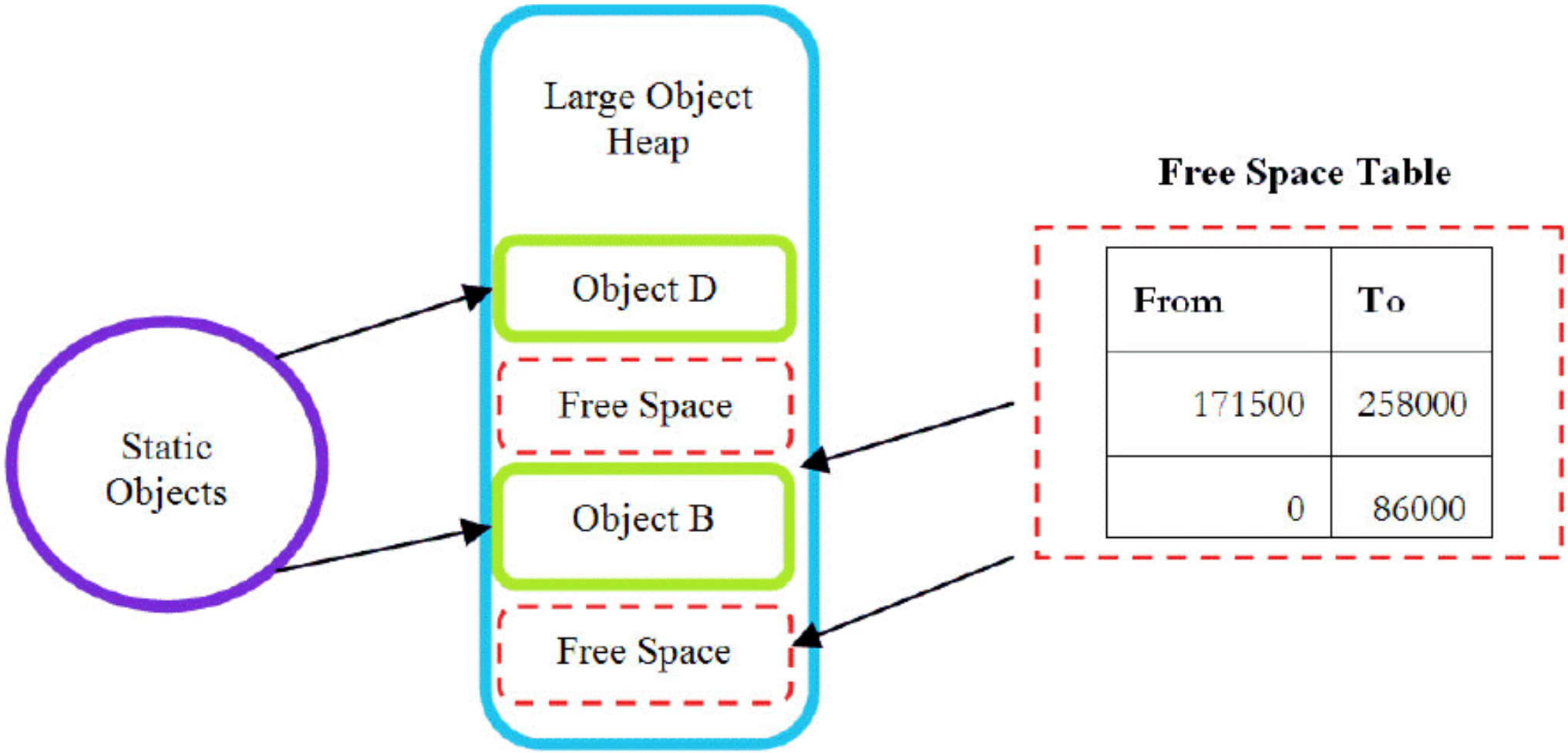


Live objects Graph

Large Object Heap

Сборщик мусора использует две отдельных кучи, одна из которых (LOH – Large Object Heap) предназначена специально для хранения очень больших объектов (размером более 85 Кб).

Доступ к этой куче во время сборки мусора осуществляется реже из-за возможных последствий в плане производительности, в которые может выливаться изменение места размещения больших объектов.


Large Object Heap
Object D
Free Space
Object B
Free Space
Static Objects
Free Space Table
From
To
171500
258000
0
86000

Сборка мусора

Итак, предположим, в данный момент CLR принялась искать неиспользуемые объекты для освобождения необходимого объема памяти. Но CLR не проходит по всем находящимся в памяти объектам и не изучает их. Вместо этого CLR выполняет проход по определенной группе объектов.

Сборка мусора

В .NET эти группы называются поколениями. Всего существует 3 поколения:

Поколение 0. Новый созданный объект, который еще никогда не помечался как кандидат на удаление;

Поколение 1. Объект, который уже однажды пережил сборку мусора. Сюда обычно попадают объекты, которые были помечены для удаления, но все-таки не удалены из-за того, что места в памяти (куче) было предостаточно;

Поколение 2. Объекты, которые пережили более одной очистки памяти сборщиком мусора.

Поколение 0

Это самое молодое поколение содержит короткоживущие объекты. Сборка мусора чаще всего выполняется в этом поколении.

Вновь распределенные объекты образуют новое поколение объектов и неявно являются сборками поколения 0. Большинство объектов уничтожается при сборке мусора для поколения 0 и не доживает до следующего поколения.

Если приложение пытается создать новый объект, когда поколение 0 заполнено, сборщик мусора выполняет сбор, чтобы попытаться освободить адресное пространство для объекта. Сборщик мусора начинает проверять объекты в поколении 0, а не все объекты в управляемой куче. Сборка мусора только в поколении 0 зачастую освобождает достаточно памяти для того, чтобы приложение могло и дальше создавать новые объекты.

Поколение 1

Это поколение содержит коротко живущие объекты и служит буфером между короткоживущими и долгоживущими объектами.

Когда сборщик мусора выполняет сборку для поколения 0, память уплотняется для достижимых объектов и они продвигаются в поколение 1. Так как объекты, оставшиеся после сборки, обычно склонны к долгой жизни, имеет смысл продвинуть их в поколение более высокого уровня. Сборщику мусора необязательно выполнять повторную проверку объектов поколений 1 и 2 при каждой сборке мусора поколения 0.

Если сборка поколения 0 не освобождает достаточно памяти, чтобы приложение могло создать новый объект, сборщик мусора может выполнить сборку мусора поколения 1, а затем поколения 2. Объекты в поколении 1, оставшиеся после сборок, продвигаются в поколение 2.

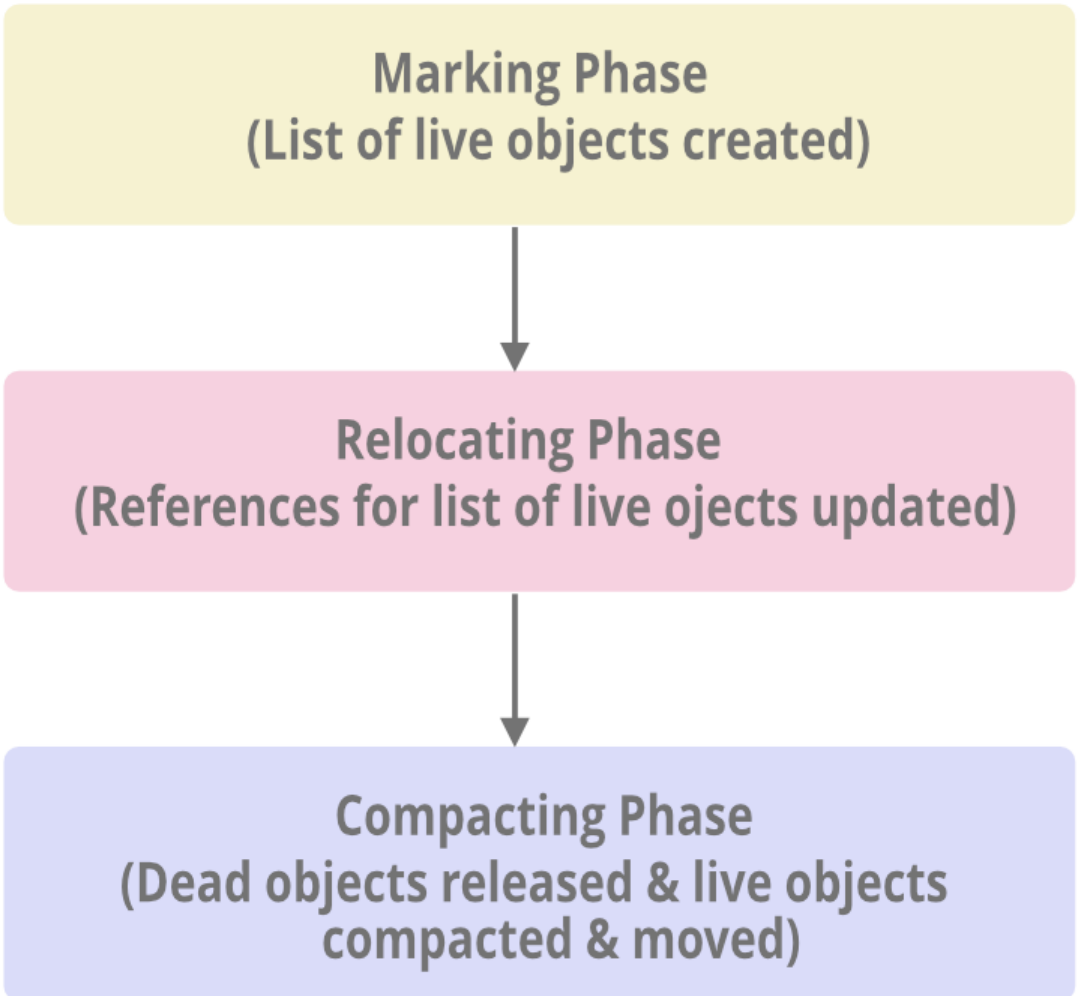
Поколение 2

Это поколение содержит долгоживущие объекты. Примером долгоживущих объектов служит объект в серверном приложении, содержащий статические данные, которые существуют в течение длительности процесса.

Объекты в поколении 2, оставшиеся после сборки, находятся там до тех пор, пока они не станут недостижимыми в следующей сборке.

Объекты в куче больших объектов (иногда называемой поколением 3) также собираются в поколении 2.

Phase in Garbage Collection



ФИНАЛИЗАЦИЯ

Финализатор

В базовом классе .NET System.Object имеется виртуальный метод Finalize(). В предлагаемой по умолчанию реализации он ничего особенного не делает. За счет его переопределения в специальных классах устанавливается специфическое место для выполнения любой необходимой данному типу логики по очистке.

Так как метод Finalize() по определению является защищенным (protected), вызывать его напрямую из класса экземпляра с помощью операции точки не допускается. Вместо этого метод будет автоматически вызываться сборщиком мусора перед удалением соответствующего объекта из памяти.

Финализатор

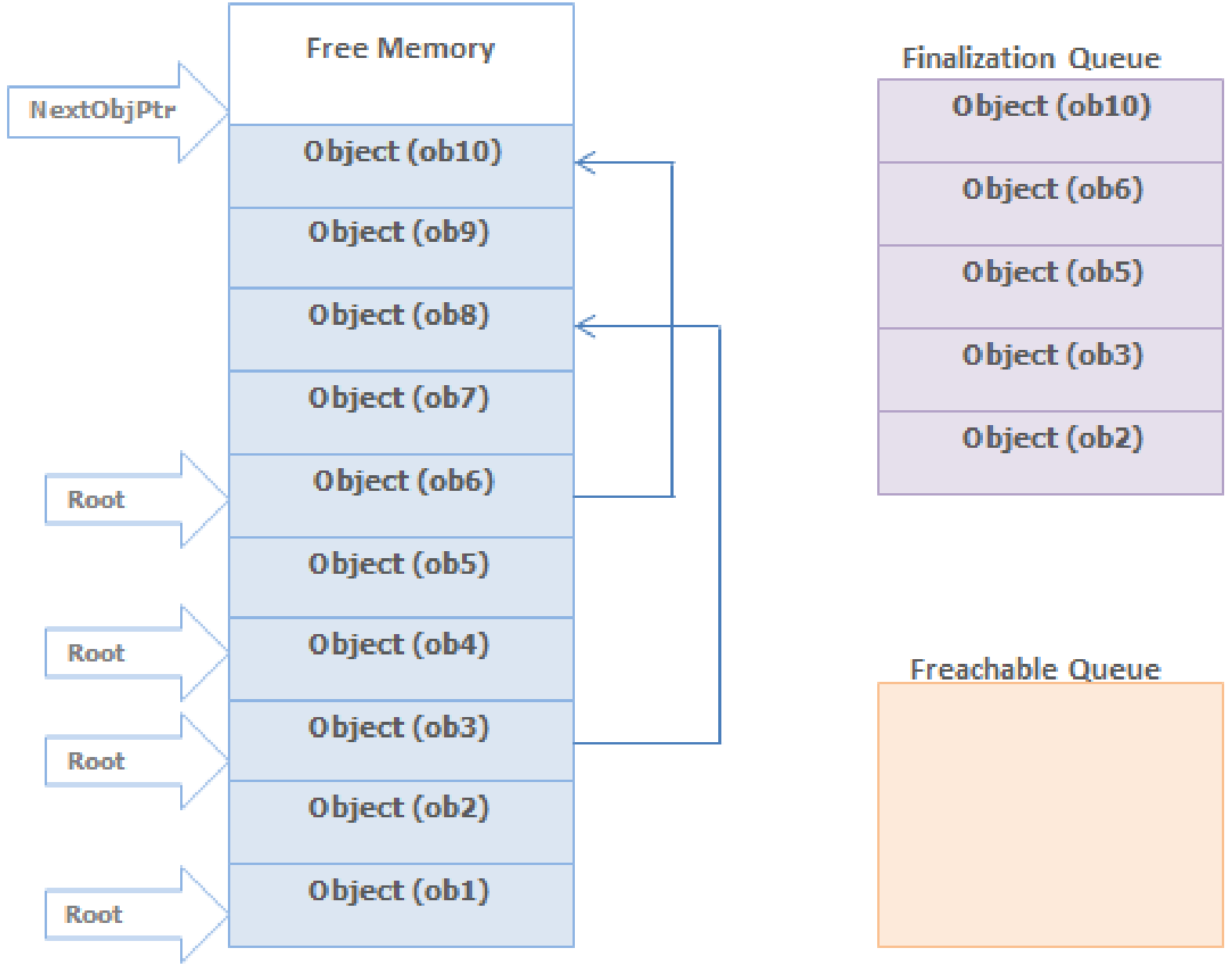
При объявлении деструктора компилятор автоматически преобразует его в переопределение метода Finalize объекта класса, однако нельзя переопределить метод Finalize самостоятельно. Нужно объявить деструктор, и компилятор выполняет преобразование.

```
class Employee
{
    ...
    // Destructor
    ~Employee
    {
        // Destructor logic.
    }
}
```

```
protected override void Finalize()
{
    try
    {
        // Destructor logic.
    }
    finally
    {
        base.Finalize();
    }
}
```

Финализация

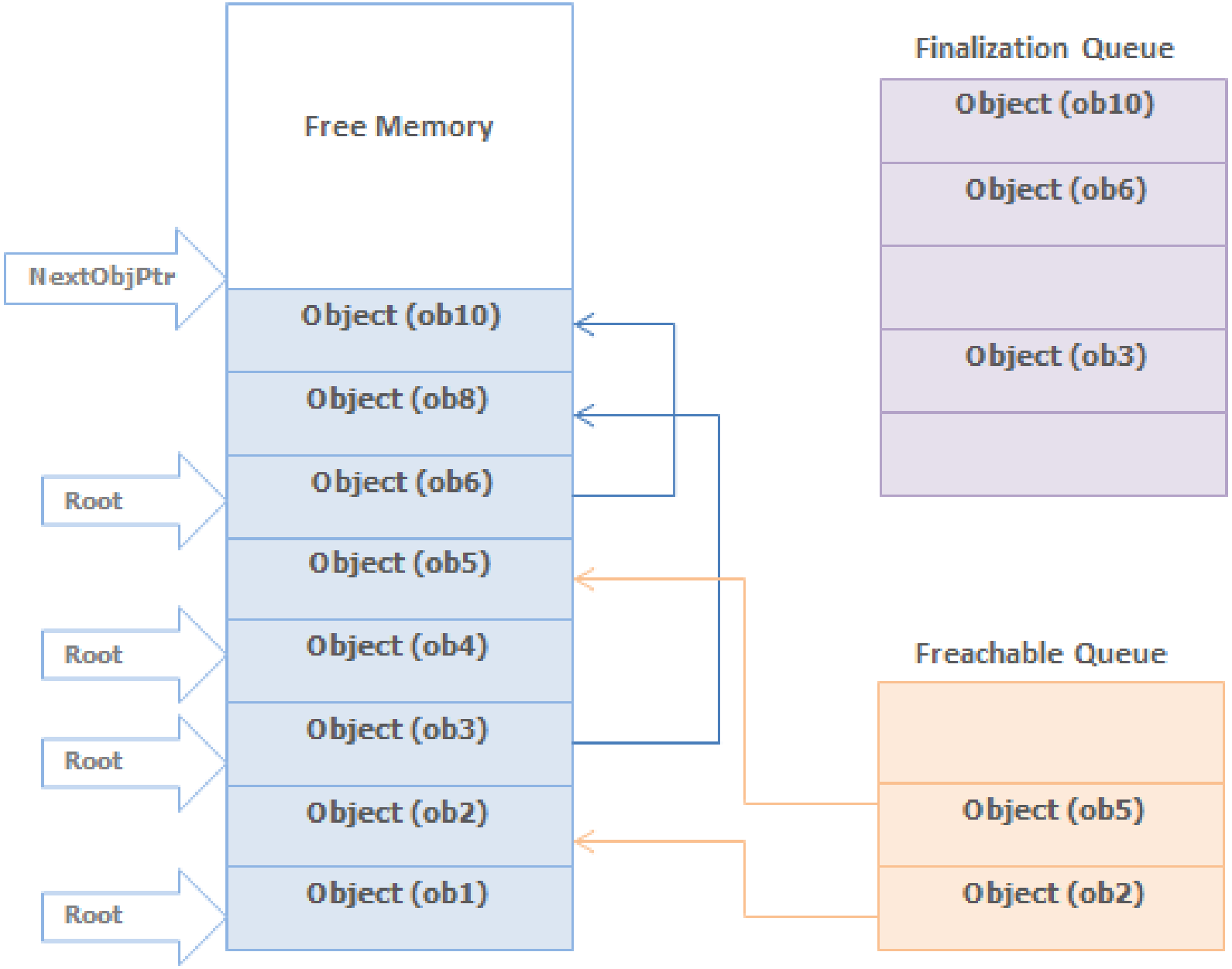
При размещении объекта в управляемой куче исполняющая среда автоматически определяет, поддерживается ли в нем какой-нибудь специальный метод Finalize(). Если да, тогда она помечает его как финализируемый (finalizable) и сохраняет указатель на него во внутренней очереди, называемой очередь финализации (finalizatlon queue).



Managed Heap With Finalize Method

Финализация

Когда сборщик мусора определяет, что наступило время удалить объект из памяти, он проверяет каждую запись в очереди финализации и копирует объект из кучи в еще одну управляемую структуру, называемую таблицей объектов, доступных для финализации (freachable queue).

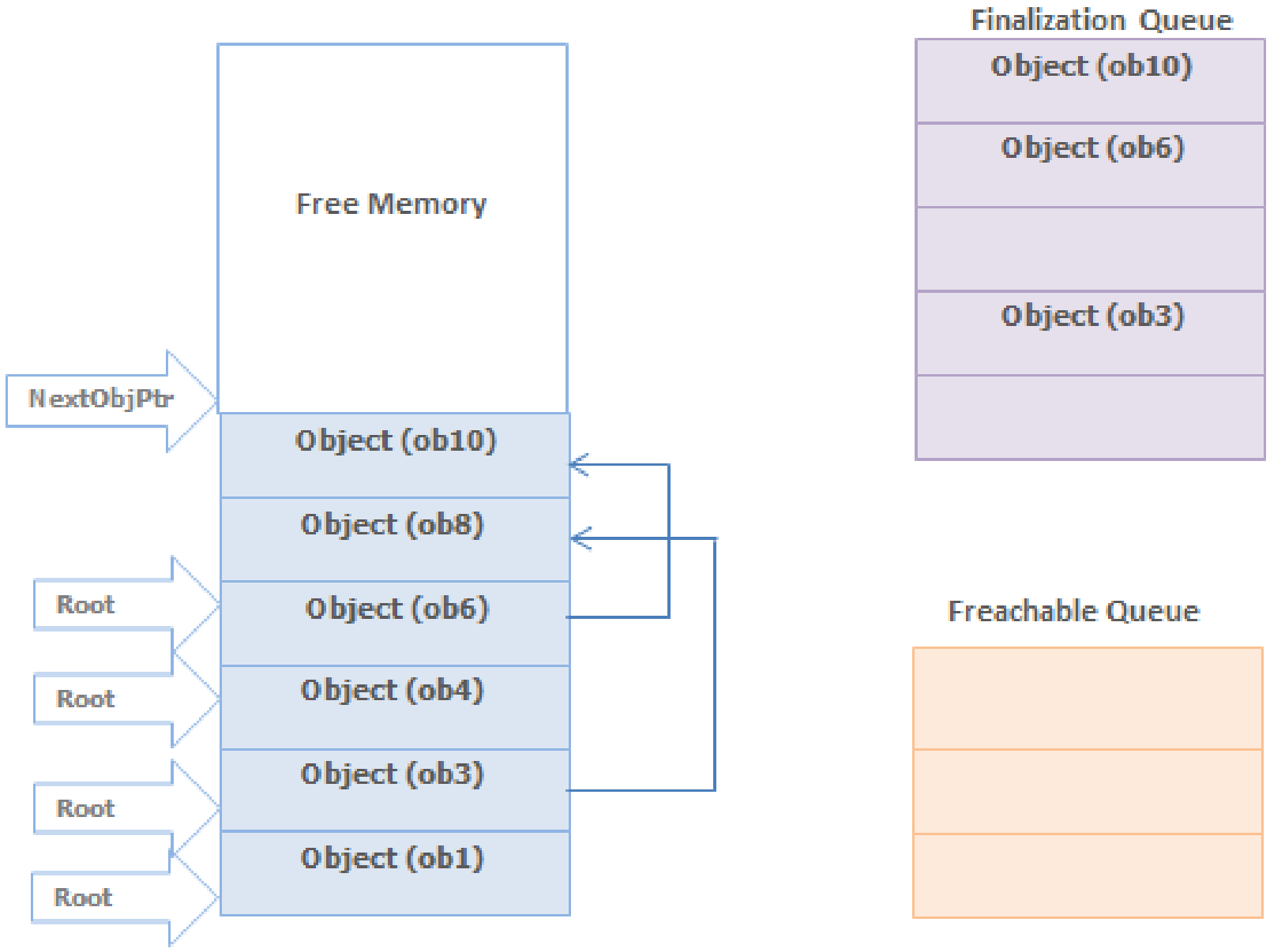


Managed Heap After Garbage Collection

Финализация

После этого он создает отдельный поток для вызова метода Finalize в отношении каждого из упоминаемых в этой таблице объектов при следующей сборке мусора.

В результате получается, что для окончательной финализации объекта требуется как минимум два процесса сборки мусора.



Managed Heap after Second Garbage Collection

НЕУПРАВЛЯЕМЫЕ РЕСУРСЫ

Неуправляемые ресурсы

Сборщик мусора связан с управляемыми объектами. Он не знает, как освободить ресурсы, связанные с неуправляемыми объектами.

Если в классе существует ссылка на неуправляемые ресурсы, при удалении последней ссылки на класс, неуправляемый объект не будет уничтожен.

Операционная система не сможет очистить ресурсы до тех пор, пока приложение не завершится.

Неуправляемые ресурсы

Могут возникнуть следующие проблемы:

- Неуправляемые блокировки (например, блокировка файла)
- Потеря данных из буферов
- Ограниченное количество соединений с БД

Шаблон dispose

Шаблон dispose является шаблоном проектирования, позволяющим высвободить неуправляемые ресурсы, используемые классом, контролируемо и своевременно.

Реализация в типе этого шаблона будет способствовать тому, что приложения будут хорошо работать и не сохранять неуправляемые ресурсы дольше, чем это необходимо.

Интерфейс IDisposable

Интерфейс IDisposable определяет единственный метод Dispose, не принимающий никаких параметров.

Метод Dispose должен освободить все неуправляемые ресурсы, принадлежащие объекту. Он также должен освободить все ресурсы, принадлежащие его базовым типам, вызовом метода Dispose родительского типа.

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

Интерфейс IDisposable

Когда реализуется поддержка интерфейса IDisposable, предполагается, что после завершения работы с объектом метод Dispose должен вручную вызываться пользователем этого объекта, прежде чем объектной ссылке будет позволено покинуть область видимости.

Благодаря этому объект может выполнять любую необходимую очистку неуправляемых ресурсов без попадания в очередь финализации и без ожидания того, когда сборщик мусора запустит содержащуюся в классе логику финализации.

Интерфейс IDisposable

Многие из классов .NET Framework, оборачивающих неуправляемые ресурсы, такие как класс TextWriter, реализуют интерфейс IDisposable.

При создании собственных классов, ссылаемых на неуправляемые типы, необходимо реализовать интерфейс IDisposable.

Для любого создаваемого напрямую объекта, если он поддерживает интерфейс IDisposable, следует всегда вызывать метод Dispose.

Интерфейс IDisposable

Некоторые из типов библиотек базовых классов, реализуя интерфейс IDisposable, предусматривают использование псевдонима для метода Dispose, чтобы заставить отвечающий за очистку метод звучать более естественно для типа, в котором он определяется.

Класс System.IO.FileStream реализует интерфейс IDisposable и поддерживает метод Dispose, но при этом в классе определяется метод Close, применяемый для той же цели.

```
FileStream fs = new FileStream ( "myFile.txt", FileMode .OpenOrCreate);
Fs.Close();
Fs.Dispose();
```

Интерфейс IDisposable

Если нужно гарантировать, чтобы метод Dispose вызывался всегда, можно включить его в качестве части процесса завершения, выполняемого сборщиком мусора. Для этого, можно добавить деструктор для своего класса и вызвать в нем метод Dispose.

Следует помнить, что финализация является потенциально дорогостоящим процессом, поэтому реализовывать эту стратегию следует только тогда, когда это действительно необходимо.

ВОПРОСЫ ПО ЛЕКЦИИ