2 | Управление ресурсами .NET

Тот простой факт, что программы .NET выполняются в управляемой среде, оказывает сильное влияние на виды проектных решений, которые создают эффективный код C#. Использование этой среды в своих интересах требует смены мышления, принятого в других средах, на то, что связано с общеязыковой исполняющей средой (Common Language Runtime — CLR) инфраструктуры .NET. Это означает понимание сборщика мусора .NET (garbage collector — GC). Это означает понимание жизненных циклов объектов. Это означает понимание способа контроля над неуправляемыми ресурсами. В настоящей главе раскрываются практики, которые помогают создавать программное обеспечение, которое наилучшим образом задействует среду и ее средства.

Совет 11. Освойте управление ресурсами .NET

Невозможно стать эффективным разработчиком, не понимая, как среда справляется с памятью и другими важными ресурсами. В .NET это означает понимание управления памятью и сборщика мусора.

Сборщик мусора обеспечивает контроль над управляемой памятью. В отличие от собственных сред вы не несете ответственности за большинство утечек памяти, висячие указатели, неинициализированные указатели и массу остальных проблем, связанных с управлением памятью. Когда вы нуждаетесь в проведении очистки, сборщик мусора работает лучше. Вы отвечаете за неуправляемые ресурсы, такие как подключения к базам данных, объекты GDI+, объекты COM и другие системные объекты. Вдобавок может случиться так, что объекты будут оставаться в памяти дольше, чем хотелось бы, поскольку вы создали связи между ними с применением обработчиков событий или делегатов. Запросы, которые выполняются при просмотре результатов, также могут привести к тому, что ссылки на объекты будут существовать дольше, чем вы ожидали (см. совет 41).

Но есть и хорошие новости: так как GC управляет памятью, определенные проектные идиомы реализовать намного легче, чем в ситуации, когда вы должны управлять всей памятью самостоятельно. Реализовать циклические ссылки, простые отношения и сложные сети объектов корректно здесь намного проще, нежели в средах, где приходится самому управлять памятью. Алгоритм пометки и уплотнения GC эффективно обнаруживает такие отношения и удаляет недостижимые сети объектов целиком. Сборщик мусора определяет достижимость объекта путем прохода по дереву объектов от корневого объекта приложения, не вынуждая каждый объект отслеживать имеющиеся на него ссылки, как происходит в COM.

Класс `EntitySet` является примером того, как упомянутый алгоритм облегчает принятие решений о владении объектами. Сущность представляет собой коллекцию объектов, загруженных из базы данных. Каждая сущность может содержать ссылки на другие сущностные объекты. Любые сущности могут также содержать ссылки на другие сущности. Как и в модели наборов сущностей базы данных, такие связи и ссылки могут быть циклическими.

Ссылки охватывают всю сеть объектов, представленную различными наборами сущностей. За освобождение памяти отвечает GC. Поскольку разработчики приложений .NET Framework не обязаны освобождать имеющиеся объекты, сложная сеть объектных ссылок не порождает проблем. Никаких решений относительно подходящей последовательности освобождения такой сети объектов принимать не придется; это работа GC. Проектное решение, положенное в основу GC, упрощает задачу идентификации такой сети объектов в качестве мусора. Приложение может прекратить ссылаться на любую сущность, когда закончит работу с ней. Сборщику мусора будет известно, по-прежнему достижима ли сущность из активных объектов в приложении. Любые объекты, которые недостижимы из приложения, являются мусором.

Сборщик мусора уплотняет управляемую кучу при каждом своем запуске. Процедура уплотнения перемещает каждый активный объект в управляемой куче так, что свободное пространство занимает один непрерывный блок памяти.

На рис. 2.1 показаны снимки кучи до и после сборки мусора. После каждой операции GC вся свободная память размещается в непрерывном блоке.

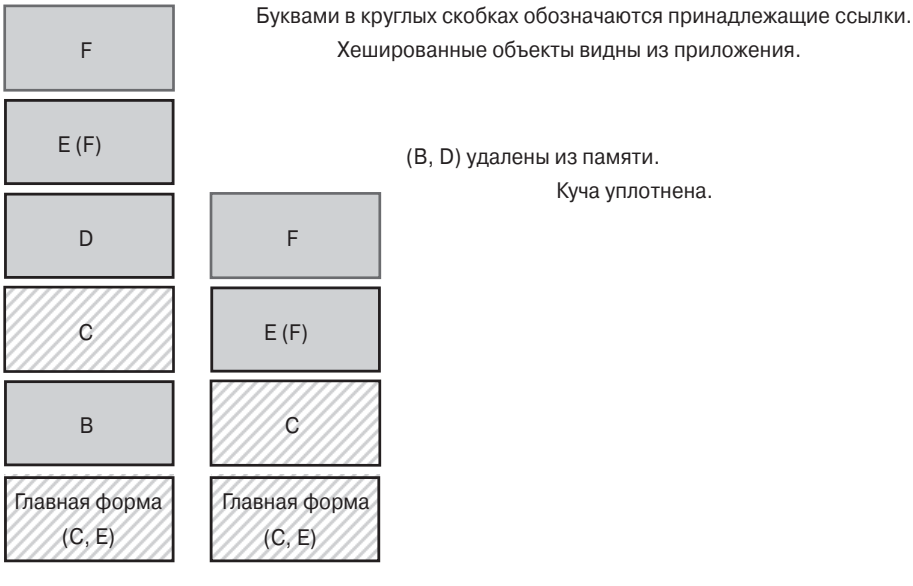

Рис. 2.1. *Сборщик мусора не только освобождает неиспользуемую память, но также перемещает другие объекты в памяти с целью уплотнения используемой памяти и доведения до максимума свободного пространства*

Как вы только что узнали, управление памятью (для управляемой кучи) является обязанностью сборщика мусора. Другие системные ресурсы должны управляться разработчиками: вами и пользователями ваших классов. Два механизма помогают контролировать время жизни неуправляемых ресурсов: финализаторы и интерфейс `IDisposable`. Финализатор представляет собой защитный механизм, который гарантирует, что ваши объекты всегда располагают способом освобождения неуправляемых ресурсов. Финализаторы обладают многими недостатками, поэтому имеется также интерфейс `IDisposable`, предлагающий менее навязчивый метод своевременного возврата ресурсов системе.

Финализаторы вызываются сборщиком мусора в некоторый момент времени после того, как объект стал мусором. Вы не знаете, когда это случится. Вам известно лишь то, что в большинстве сред подобное происходит в какой-то момент после того, как объект перестал быть достижимым. По сравнению с языком C++ это крупное изменение, которое имеет важные последствия для проектных решений. Квалифицированные программисты на C++ пишут классы, которые выделяют критические ресурсы в своих конструкторах и освобождают их в своих деструкторах:

```
// Хороший код C++, но плохой код C#:
class CriticalSection
{
  // Конструктор овладевает системным ресурсом.
  public CriticalSection()
  {
    EnterCriticalSection();
  }

  // Деструктор освобождает системный ресурс.
  ~CriticalSection()
  {
    ExitCriticalSection();
  }

  private void ExitCriticalSection()
  {
  }
  private void EnterCriticalSection()
  {
  }
}

// Использование:
void Func()
{
  // Жизненный цикл s управляет доступом к системному ресурсу.
  CriticalSection s = new CriticalSection();
  // Выполнить работу.

  // ...

  // Компилятор генерирует вызов деструктора.
  // Код выходит из критического раздела.
}
```

Показанная общая идиома C++ гарантирует освобождение ресурса безо всяких исключений. Однако в C# такой подход не работает — во всяком случае, не в той же самой манере. Детерминированная финализация не является частью среды .NET или языка C#. Попытка навязывания идиомы детерминированной финализации C++ в языке C# ни к чему хорошему не приводит. В большинстве сред финализатор C# в конце концов выполняется, но не своевременно. В предыдущем примере код, в конечном счете, покинет критический раздел, но в случае C# он не выходит из критического раздела, когда функция завершается. Это произойдет в неизвестный момент времени позже. Вы не знаете когда. И вы не можете узнать когда.

Финализаторы — единственный способ гарантировать, что неуправляемые ресурсы, выделенные объектом заданного типа, в итоге освободятся. Но финализаторы выполняются в недетерминированные моменты времени, поэтому ваши практики проектирования и кодирования должны сводить к минимуму потребность в создании финализаторов, а также минимизировать необходимость в выполнении финализаторов, которые уже существуют. В настоящей главе вы изучите приемы, позволяющие избегать создания собственных финализаторов, и узнаете, как сводить к минимуму негативное влияние финализаторов, когда они должны присутствовать.

Зависимость от финализаторов также привносит накладные расходы, связанные с производительностью. В смысле производительности объекты, требующие финализации, становятся обузой для сборщика мусора. Когда GC обнаруживает, что объект подвергается сборке мусора, но также запрашивает финализацию, он не может сразу же удалить данный элемент из памяти. Сначала он вызывает финализатор. Финализаторы не выполняются в том же потоке, в котором производится сборка мусора. Взамен GC помещает каждый объект, готовый к финализации, в очередь и выполняет все финализаторы для таких объектов. Он продолжает свою работу, удаляя другой мусор из памяти. На следующем цикле GC объекты, которые были финализированы, удаляются из памяти. На рис. 2.2 показаны три операции GC и демонстрируется разница в использовании памяти. Обратите внимание, что объекты, требующие запуска финализаторов, остаются в памяти для дополнительных циклов.

Это может навести на мысль, что объект, который требует финализации, находится в памяти на один цикл GC больше, чем необходимо. Но я несколько упростил положение дел. Из-за еще одного проектного решения GC ситуация сложнее. Для оптимизации работы сборщик мусора .NET определяет поколения, которые помогают ему быстрее идентифицировать самых вероятных кандидатов на сборку. Любой объект, созданный с момента последней операции сборки мусора, является объектом поколения 0. Любой объект, который пережил одну операцию GC, представляет собой объект поколения 1. Любой объект, переживший две или больше операций GC, становится объектом поколения 2. Цель поколений заключается в отделении недолговечных объектов от объектов, которые существуют на протяжении времени жизни приложения. Объекты поколения 0 — это главным образом такие недолговечные объектные переменные. Переменные-члены и глобальные переменные быстро переходят в поколение 1 и со временем в поколение 2.

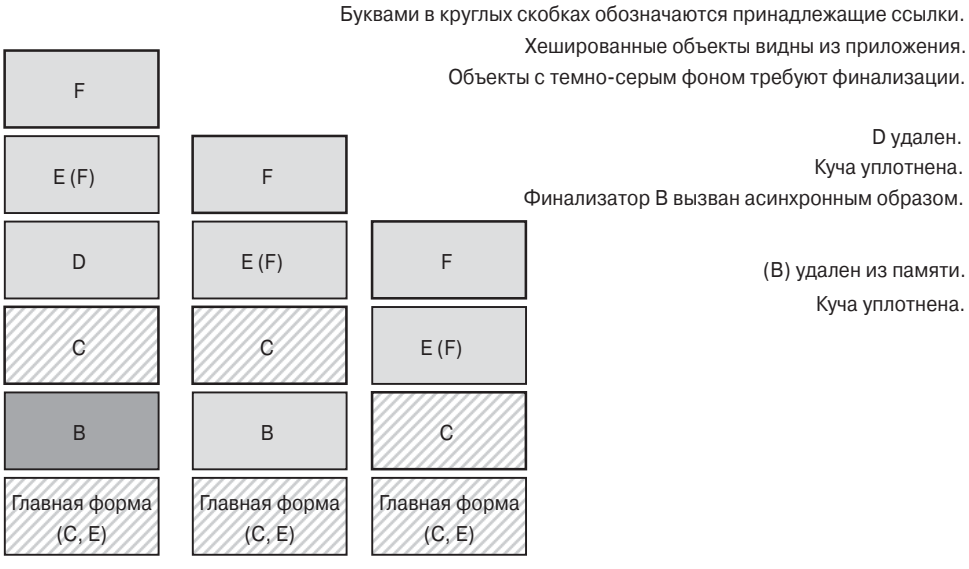

Рис. 2.2. *Данная последовательность иллюстрирует влияние финализаторов на сборщик мусора. Объекты остаются в памяти дольше и необходимо породить дополнительный процесс для выполнения финализаторов*

Сборщик мусора оптимизирует свою работу, ограничивая частоту исследования объектов поколений 1 и 2. Каждый цикл GC проверяет объекты поколения 0. Приблизительно один цикл GC из десяти исследует объекты поколений 0 и 1. Примерно один цикл GC из ста поверяет все объекты. Снова подумайте о финализации и связанными с ней затратами: объект, который требует финализации, может оставаться в памяти на девять циклов GC дольше, чем в случае, если бы он не нуждался в финализации. Если объект все еще не был финализирован, тогда он переходит в поколение 2, в котором он будет находиться на протяжении дополнительных ста циклов GC до тех пор, пока не произойдет следующая сборка для поколения 2.

Я потратил некоторое время на объяснения, почему финализаторы не являются удачным решением. Тем не менее, освобождать ресурсы по-прежнему необходимо. Вы решаете эти проблемы с применением интерфейса `IDisposable` и стандартного паттерна “Освобождение” (Dispose), как рассматривается в совете 17 далее в главе.

Итак, запомните, что управляемая среда, где сборщик мусора возлагает на себя обязанности по управлению памятью, обеспечивает крупное преимущество: утечки памяти и многие другие проблемы, связанные с указателями, больше вас не касаются. Ресурсы, не имеющие отношения к памяти, вынуждают вас создавать финализаторы для обеспечения их надлежащей очистки. Финализаторы могут оказывать серьезное влияние на производительность программы, но вы должны предусматривать их во избежание утечек ресурсов. Реализация и использование интерфейса `IDisposable` устраняет непроизводительные расходы сборщика мусора, обусловленные финализаторами. В следующем совете описаны специфические приемы, которые помогут создавать программы, более эффективно эксплуатирующие эту среду.

Совет 12. Отдавайте предпочтение инициализаторам членов перед операторами присваивания

Часто классы имеют более одного конструктора. С течением времени переменные-члены и конструкторы могут легко потерять синхронизацию. Чтобы гарантировать невозможность возникновения такой ситуации, лучше всего инициализировать переменные там, где они объявляются, а не в теле каждого конструктора. Синтаксис инициализаторов должен применяться как для статических переменных, так и для переменных экземпляра.

Конструирование переменной-члена при ее объявлении в C# вполне естественно. Нужно всего лишь инициализировать переменную, когда она объявляется:

```
public class MyClass
{
  // Объявить и инициализировать коллекцию.
  private List<string> labels = new List<string>();
}
```

Безотносительно к количеству конструкторов, которые в итоге будут добавлены к типу `MyClass`, переменная-член `labels` инициализируется должным образом. Компилятор генерирует в начале каждого конструктора код для выполнения всех инициализаторов, которые были определены вместе с переменными-членами экземпляра. При добавлении нового конструктора `labels` инициализируется. Кроме того, в случае добавления новой переменной-члена не придется добавлять код инициализации в каждый конструктор; вполне достаточно инициализировать переменную в месте ее определения. В равной степени важно и то, что инициализаторы добавляются к стандартному конструктору, генерируемому компилятором. Компилятор C# создает стандартный конструктор для типов, в которых не определяются явно какие-нибудь конструкторы.

Инициализаторы — это нечто большее, чем просто удобное сокращение для операторов в теле конструктора. Операторы, генерируемые инициализаторами, помещаются в объектный код перед телом каждого конструктора. Инициализаторы выполняются до выполнения конструктора базового класса для типа, причем в порядке, в котором определялись связанные с ними переменные.

Использование инициализаторов является простейшим способом избежать наличия неинициализированных переменных в типах, но он не безупречен. В трех случаях синтаксис инициализаторов применяться не должен. Первая ситуация касается инициализации объекта значением 0 или `null`. Стандартная системная инициализация устанавливает все в 0 перед выполнением любого кода. Генерируемая системой инициализация значением 0 делается на очень низком уровне с использованием инструкций центрального процессора для установки в 0 сразу всего блока памяти. Любая дополнительная инициализация значением 0 с вашей стороны будет излишней. Компилятор C# послушно добавит дополнительные инструкции для установки памяти в 0 еще раз. Это не ошибка, но может привести к созданию хрупкого кода.

```
public struct MyValType
{
  // Детали не показаны.
}
MyValType myVal1;                       // инициализируется в 0
MyValType myVal2 = new MyValType();     // также 0
```

Оба оператора инициализируют переменную всеми нулями. Первый делает это путем установки в 0 памяти, содержащей `myVal1`. Второй применяет инструкцию `initobj` языка IL, которая служит причиной выполнения операций упаковки и распаковки над переменной `myVal2`, что займет немало добавочного времени (см. совет 9).

Вторая неэффективность возникает, когда для одного объекта создается несколько инициализаций. Синтаксис инициализаторов должен использоваться только в отношении переменных, которые одинаково инициализируются во всех конструкторах. В следующей версии класса `MyClass` имеется путь, по которому в качестве части конструирования его экземпляра создаются два разных объекта `List`:

```
public class MyClass2
{
  // Объявить и инициализировать коллекцию.
  private List<string> labels = new List<string>();

  MyClass2()
  {
  }
  MyClass2(int size)
  {
    labels = new List<string>(size);
  }
}
```

В случае создания нового экземпляра `MyClass2` с указанием размера коллекции создаются два объекта `List`. Один из них немедленно подвергается сборке мусора. Инициализатор переменой выполняется перед любым конструктором. В теле конструктора создается второй объект `List`. Компилятор создает такую версию `MyClass2`, которую вы бы никогда не написали вручную. (В совете 14 рассматривается подходящий способ выхода из такой ситуации.)

```
public class MyClass2
{
  // Объявить и инициализировать коллекцию.
  private List<string> labels;

  MyClass2()
  {
    labels = new List<string>();
  }
  MyClass2(int size)
  {
    labels = new List<string>();
    labels = new List<string>(size);
  }
}
```

Попасть в ту же самую ситуацию можно во время применения неявных свойств. Для элементов данных, которым подходят неявные свойства, в совете 14 демонстрируется способ сведения к минимуму дублированного кода, когда инициализируются данные, хранящиеся в неявных свойствах.

Последняя причина переноса инициализации в тело конструктора связана с упрощением обработки исключений. Инициализаторы нельзя помещать в блок `try`. Любые исключения, которые могут возникнуть при конструировании переменных-членов, распространятся за пределы объекта. Предпринять какие-то восстановительные действия внутри класса не удастся. Код инициализации потребуется переместить в тела конструкторов, чтобы можно было предусмотреть подходящий код восстановления и элегантной обработки исключений (см. совет 47).

Инициализаторы членов являются простейшим способом обеспечить инициализацию переменных-членов в типе независимо от того, какой конструктор вызывается. Инициализаторы выполняются перед любым конструктором типа. Использование такого синтаксиса означает невозможность забыть о добавлении надлежащей инициализации при определении новых конструкторов для будущего выпуска. Применяйте инициализаторы, когда все конструкторы создают переменную-член аналогичным образом; код будет легче воспринимать и сопровождать.

Совет 13. Используйте подходящую инициализацию для статических членов класса

Вам известно, что статические переменные-члены в типе должны инициализироваться до создания любых экземпляров этого типа. Язык C# позволяет применять для такой цели статические инициализаторы и статический конструктор. *Статический конструктор* представляет собой специальную функцию, которая выполняется перед первым обращением к любым другим методам, переменным или свойствам, определенным в данном классе. Эта функция используется для инициализации статических переменных, внедрения паттерна “Одиночка” или выполнения другой необходимой работы, прежде чем класс станет пригодным к употреблению. При инициализации статических переменных не должны применяться конструкторы экземпляра, некоторые специальные закрытые функции или любая другая идиома. Для статических полей, которые требуют сложной или затратной инициализации, подумайте об использовании `Lazy<T>`, чтобы инициализация выполнялась при первом обращении к таким полям.

Как и с инициализацией экземпляра, синтаксис инициализаторов можно применять в качестве альтернативы статическому конструктору. Если вы просто нуждаетесь в выделении статического члена, тогда используйте синтаксис инициализаторов. Когда инициализировать статические переменные-члены необходимо посредством более сложной логики, то создавайте статический конструктор.

Чаще всего статический конструктор применяется при реализации паттерна “Одиночка” в C#. Сделайте конструктор экземпляра закрытым и добавьте инициализатор:

```
public class MySingleton
{
  private static readonly MySingleton theOneAndOnly = new MySingleton();
  public static MySingleton TheOnly
  {
    get { return theOneAndOnly; }
  }
  private MySingleton()
  {
  }
  // Остальной код не показан.
}
```

При наличии более сложной логики для инициализации одиночки паттерн "Одиночка" может быть легко реализован следующим образом:

```
public class MySingleton2
{
  private static readonly MySingleton2 theOneAndOnly;
  static MySingleton2()
  {
    theOneAndOnly = new MySingleton2();
  }
  public static MySingleton2 TheOnly
  {
    get { return theOneAndOnly; }
  }
  private MySingleton2()
  {
  }
  // Остальной код не показан.
}
```

Подобно инициализаторам экземпляра статические инициализаторы выполняются перед вызовом любых статических конструкторов. И да, ваши статические инициализаторы могут выполняться до статического конструктора базового класса.

Среда CLR автоматически вызывает статический конструктор перед первым обращением к типу в пространстве приложения (`AppDomain`). Допускается определять только один статический конструктор, и он не должен принимать аргументы. Поскольку статические конструкторы вызываются CLR, вы должны заботиться об исключениях, генерируемых в нем. Если вы позволите исключению выйти из статического конструктора, тогда среда CLR прекратит выполнение программы, сгенерировав исключение `TypeInitializationException`. Ситуация, когда вызывающий код перехватывает исключение, еще более коварна. Код, который пытается создать тип, будет отказывать до тех пор, пока не выгрузится `AppDomain`. Среда CLR могла не инициализировать тип за счет выполнения статического конструктора. Она не будет пытаться делать это снова, и тип окажется некорректно инициализированным. Объект такого типа (или любого производного от него типа) не будет полностью определенным. Следовательно, он недопустим.

Исключения являются наиболее распространенной причиной использования статического конструктора вместо статических инициализаторов. Если вы применяете статические инициализаторы, то не сможете самостоятельно перехватывать исключения. В случае статического конструктора это делать можно (см. совет 47):

```
static MySingleton2()
{
  try
  {
    theOneAndOnly = new MySingleton2();
  }
  catch
  {
    // Попытка восстановления.
  }
}
```

Статические инициализаторы и статические конструкторы предоставляют более ясный и чистый способ для инициализации статических членов класса. Их легче понимать и проще обеспечивать правильность. Они были добавлены в язык специально для устранения трудностей, связанных с инициализацией статических членов в других языках.

Совет 14. Сводите к минимуму дублирование логики инициализации

Написание кода конструкторов часто является повторяющейся задачей. Многие разработчики пишут первый конструктор и затем копируют код в другие конструкторы, чтобы создать многочисленные перегруженные версии, определенные в интерфейсе класса. В идеале вы не один из них. Но если вы поступаете так, тогда прекратите это. Опытные программисты на C++ выносили бы общие алгоритмы в закрытые вспомогательные методы. И это также прекратите. Когда вы обнаруживаете, что многие конструкторы содержат одну и ту же логику, выносите ее в общий конструктор. Вы получите преимущества устранения дублированного кода, а инициализаторы конструкторов генерируют гораздо более эффективный объектный код. Компилятор C# распознает инициализатор конструктора как специальный синтаксис и удаляет дублирующие инициализаторы переменных, а также дублирующие вызовы конструкторов базового класса. В результате для надлежащей инициализации финального объекта выполняется минимальный объем кода. Также понадобится написать наименьший объем кода за счет делегирования обязанностей общему конструктору.

Инициализаторы конструкторов позволяют одному конструктору вызывать другой конструктор. Ниже приведен простой пример:

```
public class MyClass
{
  // Коллекция данных.
  private List<ImportantData> coll;

  // Имя экземпляра:
  private string name;
```

```
  public MyClass() :
    this(0, "")
  {
  }

  public MyClass(int initialCount) :
    this(initialCount, string.Empty)
  {
  }

  public MyClass(int initialCount, string name)
  {
    coll = (initialCount > 0) ?
      new List<ImportantData>(initialCount) :
      new List<ImportantData>();
    this.name = name;
  }
}
```

В версии C# 4.0 появились стандартные параметры, которые можно использовать для минимизации дублирования кода в конструкторах. Все конструкторы класса `MyClass` можно было бы заменить одним конструктором, в котором указываются стандартные значения для всех или многих параметров:

```
public class MyClass
{
  // Коллекция данных.
  private List<ImportantData> coll;

  // Имя экземпляра:
  private string name;

  // Необходимо для удовлетворения ограничения new().
  public MyClass() :
    this(0, string.Empty)
  {
  }

  public MyClass(int initialCount = 0, string name = "")
  {
    coll = (initialCount > 0) ?
      new List<ImportantData>(initialCount) :
      new List<ImportantData>();
    this.name = name;
  }
}
```

Выбор применения стандартных параметров вместо многочисленных перегруженных версий сопряжен с компромиссами. Стандартные параметры открывают больше вариантов вашим пользователям. В приведенной выше версии `MyClass` стандартные значения предусмотрены для обоих параметров. Пользователи могут указывать другое значение для любого или для обоих параметров. Воспроизведение всех перестановок с использованием перегрузки потребовало бы четырех перегруженных конструкторов: конструктор без параметров, конструктор, запрашивающий `initialCount`, конструктор, запрашивающий `name`, и конструктор, запрашиваю-

щий оба параметра. С добавлением в класс дополнительных членов растет количество потенциальных перегруженных версий, т.к. увеличивается число перестановок всех параметров. Подобная сложность превращает стандартные параметры в очень мощный механизм минимизации количества потенциальных перегруженных версий, которые придется создавать.

Определение стандартных значений для всех параметров конструктора типа означает, что пользовательский код будет допустимым при вызове `new MyClass()`. Если вы намерены поддерживать такую концепцию, то должны создать в типе явный конструктор без параметров, как было показано в примере кода выше. Несмотря на то что в большинстве случаев приемлемо определение стандартных значений для всех параметров, обобщенные классы, применяющие ограничение `new()`, не будут принимать конструктор с параметрами, которые имеют стандартные значения. Чтобы удовлетворять ограничению `new()`, класс обязан иметь явный конструктор без параметров. Следовательно, вы должны создать такой конструктор, чтобы клиенты могли использовать ваш тип в обобщенных классах или методах, которые обеспечивают соблюдение ограничения `new()`. Это не говорит о том, что каждый тип нуждается в конструкторе без параметров. Однако если вы его поддерживаете, тогда обеспечьте добавление кода, чтобы конструктор без параметров работал во всех случаях, даже когда он вызывается из обобщенного класса с ограничением `new()`.

Вы наверняка заметили, что во втором конструкторе указано стандартное значение `""` для параметра `name`, а не более привычное `string.Empty`. Причина в том, что `string.Empty` не является константой этапа компиляции, а представляет собой статическое свойство, определенное в классе `string`. Поскольку это не константа этапа компиляции, `string.Empty` нельзя применять в качестве стандартного значения для параметра.

Тем не менее, использование стандартных параметров вместо перегруженных версий порождает более сильную связь между вашим классом и клиентами, которые с ним работают. В частности, имя формального параметра становится частью открытого интерфейса, как и текущее стандартное значение. Изменение значений параметров требует перекомпиляции всего клиентского кода, чтобы внесенные изменения были учтены. Это делает перегруженные конструкторы более устойчивыми перед лицом возможных изменений в будущем. Вы можете добавлять новые конструкторы либо изменять стандартное поведение конструкторов, в которых не указаны значения, без нарушения работы клиентского кода.

Предпочтительное решение данной проблемы обеспечивают стандартные параметры. Однако некоторые API-интерфейсы для создания объектов применяют рефлексию и полагаются на конструктор без параметров. Конструктор со стандартными значениями для всех аргументов — не то же самое, что конструктор без параметров. Может понадобиться написать отдельные конструкторы и поддерживать их как независимые функции. В случае конструкторов это может означать большой объем дублированного кода. Используйте соединение конструкторов в цепочку, заставляя один конструктор вызывать другой конструктор, определенный в том же классе, вместо создания общей вспомогательной процедуры. Такому альтернативному способу вынесения общей логики конструкторов присущи некоторые недостатки:

```
public class MyClass
{
  private List<ImportantData> coll;
  private string name;
  public MyClass()
  {
    commonConstructor(0, "");
  }
  public MyClass(int initialCount)
  {
    commonConstructor(initialCount, "");
  }
  public MyClass(int initialCount, string Name)
  {
    commonConstructor(initialCount, Name);
  }
  private void commonConstructor(int count,
      string name)
  {
    coll = (count > 0) ?
      new List<ImportantData>(count) :
      new List<ImportantData>();
    this.name = name;
  }
}
```

Версия класса выглядит такой же, но генерирует гораздо менее эффективный объектный код. Компилятор добавляет в конструкторы код для выполнения нескольких функций. Он добавляет операторы для всех инициализаторов переменных (см. совет 12 ранее в главе). Он вызывает конструктор базового класса. Когда вы предусматриваете собственную общую вспомогательную функцию, компилятор не в состоянии вынести этот дублированный код. Код IL для второй версии будет таким же, как если бы вы написали следующий код:

```
public class MyClass
{
  private List<ImportantData> coll;
  private string name;
  public MyClass()
  {
    // Здесь должны находиться инициализаторы экземпляра.
    object();  // Не допускается, только в целях иллюстрации.
    commonConstructor(0, "");
  }
  public MyClass(int initialCount)
  {
    // Здесь должны находиться инициализаторы экземпляра.
    object();  // Не допускается, только в целях иллюстрации.
    commonConstructor(initialCount, "");
  }
```

```
  public MyClass(int initialCount, string Name)
  {
    // Здесь должны находиться инициализаторы экземпляра.
    object();  // Не допускается, только в целях иллюстрации.
    commonConstructor(initialCount, Name);
  }
  private void commonConstructor(int count,
      string name)
  {
    coll = (count > 0) ?
      new List<ImportantData>(count) :
      new List<ImportantData>();
    this.name = name;
  }
}
```

Если бы вы могли написать код конструирования для первой версии, как его видит компилятор, то поступили бы так:

```
// Не допускается, иллюстрирует генерируемый код IL:
public class MyClass
{
  private List<ImportantData> coll;
  private string name;
  public MyClass()
  {
    // Инициализаторы переменных здесь отсутствуют.
    // Вызвать третий конструктор, показанный ниже.
    this(0, "");  // Не допускается, только в целях иллюстрации.
  }
  public MyClass(int initialCount)
  {
    // Инициализаторы переменных здесь отсутствуют.
    // Вызвать третий конструктор, показанный ниже.
    this(initialCount, "");
  }
  public MyClass(int initialCount, string Name)
  {
    // Здесь должны находиться инициализаторы экземпляра.
    // object();  // Не допускается, только в целях иллюстрации.
    coll = (initialCount > 0) ?
      new List<ImportantData>(initialCount) :
      new List<ImportantData>();
    name = Name;
  }
}
```

Отличие в том, что компилятор не генерирует множество вызовов конструктора базового класса, равно как не копирует инициализаторы переменных экземпляра в тело каждого конструктора. Тот факт, что конструктор базового класса вызывается только в последнем конструкторе, также важен: помещать в определение конструк-

тора более одного инициализатора конструктора нельзя. Можно делегировать работу другому конструктору в классе с применением `this()` или вызывать конструктор базового класса, используя `base()`. Делать и то, и другое невозможно.

Все еще не нашли место, занимаемое инициализаторами конструктора? Тогда подумайте о константах только для чтения. В следующем примере имя объекта не должно изменяться на протяжении его времени жизни. Это значит, что вы должны сделать его допускающим только чтение. В таком случае общая вспомогательная функция будет генерировать ошибку на этапе компиляции:

```
public class MyClass
{
  // Коллекция данных.
  private List<ImportantData> coll;
  // Количество для этого экземпляра.
  private int counter;
  // Имя этого экземпляра:
  private readonly string name;

  public MyClass()
  {
    commonConstructor(0, string.Empty);
  }

  public MyClass(int initialCount)
  {
    commonConstructor(initialCount, string.Empty);
  }

  public MyClass(int initialCount, string Name)
  {
    commonConstructor(initialCount, Name);
  }

  private void commonConstructor(int count,
      string name)
  {
    coll = (count > 0) ?
      new List<ImportantData>(count) :
      new List<ImportantData>();
    // ОШИБКА, связанная с изменением имени за пределами конструктора.
    // this.name = name;
  }
}
```

Компилятор обеспечит для свойства `this.name` природу, допускающую только чтение, и не разрешит модифицировать его в любом коде, находящемся не в конструкторе. Инициализаторы конструктора C# предлагают альтернативу. Все классы кроме самых тривиальных содержат более одного конструктора. Их работа заключается в инициализации всех членов объекта. По самой своей природе они имеют похожую или в идеальном случае разделяемую логику. Применяйте инициализатор конструктора C# для вынесения общих алгоритмов, чтобы их можно было написать один раз и выполнять один раз.

Стандартные параметры и перегруженные версии занимают свою нишу. Обычно вы должны отдавать предпочтение стандартным значениям перед перегруженными конструкторами. В конце концов, если вы вообще позволяете клиентскому коду указывать значения параметров, тогда ваш конструктор должен быть в состоянии обрабатывать любые значения, которые были указаны. Ваши исходные стандартные значения всегда должны быть приемлемыми и не приводить к генерации исключений. Следовательно, хотя изменение стандартных значений параметров формально является критическим изменением, оно не должно быть видимым для ваших клиентов. Их код по-прежнему будет использовать первоначальные значения, которые должны обеспечивать разумное поведение. Это сводит к минимуму потенциальные риски, связанные с применением стандартных значений.

Это последний совет относительно инициализации объектов в C#. Наступило подходящее время просмотреть полную последовательность событий для конструирования экземпляра типа. Вы должны понимать последовательность операций и стандартную инициализацию объекта. Во время конструирования вы должны стремиться инициализировать каждую переменную-член в точности один раз. Лучший способ достичь указанного — инициализировать значения как можно раньше. Ниже описан порядок выполнения операций для конструирования экземпляра типа.

1. Хранилище статических переменных устанавливается в 0.
2. Выполняются инициализаторы статических переменных.
3. Выполняются статические конструкторы базового класса.
4. Выполняются статические конструкторы.
5. Хранилище переменных экземпляра устанавливается в 0.
6. Выполняются инициализаторы переменных экземпляра.
7. Выполняется подходящий конструктор экземпляра базового класса.
8. Выполняется конструктор экземпляра.

Создание последующих экземпляров того же типа начинается с шага 5, потому что инициализаторы класса выполняются только раз. Кроме того, шаги 6 и 7 оптимизируются, так что инициализаторы конструктора приводят к удалению компилятором дублированных инструкций.

Компилятор языка C# гарантирует, что во время создания объекта все каким-нибудь образом будет инициализировано. По меньшей мере, вы получаете гарантию того, что вся память, используемая объектом, была установлена в 0, когда экземпляр создан. Это касается и статических членов, и членов экземпляра. Ваша цель — обеспечить инициализацию всех значений желаемым способом и выполнение инициализирующего кода только один раз. Применяйте инициализаторы для инициализации простых ресурсов. Используйте конструкторы для инициализации членов, которые требуют более сложной логики. Также учитывайте вызовы других конструкторов, чтобы свести к минимуму дублирование кода.