|  |
| --- |
| , RD Dep. |
| Конспект и раздаточный материал  NET.C#.03 Объявление и вызов методов в C# |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial version | Анжелика Кравчук |  |  |  |
| 1.1 | Review and corrections. | Владимир Тихон |  |  |  |

Contents

[1. Урок 1: Определение и вызов методов 3](#_Toc301287620)

[1.1. Что такое метод? 3](#_Toc301287621)

[1.2. Создание метода 4](#_Toc301287622)

[1.3. Вызов метода 6](#_Toc301287623)

[1.4. Создание и вызов перегруженных методов 7](#_Toc301287624)

[1.5. Использование массива параметров 8](#_Toc301287625)

[1.6. Рефакторинг для извлечения метода 10](#_Toc301287626)

[1.7. Тестирование метода 11](#_Toc301287627)

[1.8. Демонстрация: Рефакторинг и тестирование метода 15](#_Toc301287628)

[2. Урок 2: Необязательные и выходные параметры 15](#_Toc301287629)

[2.1. Необязательные параметры 16](#_Toc301287630)

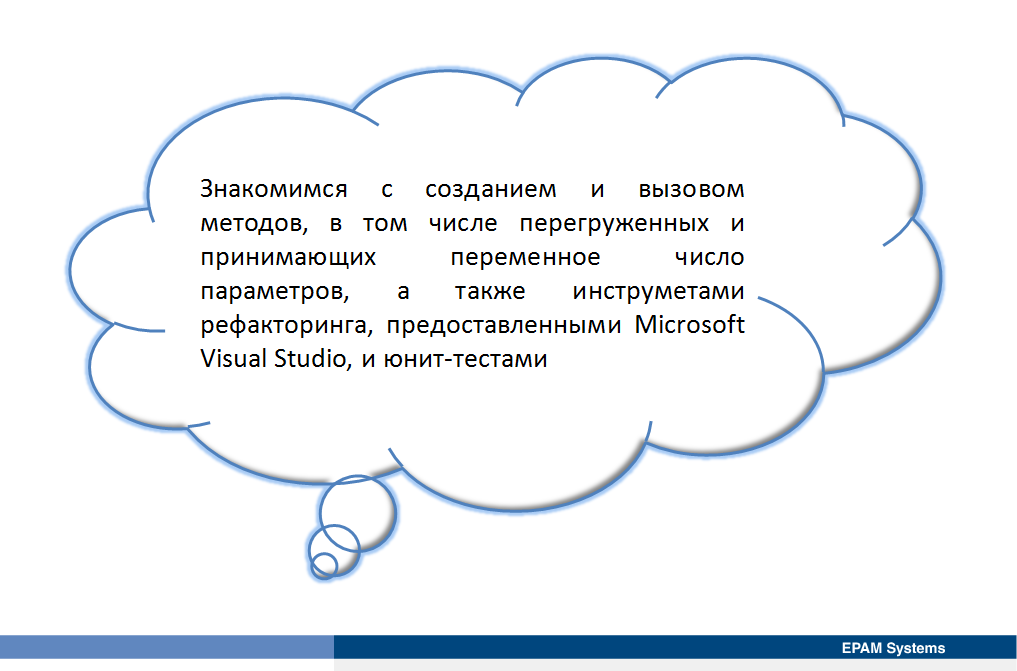
[2.2. Именованные аргументы 18](#_Toc301287631)

[2.3. Выходные параметры 19](#_Toc301287632)

[2.4. Демонстрация: Необязательные и выходные параметры, именованные аргументы 20](#_Toc301287633)

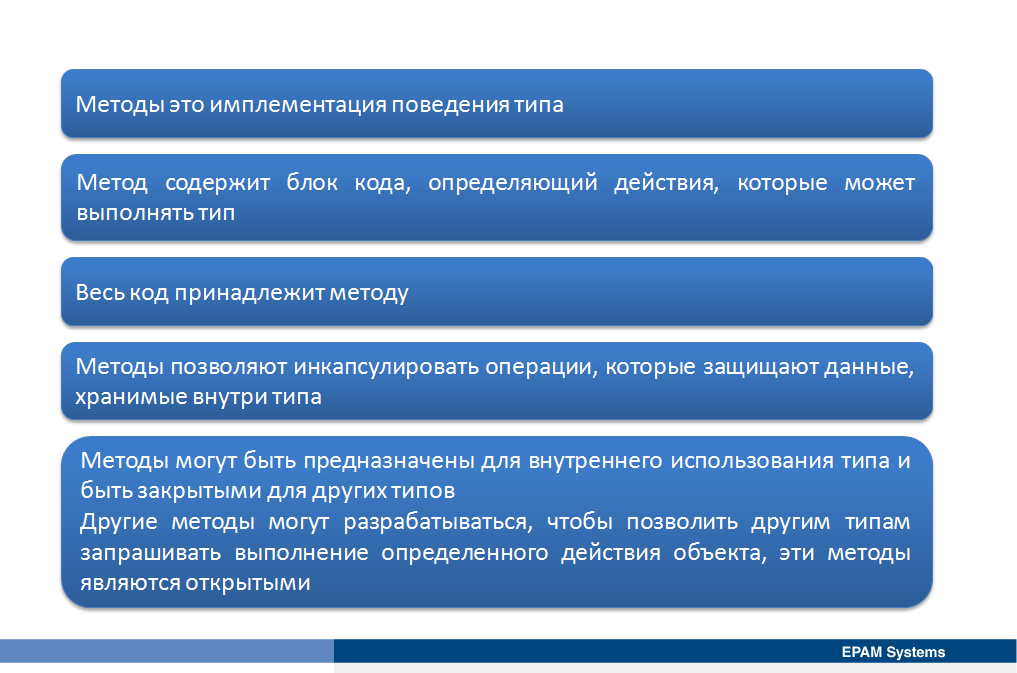
[3. Урок 3. Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения 20](#_Toc301287634)

# Урок 1: Определение и вызов методов



Урок знакомит с методами, их созданием и вызовом, объясняет, как создать перегруженные методы, а также методы, принимающие переменное число параметров. В уроке объясняется использование инструментов рефакторинга, предоставленных Microsoft Visual Studio для создания метода из существующего блока кода, и создание юнит-тестов для проверки работоспособности методов.

## Что такое метод?



Методы это имплементация поведения типа. Метод содержит блок кода, определяющий действия, которые может выполнять тип. Весь код принадлежит методу, нельзя написать программу на С#, которая не содержит по крайней мере один метод.

Возможность определения и вызова методов является одним из основных компонентов объектно-ориентированного программирования, поскольку методы позволяют инкапсулировать операции, которые защищают данные, хранимые внутри типа.

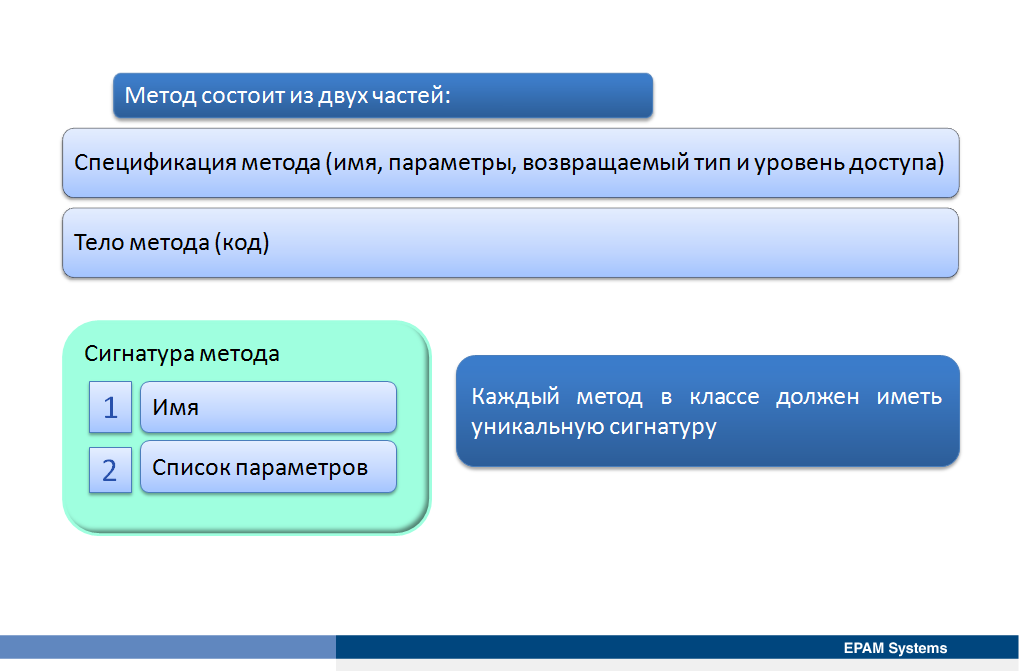
Как правило любое приложение, разрабатываемое с использованием Microsoft .NET Framework и Microsoft Visual C#, имеет много методов с конкретным назначением. Некоторые методы имеют для работы приложения основополагающее значение. Например, все приложения C# должны иметь метод, называемый Main, определяющий точку входа приложения (при запуске приложения C# CLR исполняет метод Main для этого приложения).

Методы могут быть предназначены для внутреннего использования типа, и тогда они являются закрытыми для других типов. Другие методы могут разрабатываться, чтобы позволить другим типам запрашивать выполнение определенного действия объекта, эти методы являются открытыми.

C# поддерживает два типа методов:

* *Экземплярные методы (instance methods)*. Экземплярные методы выполняются в контексте конкретного объекта, и имеют непосредственный доступ к данным, принадлежащим объекту. Например, метод ToString является экземплярным методом. Вызов экземплярных методов осуществляется c указанием имени объекта, для которого они вызываются.
* *Статические методы (static methods).* Статические методы связанны не с конкретным объектом, а с типом. К статическим методам относятся, например, методы принадлежащие классу Convert, такие как Convert.ToInt32. Эти методы вызывают, указывая не имя объекта, а тип.

## Создание метода



Метод состоит из двух частей:

* Спецификация метода.
* Тело метода.

Спецификация метода определяет имя метода, параметры, которые он может принимать, возвращаемый тип и уровень доступа метода. Сочетание имени метода и списка его параметров называются сигнатурой метода. Каждый метод в классе должен иметь уникальную сигнатуру.

Имя метода имеет те же ограничения, что и имя переменной, оно должно начинаться с буквы или символа подчеркивания и может содержать только буквы, символы подчеркивания и цифры. C# учитывает регистр, таким образом, класс может содержать два метода, имеющих одинаковое, но отличающиеся регистром одной или более букв имя, хотя это не является хорошим стилем программирования. При выборе имени метода следует соблюдать следующие рекомендации:

* *Использовать при именовании глаголы или фразы глаголов*. Это помогает другим разработчикам понять назначениие кода.
* *Использовать при именовании стиль «Pascal case».* Не начинать имена методов с символа подчеркивания или строчной буквы.

Тело метода это блок кода C#, который реализуется с использованием любой из имеющихся в C# программных конструкций. Тело метода заключается в фигурные скобки. Внутри тела метода можно определить переменные. Эти переменные существуют только во время работы метода. Когда метод завершаются, они покидают область видимости.

Параметры являются локальными переменными, которые создаются при работе метода и заполняются значениями, указанными при вызове метода. Все методы должны иметь список параметров. Параметры задаются в скобках после имени метода и разделеляются запятыми. Если метод не принимает параметров, необходимо указать пустой список параметров. Для каждого параметра необходимо указать тип и имя. В соответствии с соглашением, параметры именуются в соответствии со стилем «Camel case». Имена параметров могут предоставляться приложениям, использующим методы, с помощью Microsoft IntelliSense в Visual Studio, таким образом имена параметров сохраняют смысл.

Все методы должны иметь тип возвращаемого значения. Не возвращающий значение метод имеет тип void. Тип возвращаемого значения указывается при определении метода перед именем метода. При объявлении метода, возвращающего данные, необходимо в код метода включить оператор return.

string MyMethod()

{

return "Hello";

}

Выражение, которое специфицирует оператор return, должно иметь тот же тип, что и метод. При выполнении оператора return выражение вычисляется и передается оператору, вызвавшему метод. При этом метод завершает свою работу, поэтому любые операторы, расположеные после оператора return не выполняются. Если тип возвращаемого значения метода void, инструкцию return без значения можно использовать для завершения выполнения метода. Если ключевое слово return отсутствует, выполнение метода завершится, когда будет достигнут конец его кода. В следующем примере метод не принимает параметров и не возвращает значение.

void ClearReport()

{

// Perform some processing here.

}

Ниже показан метод, принимающий два параметра string, но не возвращающий значение.

void CreateReport(string reportName, string reportDescription)

{

// Perform some processing here.

}

В следующем примере метод принимает два параметра string и возвращает логический результат с помощью оператора **return**.

bool LockReport(string reportName, string userName)

{

bool success = false;

// Perform some processing here.

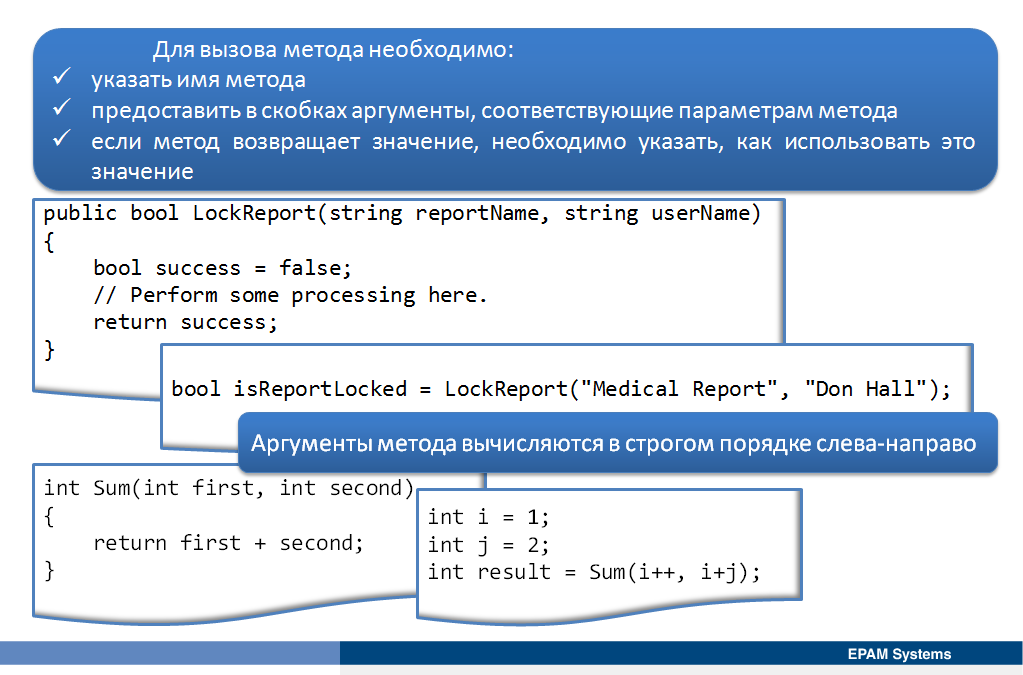
return success;

}

Любые переменные, объявленные в блоке метода доступны только другим операторам в этом блоке метода.

<http://go.microsoft.com/fwlink/?LinkId=192905>

## Вызов метода



Метод вызывается для запуска кода в нем. Для того, чтобы использовать метод, не обязательно понимать, как работает его код, можно даже не иметь доступа к этому коду, если он находится в классе сборки, например, библиотеке классов .NET Framework.

Для вызова метода необходимо указать имя метода и предоставить в скобках аргументы, соответствующие параметрам метода. Если метод возвращает значение, необходимо указать, как использовать это значение, как правило, присвоив его переменной того же типа.

Метод LockReport, вызываемый в следующем примере, блокирует сообщение для конкретного пользователя. Метод возвращает логическое значение, указывающее успешнсть выполненной операции.

public bool LockReport(string reportName, string userName)

{

bool success = false;

// Perform some processing here.

return success;

}

Метод LockReport принимает два параметра типа string. Первый параметр представляет имя отчета, который нужно заблокировать, а второй – имя пользователя, который блокирует сообщение. В следующем примере кода показано, как можно вызвать этот метод. Возвращаемое значение присваивается булевой переменной isReportLocked.

bool isReportLocked = LockReport("Medical Report", "Don Hall");

Аргументы метода вычисляются в строгом порядке слева-направо. Это важно, когда вычисление аргумента изменяет значения другого аргумента. Например, следующий метод принимает два целых параметра, складывает их и возвращает полученную сумму.

int Sum(int first, int second)

{

return first + second;

}

При вызове этого метода, значение i равное 1 используется в качестве первого аргумента, затем i увеличивается до 2, потом значение 2 + 2 используется в качестве второго аргумента. Поэтому, возвращаемое значение result будет равно 5.

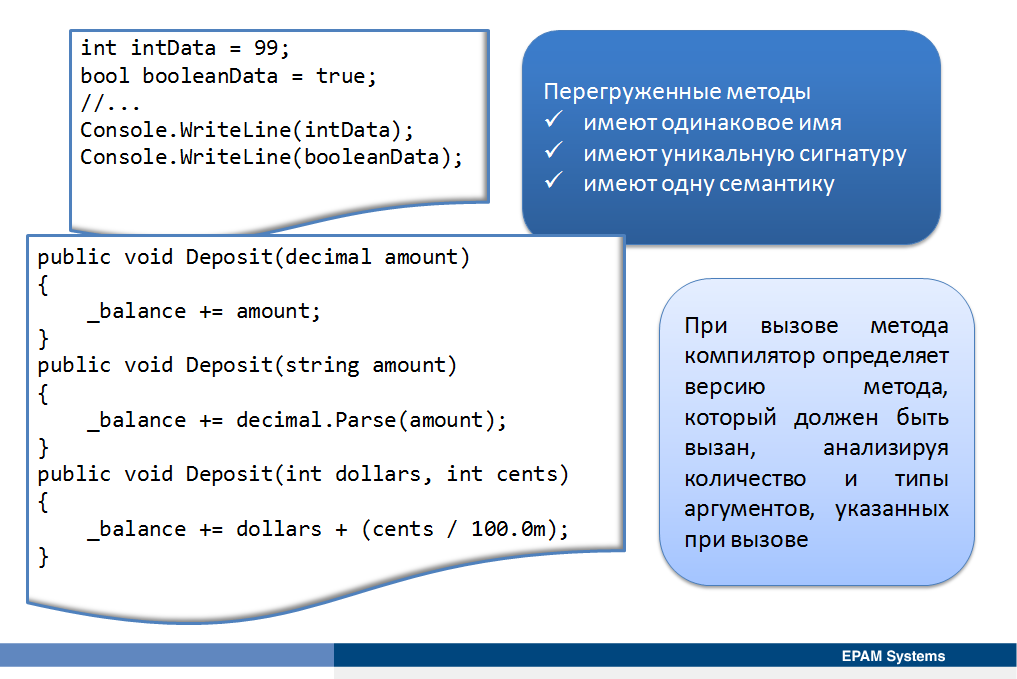
int i = 1;

int j = 2;

int result = Sum(i++, i+j);

<http://go.microsoft.com/fwlink/?LinkId=192905>

## Создание и вызов перегруженных методов



Иногда бывает полезно определить несколько реализаций метода, принимающих различный набор параметров. При этом каждая версия метода выполняет одинаковую, но работающую с различными типами, операцию. Примером такого метода в .NET Framework является метод WriteLine класса Console. Этот метод имеет 19 различных версий, которые позволяют выводить на экран разные типы данных. Например, в следующем примере кода выводятся целое и логическое значения с помощью двух операторов Console.WriteLine, типы параметров, указанные в каждом случае различны.

int intData = 99;

bool booleanData = true;

//...

Console.WriteLine(intData);

Console.WriteLine(booleanData);

Такая техника называется перегрузкой методов. Можно создать столько перегруженных версий метода, сколько необходимо, при условии, что тип и количество параметров отличаются для каждой версии (сигнатура каждого метода должна быть уникальной).

Перегрузку методов следует использовать только для создания различных методов, которые имеют одну семантику.

Перегруженные методы имеют одинаковое имя, чтобы подчеркнуть их общий смысл. Однако, каждый перегруженный метод должен иметь уникальную сигнатуру, чтобы отличить его от других перегруженных версий метода в классе. Сигнатура метода содержит имя метода и список его параметров, тип возвращаемого значения не является частью сигнатуры[[1]](#footnote-1). Таким образом, нельзя определить перегруженные методы, которые отличаются только типом возвращаемого значения. В следующем примере показано определение трех перегруженных методов Deposit в классе BankAccount:

* Первый метод Deposit принимает параметр, представляющий собой сумму для депозита в виде дробного числа.
* Второй метод Deposit принимает параметр, представляеющий сумму для депозита в виде строки.
* Третий метод Deposit принимает два параметра, представляющие сумму для депозита в долларах и центах.

publiс class BankAccount

{

private decimal \_balance;

public void Deposit(decimal amount)

{

\_balance += amount;

}

public void Deposit(string amount)

{

\_balance += decimal.Parse(amount);

}

public void Deposit(int dollars, int cents)

{

\_balance += dollars + (cents / 100.0m);

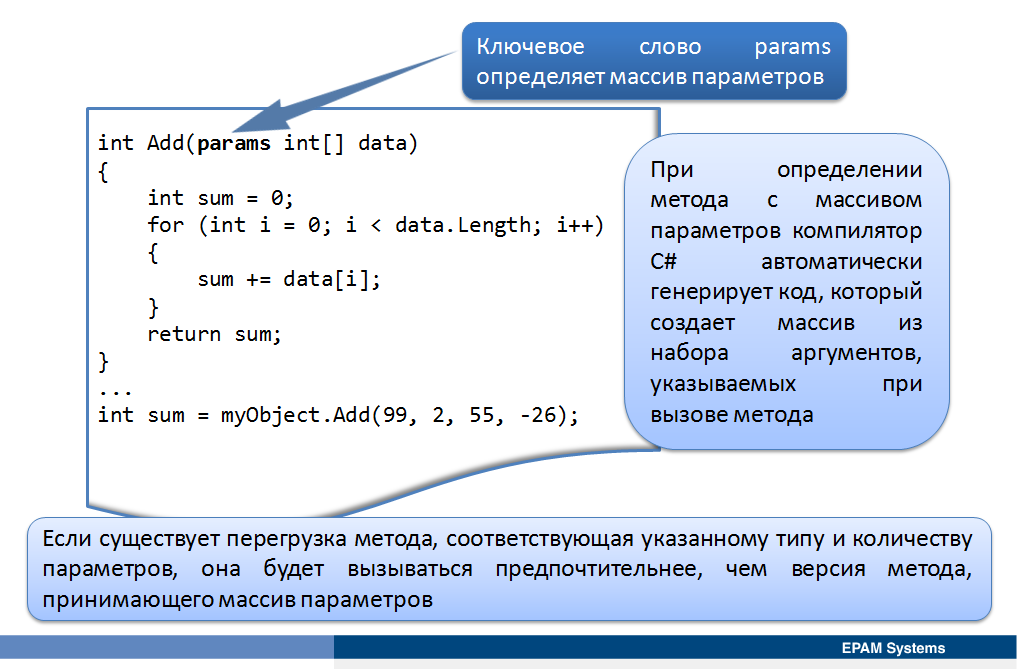
}

}

При вызове метода Deposit, компилятор определяет версию метода, который должен быть вызан, анализируя количество и типы аргументов, указанных при вызове.

http://go.microsoft.com/fwlink/?LinkId=192906

## Использование массива параметров



Перегрузка методов является полезной, но она не всегда подходящей техникой. Например, не всегда возможна перегрузка метода, принимающего переменное число параметров, особенно если не существует теоретических ограничений на их количество. Например, если неободимо определить метод Add для рассчета суммы множества целых значений, можно определить перегруженные версии этого метода следующим образом.

int Add(int one, int two)

{

return one + two;

}

int Add(int one, int two, int three)

{

return one + two + three;

}

int Add(int one, int two, int three, int four)

{

return one + two + three + four;

}

Такое решение хорошо работает, если нужно найти сумму двух, трех или четырех целых чисел, но если нужно суммировать пять, шесть, семь или даже 100 чисел? Конечно можно определить 99 перегрузок, но как далеко при этом можно зайти? Один из способов это обойти – передавать параметры методу в виде массива. В теории, не существует ограничений на размер массива (на практике, максимальный размер массива определяется объемом памяти, доступной приложению). Используя этот подход, можно определить единственную версию метода Add, которая выглядит следующим образом.

int Add(int[] data)

{

int sum = 0;

for (int i = 0; i < data.Length; i++)

{

sum += data[i];

}

return sum;

}

Недостатком такого подхода является необходимость вручную объявлять и заполнять массив данных, а затем передавать его методу каждый раз, вызывая его, как показано ниже.

int[] myData = new int[4];

myData[0] = 99;

myData[1] = 2;

myData[2] = 55;

myData[3] = -26;

//...

int sum = myObject.Add(myData);

Оптимальным решением данной проблемы является использование массива параметров. Ключевое слово params обеспечивает полезный сокращенный подход для осуществления этой техники. При определении метода с массивом параметров компилятор C# автоматически генерирует код, который создает массив из набора аргументов, указываемых при вызове метода. В следующем примере показано определение метода с ключевым словом params и его вызов с переменным числом аргументов.

int Add(params int[] data)

{

int sum = 0;

for (int i = 0; i < data.Length; i++)

{

sum += data[i];

}

return sum;

}

...

int sum = myObject.Add(99, 2, 55, -26);

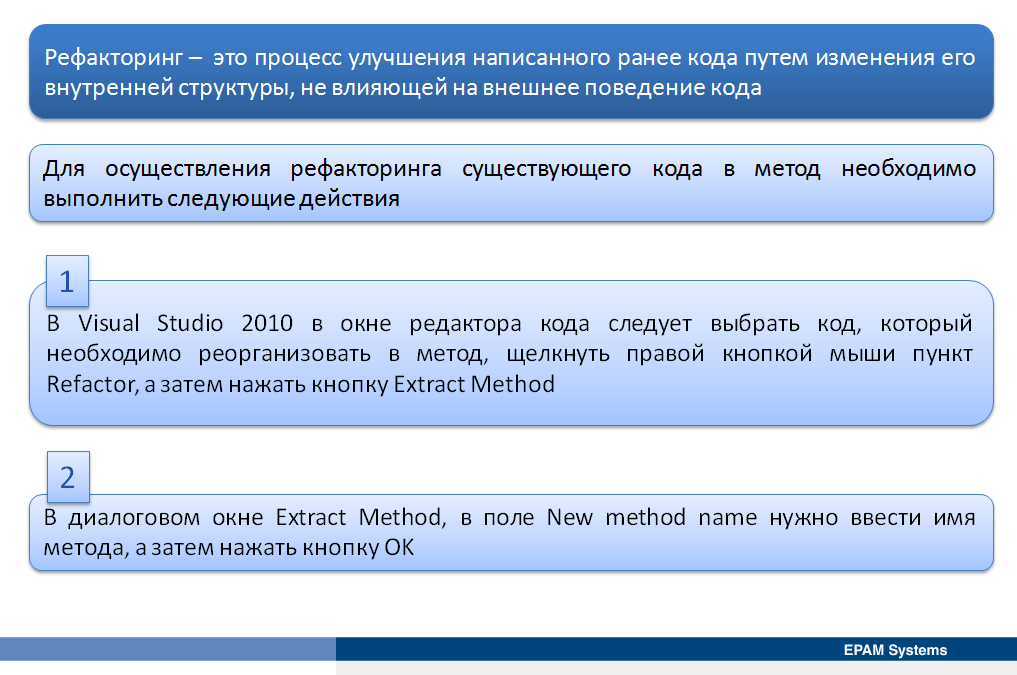
Этот метод отличается от предыдущего лишь использованием ключевого слова params. Аргументы определяются при вызове метода Add. Если аргументы имеют тип, соответствующий типу массива, заданного ключевым словом params, они собираются вместе в массив, который передается в качестве аргумента методу Add.

Если существует перегрузка метода, соответствующая указанному типу и количеству параметров, она будет вызываться предпочтительнее, чем версия метода, принимающего массив параметров.

Массив параметров можно использовать в сочетании с другими параметрами, при этом он должен быть последним параметром в списке параметров, указанных в методе. Метод может принимать в качестве параметра только один массив параметров.

http://go.microsoft.com/fwlink/?LinkId=192907

## Рефакторинг для извлечения метода



Рефакторинг – это процесс улучшения написанного ранее кода путем изменения его внутренней структуры, не влияющей на внешнее поведение кода. При написании приложения часто в его коде можно найти неоднократно записанный или очень похожий код. Когда это происходит, необходимо рассмотреть рефакторинг кода в метод. Таким образом, если логика реализуется изменениниями кода, нужно только обновить код в одном месте, что делает приложение гораздо простым в обслуживании.

Visual Studio 2010 предоставляет Extract Method мастер, который позволяет создать новый метод из существующих блоков кода. Для осуществления рефакторинга существующего кода в метод необходимо выполнить следующие действия.

1. В Visual Studio 2010 в окне редактора кода следует выбрать код, который необходимо реорганизовать в метод, щелкнуть правой кнопкой мыши пункт Refactor, а затем нажать кнопку Extract Method.
2. В диалоговом окне Extract Method, в поле New method name нужно ввести имя метода, а затем нажать кнопку OK.

Следующие примеры показывают процесс рефакторинга кода в метод, реализованный с помощью Visual Studio.

Первоначальная часть кода

string messageContents = "My message text here";

string filePath = @"C:\Users\Student\Desktop";

if (messageContents == null || messageContents == String.Empty)

{

throw new ArgumentException("Message cannot be empty");

}

if (filePath == null || !System.IO.File.Exists(filePath))

{

throw new ArgumentException("File path must exist");

}

File.AppendAllText(filePath, messageContents);

Часть кода после рефакторинга

string messageContents = "My message text here";

string filePath = @"C:\Users\Student\Desktop";

LogMessage(messageContents, filePath);

File.AppendAllText(filePath, messageContents);

private static void LogMessage(string messageContents, string filePath)

{

if (messageContents == null || messageContents == String.Empty)

{

throw new ArgumentException("Message cannot be empty");

}

if (filePath == null || !System.IO.File.Exists(filePath))

{

throw new ArgumentException("File path must exist");

}

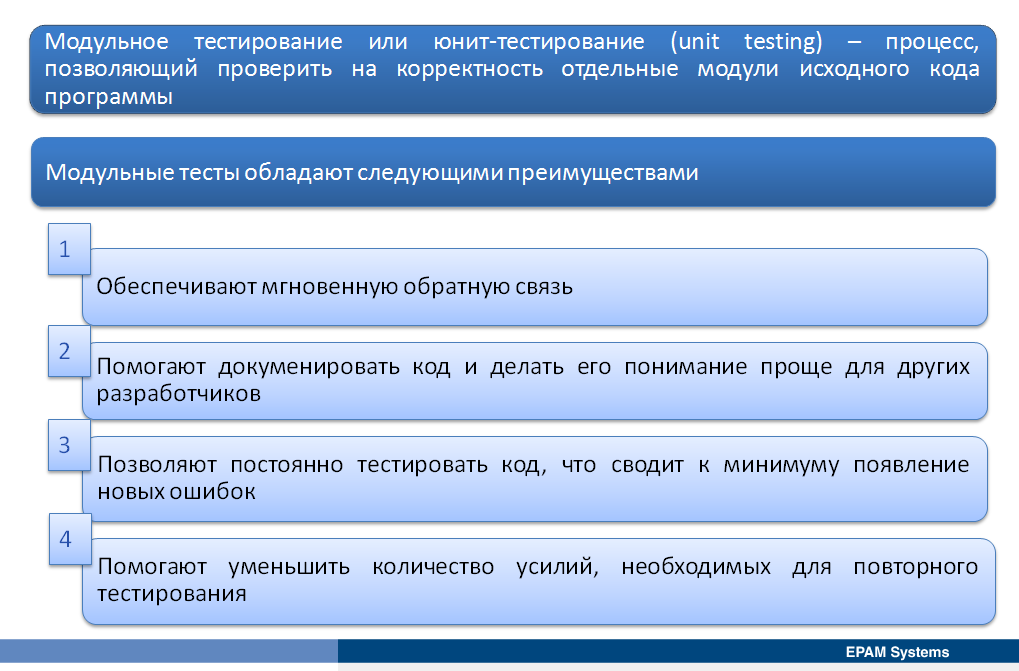
}

Следует обратить внимание на то, как Visual Studio определил переменные, которые были использованы в разделе кода, а затем включил их в качестве параметров в сигнатуру нового метода.

Кроме рефакторинга для извлечения метода, Visual Studio 2010 предоставляет другие операции рефакторинга, которые могут помочь улучшить внутреннюю структуру приложения. Например, рефакторинг для переименования, рефакторинг для инкапсуляции поля, рефакторинг для извлечения интерфейса, рефакторинг для удаления параметров, рефакторинг для упорядочения параметров.

http://msdn.microsoft.com/ru-ru/library/719exd8s.aspx

## Тестирование метода



При создании любого приложения, проверка того, что приложение функционирует правильно, должна быть частью процесса разработки и не должна игнорироваться. Кроме того, возможность повторить тест быстро и легко после изменения кода является важным принципом разработки программного обеспечения. Модульное тестирование или юнит-тестирование (unit testing) – процесс, позволяющий проверить на корректность отдельные модули исходного кода программы. Модульные тесты обладают следующими преимуществами.

* Обеспечивают мгновенную обратную связь.
* Помогают докуменировать код и делать его понимание проще для других разработчиков.
* Позволяют постоянно тестировать код, что сводит к минимуму появление новых ошибок.
* Помогают уменьшить количество усилий, необходимых для повторного тестирования.

При написание тестов следует придерживаться следующих правил. Тесты должны быть независимые, параллельные, простые и быстрые. Существует стандартный шаблон для написания тестов – «Triple A» (Arrange-Act-Assert). Согласно этому шаблону тест состоит из трех частей (Рис. 1):

* Arrange (Устанавить) – осуществить настройку входных данных для теста;
* Act (Выполнить) – выполнить действие, результаты которого тестируются;
* Assert (Проверить) – проверить результаты выполнения.

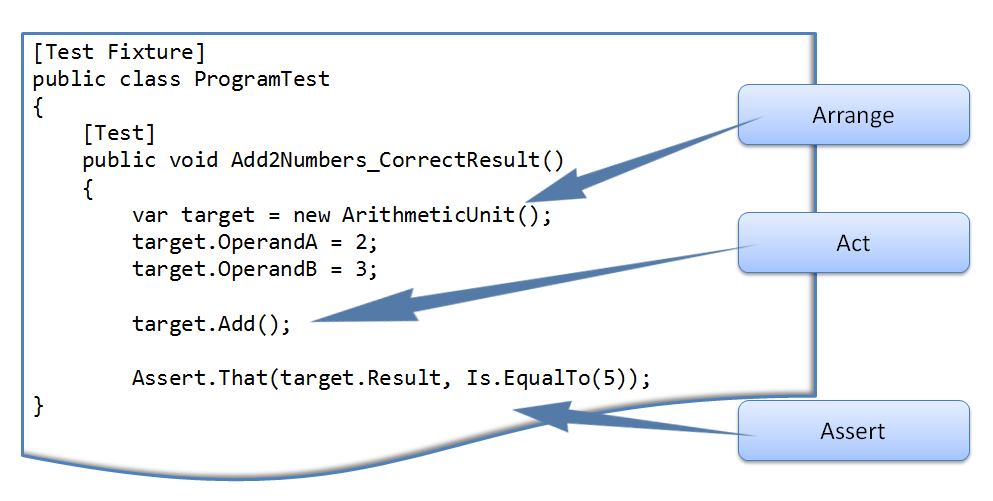


Рис. 1.

Модульные тесты в Visual Studio 2010 помогают упростить процесс тестирования и способствовать тому, чтобы код получил достаточную зону действия, не увеличив при этом количество ошибок. Юнит-тесты достигают этого, позволяя создать серию тестов, которые можно запустить в любое время, определяя, работает ли приложение все еще так, как ожидалось.

Следующие действия для создания юнит-теста предполагают, что разработан метод похожий на следующий.

public int Calculate(int operandOne, int operandTwo)

{

int result = 0;

// Perform some calculation.

return result;

}

1. В Visual Studio 2010, в окне редактора кода, содержащего метод, щелкнуть правой кнопкой мыши и выбрать команду Create Unit Tests.
2. В диалоговом окне Create Unit Tests, выполнить следующие действия, а затем нажать кнопку OK:
3. В списоке Current selection, ракрыть узлы, а затем выбрать метод, для которого создается тест.
4. В списке Output project выбрать Create a new Visual C# test project.

Если решение уже содержит Unit Test project, можно выбрать этот проект в списке Output project.

1. В диалоговом окне New Test Project, в поле Enter a name for your new project ввести имя для тестового проекта, а затем нажать кнопку Create.

При нажатии кнопки Create Visual Studio 2010 создает новый проект модульного тестирования (Unit Test project) с указанным именем, а затем добавляет этот проект в решение. Проект модульного тестирования содержит файл класса, который содержит несколько членов, наиболее значительным из которых является каркас метода теста.

В следующем примере показан код метода теста, созданный Visual Studio 2010, для проверки метода Calculate.

/// <summary>

///A test for Calculate

///</summary>

[TestMethod()]

public void CalculateTest()

{

Class1 target = new Class1(); // **TODO: Initialize to an appropriate value**

int operandOne = 0; // **TODO: Initialize to an appropriate value**

int operandTwo = 0; // **TODO: Initialize to an appropriate value**

int expected = 0; // **TODO: Initialize to an appropriate value**

int actual;

actual = target.Calculate(operandOne, operandTwo);

Assert.AreEqual(expected, actual);

Assert.Inconclusive("Verify the correctness of this test method.");

}

Метод CalculateTest содержит код для выполнения следующих задач:

* Инициализацию класса, содержащего метод Calculate.
* Инициализацию двух параметров int.
* Инициализацию параметра int для возвращаемого значения.
* Вызов метода Calculate, принимающего два параметра int.
* Определение, является ли результат, который вернулся из метода Calculate ожидаемым, используя вызов метода Assert.AreEqual.

После того как создан проект модульного тестирования и определен тестовый метод, можно запустить тест в Visual Studio. Для этого следует выполнить следующие действия:

1. В Visual Studio 2010 в меню Test выбрать пункт Windows, а затем нажать кнопку Test View. Окно Test View перечисляет все методы тестов в тестовом проекте и элементы управления, позволяющие выполнять тесты.
2. В окне Test View выбрать тесты, которые необходимо запустить, щелкнуть правой кнопкой мыши и выбрать пункт Run Selection. Если при запуске тестов необходимо отладить код, нажать кнопку Debug Selection.

Помимо встроенного в Visual Studio фреймворка MS Test, который предоставляет непосредственно Microsoft, существуют сторонние библиотеки (testing frameworks) для unit тестирования, такие как: NUnit, MBUnit, Xunit и др.

**Разработка через тестирование (test-driven development, TDD).** Разработка через тестирование – это техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и, наконец, проводится рефакторинг нового кода по соответствующим стандартам.

Разработка через тестирование требует от разработчика создания автоматизированных модульных тестов, определяющих требования к коду непосредственно перед написанием самого кода. Тест содержит проверки условий, которые могут либо выполняться, либо нет. Когда они выполняются, говорят, что тест пройден. Прохождение теста подтверждает поведение, предполагаемое программистом. Разработка через тестирование происходит в несколько этапов (Рис. 2):

* Добавление теста.
* Запуск всех тестов: убедиться, что новые тесты не проходят.
* Написание кода.
* Запуск всех тестов: убедиться, что все тесты проходят.
* Рефакторинг.
* Повторение цикла.

Рис. 2

**Добавление теста.** При разработке через тестирование, добавление каждой новой функциональности в программу, начинается с написания теста. Неизбежно этот тест не будет проходить, поскольку соответствующий код еще не написан. Если же написанный тест прошел, это означает, что либо предложенная «новая» функциональность уже существует, либо тест имеет недостатки. Чтобы написать тест, разработчик должен четко понимать предъявляемые к новой возможности требования. Для этого рассматриваются возможные сценарии использования и пользовательские истории. Новые требования могут также повлечь изменение существующих тестов. Это отличает разработку через тестирование от техник, когда тесты пишутся после того как код уже написан: она заставляет разработчика сфокусироваться на требованиях до написания кода – тонкое, но важное отличие.

**Запуск всех тестов: убедиться, что новые тесты не проходят.** На этом этапе проверяют, что только что написанные тесты не проходят. Этот этап так же проверяет сами тесты: написанный тест может проходить всегда и соответственно быть бесполезным. Новые тесты должны не проходить по объяснимым причинам. Это увеличит уверенность (хотя не будет гарантировать полностью), что тест действительно тестирует то, для чего он был разработан.

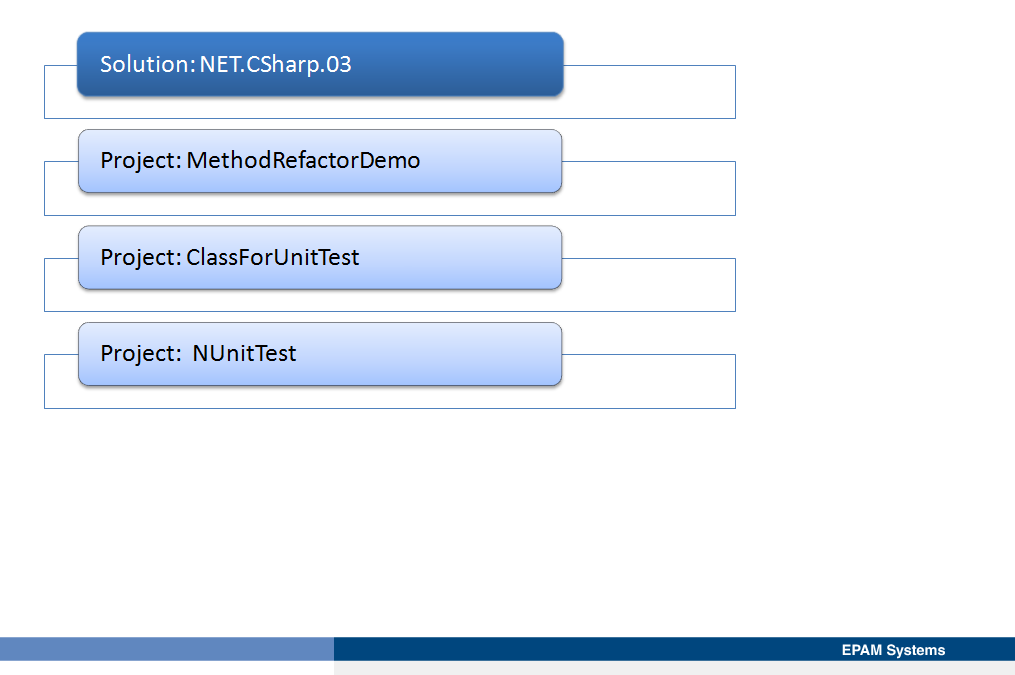
**Написание кода.** На этом этапе пишется новый код, так, что тест проходил. Этот код не обязательно должен быть идеален, допустимо, чтобы он проходил тест каким-то неэлегантным способом. Это приемлемо, поскольку последующие этапы улучшат и отполируют его. Важно писать код, предназначенный именно для прохождения теста. Не следует добавлять лишней, и соответственно не тестируемой, функциональности.

**Запуск всех тестов: убедиться, что все тесты проходят.** Если все тесты проходят, программист может быть уверен, что код удовлетворяет всем тестируемым требованиям. После этого можно приступить к заключительному этапу цикла.

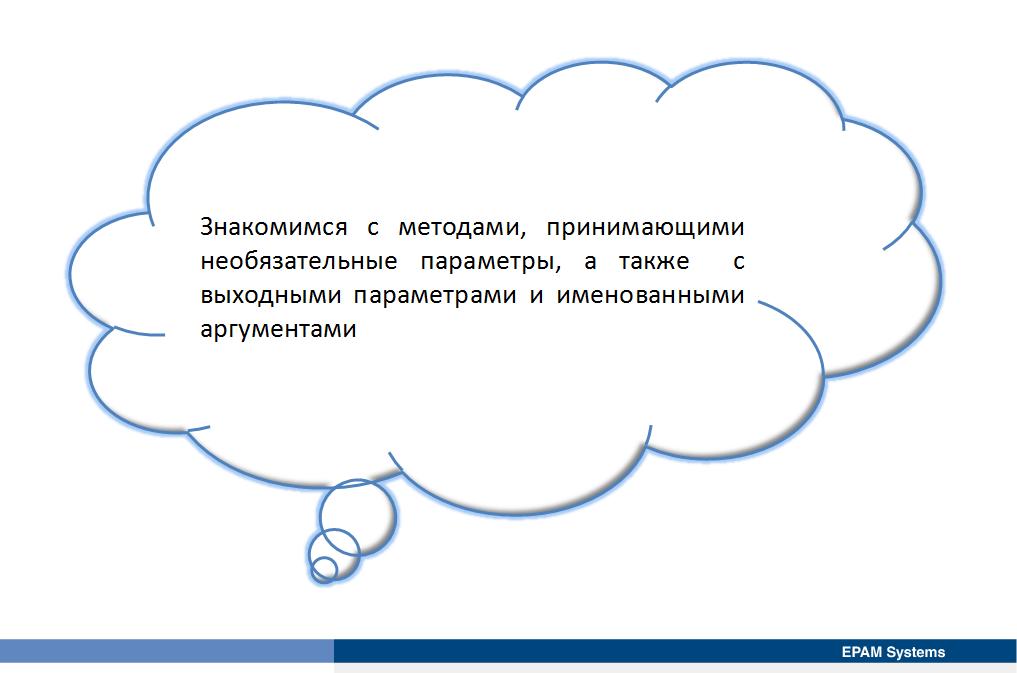
**Рефакторинг.** На этом этапе код может быть почищен. Перезапуская тесты, разработчик может быть уверен, что рефакторинг не испортит существующую функциональность. Во время рефакторинга имеет смысл устранить повторяющиеся фрагменты между тестирующим и тестируемым кодом. Например, магические числа и строчки используемые и там, и там.

**Повторение цикла.** Описанный цикл повторяется, реализуя все новую и новую функциональность. Шаги между запусками тестов следует делать небольшими, от 1 до 10 изменений. Если новый код не удовлетворяет новым тестам или старые тесты перестают проходить, программист должен вернуться к отладке.

## Демонстрация: Рефакторинг и тестирование метода



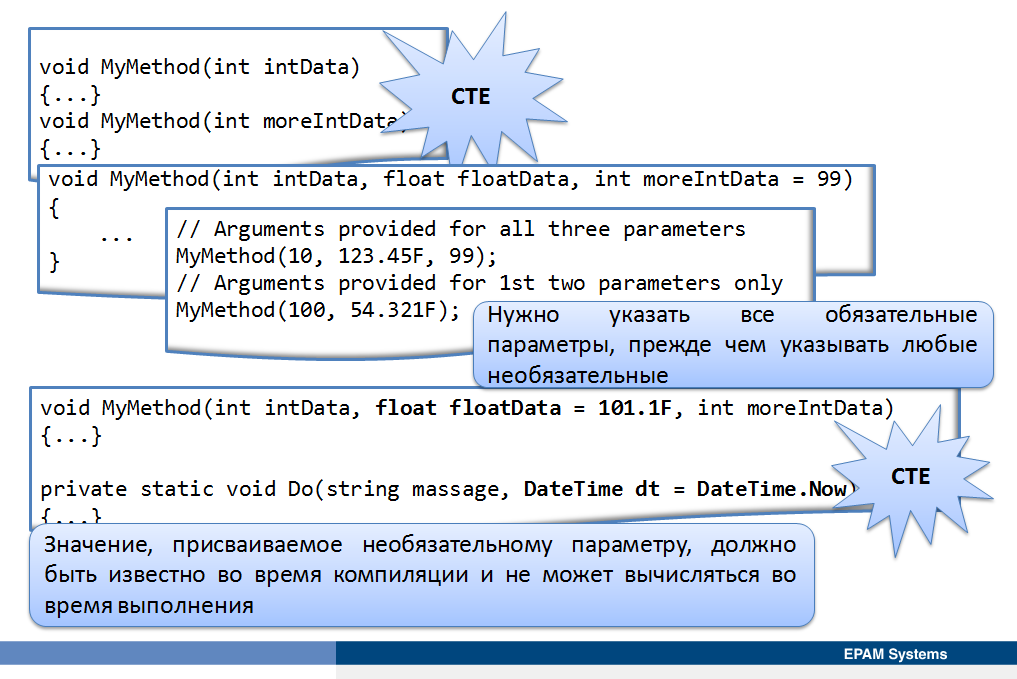
# Урок 2: Необязательные и выходные параметры



В изложенном выше материале было продемонстрировано определение методов, принимающих переменное число аргументов с помощью массива параметров. Однако, иногда нужно определить метод, имеющий фиксированное число параметров, но позволяющий приложению задать аргументы только для параметров, которые нужны. Добиться этого можно путем определения метода, принимающего необязательные (опциональные) параметры.

По умолчанию, любые аргументы, указанные во время вызова метода передаются по значению в параметры, котрые специфицирует метод. При завершении работы метода параметры будут уничтожены, и любые изменения, внесенные в значения этих параметров, будут потеряны. Выходные параметры обеспечивают механизм, позволяющий передавать данные из метода обратно в код, который вызывает метод.

## Необязательные параметры



Определяя перегруженные методы, можно реализовать различные варианты метода, которые принимают различные параметры. При создании приложения, использующего перегруженные методы, компилятор определяет, в каких конкретных случаях следует использовать каждый метод для выполнения вызова каждого метода.

Однако существует ряд языков и технологий, которые могут использоваться разработчиками для создания Windows основанных приложений и компонент, которые не соблюдают эти правила. Ключевой особенностью C# и других языков, предназначенных для .NET Framework, является способность взаимодействовать с приложениями и компонентами, написанных с использованием других технологий. Одной из основных технологий, которые использует Windows, является Component Object Model (COM). COM не поддерживает перегруженные методы, однако он использует методы, которые могут принимать необязательные параметры. Чтобы было легче включить COM библиотеки и компоненты в C# решение, C# также поддерживает необязательные параметры.

Необязательные параметры полезны также в других ситуациях. Они обеспечивают компактное и простое решение, когда не представляется возможным использовать перегрузку, поскольку типы параметров не меняются так, чтобы компилятор проводил различие между реализациями. Например, метод MyMethod принимает три параметра: два параметра типа int и параметр типа float.

void MyMethod(int intData, float floatData, int moreIntData)

{

...

}

Если нужна реализация метода MyMethod, имеющего только два параметра – int intData и float floatData, можно перегрузить метод следующим образом

void MyMethod(int intData, float floatData)

{

...

}

При написании оператора, вызывающего метод MyMethod, можно предоставить два или три параметра соответствующих типов, и компилятор использует информацию о типе, чтобы определить, какую из перегруженных версий метода вызвать.

int arg1 = 99;

float arg2 = 100.0F;

int arg3 = 101;

// Call overload with three parameters

MyMethod(arg1, arg2, arg3);

// Call overload with two parameters

MyMethod(arg1, arg2);

Если же необходимо реализовать еще две версии метода MyMethod, которые принимают только первый и третий параметр, можно попытаться осуществить эти перегрузки следующим образом.

void MyMethod(int intData)

{

...

}

void MyMethod(int moreIntData)

{

...

}

Однако эти перегруженные версии метода MyMethod имеют одинаковые синатуры, поэтому возникнет ошибка компиляции «Type ‘typename’ already defines a member called ‘MyMethod’ with the same parameter types.».

Использование необязательных параметров параметров может помочь решить эту проблему. Необязательные параметры позволяют при определении метода предоставить для параметров в списке параметров значения по умолчанию. Значения по умолчанию указываются с помощью оператора присваивания. В следующем примере показано определение метода с необязательным параметром. Все остальные параметры являются обязательными.

void MyMethod(int intData, float floatData, int moreIntData = 99)

{

...

}

При использовании необязательных параметров нужно указать все обязательные параметры, прежде чем указывать любые необязательные. В следующем примере кода возникнет ошибка компилятора.

void MyMethod(int intData, float floatData = 101.1F, int moreIntData)

{

...

}

//compile error

Еще один важный момент, о котором следует помнить при использовании необязательных параметров, состоит в том, что значение, присваиваемое необязательному параметру, должно быть известно во время компиляции и не может вычисляться во время выполнения (в этом случае на этапе компиляции сообщается об ошибке). Таким образом, объявление следующего метода

private static void Do(string massage, DateTime dt = DateTime.Now)

{

...

}

вызовет ошибку компиляции.

Вызвать метод, принимающий необязательные параметры, можно таким же образом, как вызывается любой другой метод – указав имя метода и передав необходимые аргументы. Разница с методами, которые принимают обязательные параметры состоит в том, что существует возможность пропустить соответствующие аргументы, и, тогда метод будет использовать значения по умолчанию.

В следующем примере первый вызов метода MyMethod задает значения для всех трех параметров. Второй – определяет только два аргумента, и эти значения передаются первому и второму параметрам, параметр MoreIntData принимает значение по умолчанию 99.

// Arguments provided for all three parameters

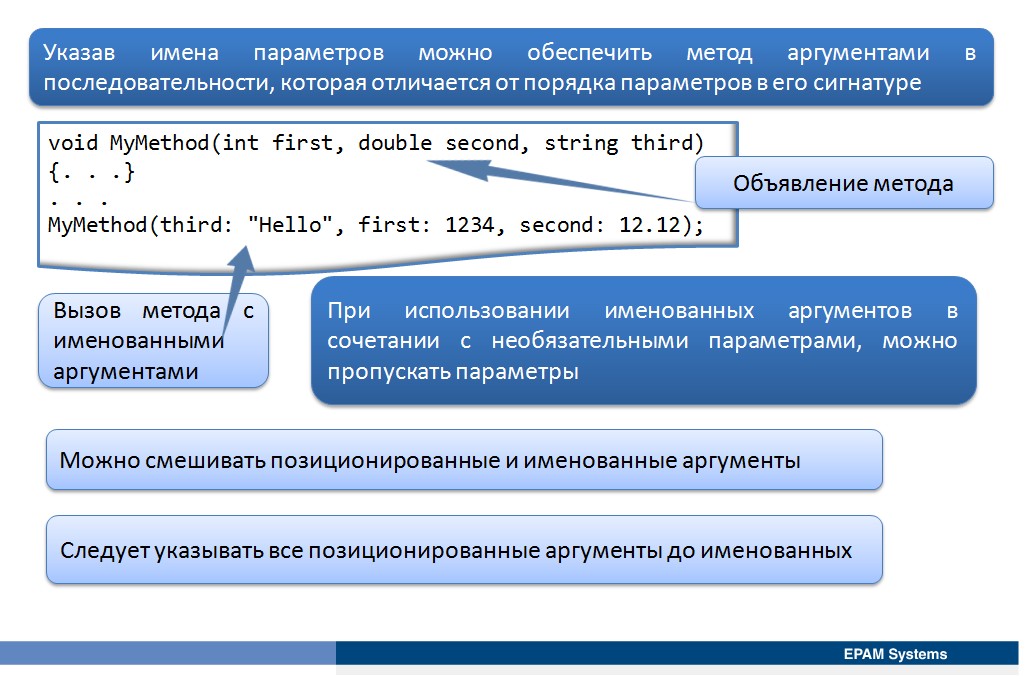
MyMethod(10, 123.45F, 99);

// Arguments provided for 1st two parameters only

MyMethod(100, 54.321F);

<http://go.microsoft.com/fwlink/?LinkId=192909>

## Именованные аргументы



Традиционно при вызове метода, порядок и положение аргументов в вызове метода должны соответствовать порядку параметров в сигнатуре метода. Если аргументы были разрегулированы и типы не соответствуют друг другу, возникнет ошибка компиляции.

В C# можно указав имена параметров, обеспечить метод аргументами в последовательности, которая отличается от порядка параметров в его сигнатуре. Для использования именованных аргументов при вызове метода необходимо указать имя параметра и соответствующие ему значение, разделенные двоеточием. В следующем примере показан соответствующи синтаксис.

// Method declaration.

void MyMethod(int first, double second, string third)

{

}

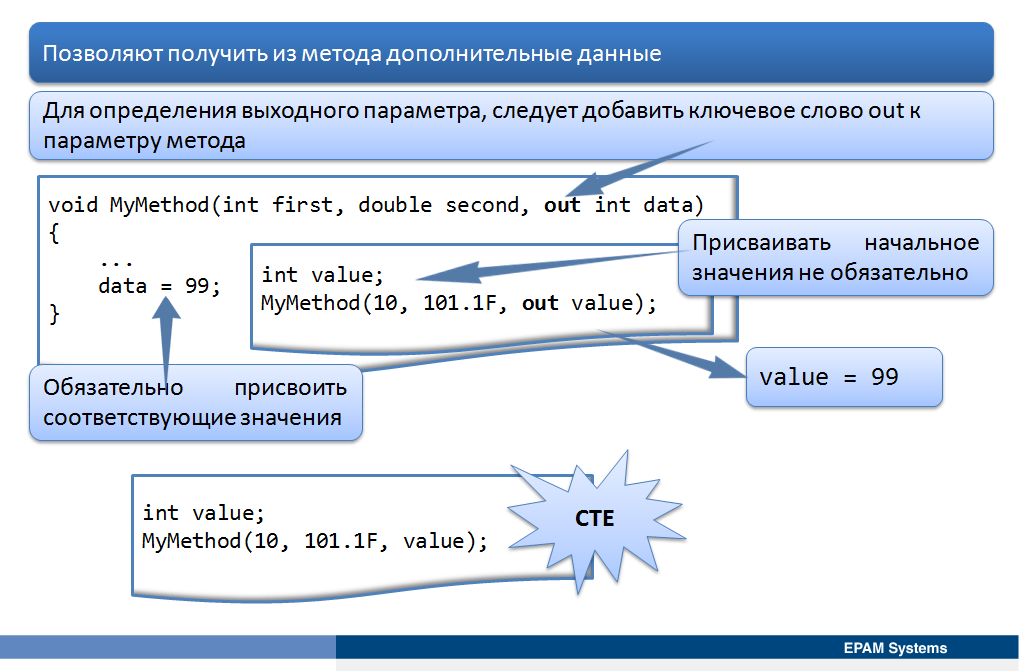
...

// Method call using named arguments.

MyMethod(third: "Hello", first: 1234, second: 12.12);

При использовании именованных аргументов в сочетании с необязательными параметрами, можно легко пропускать параметры. Эти параметры будут получать значения по умолчанию. Однако, если будет пропущен какой-либо обязательный параметр, код не будет компилироваться. Можно смешивать позиционированные и именованные аргументы. Однако, следует указывать все позиционированные аргументы до именованных.

## Выходные параметры



В методе можно указать тип возвращаемого значения и использовать оператор return для передачи значения в код, который его вызывает. Выходные параметры позволяют получить дополнительные данные из метода. При добавлении выходного параметра в метод тело метода присваивает значение этому параметру. При завершении работы метода значение выходного параметра присваивается переменной, которая определяется как соответствующий аргумент при вызове метода. Чтобы определить выходной параметр, следует добавить ключевое слово out к параметру метода.

void MyMethod(int first, double second, out int data)

{

...

data = 99;

}

Для использования выходных параметров необходимо указать переменную для соответствующего аргумента при вызове метода и добавить префикс out и к аргументу. При попытке указать аргумент, который не является переменной или пропуске ключевого слова out возникнет ошибка компиляции. Локальным переменным, передаваемым в качестве выходных параметров, присваивать начальные значения не обязательно. Методы, которым при определении (с помощью ключевого слова out) указано принимать выходные параметры, должны перед выходом обязательно присваивать им соответствующие значения (в противном случае компилятор сообщит об ошибке).

В следующем примере показано использование выходного параметра для возвращения значения из метода.

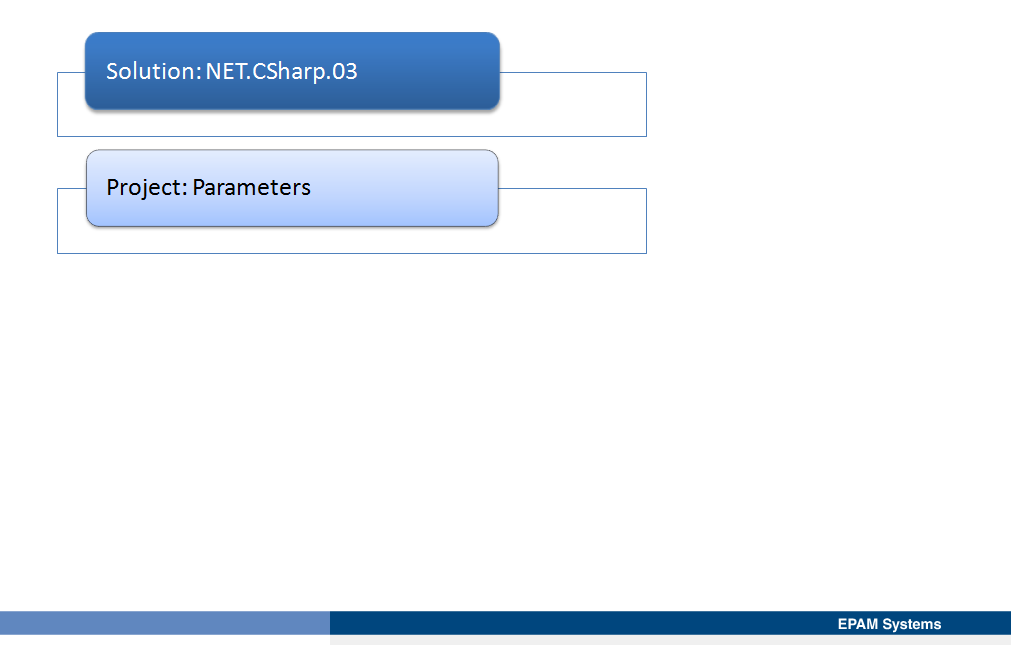
int value;

MyMethod(10, 101.1F, out value);

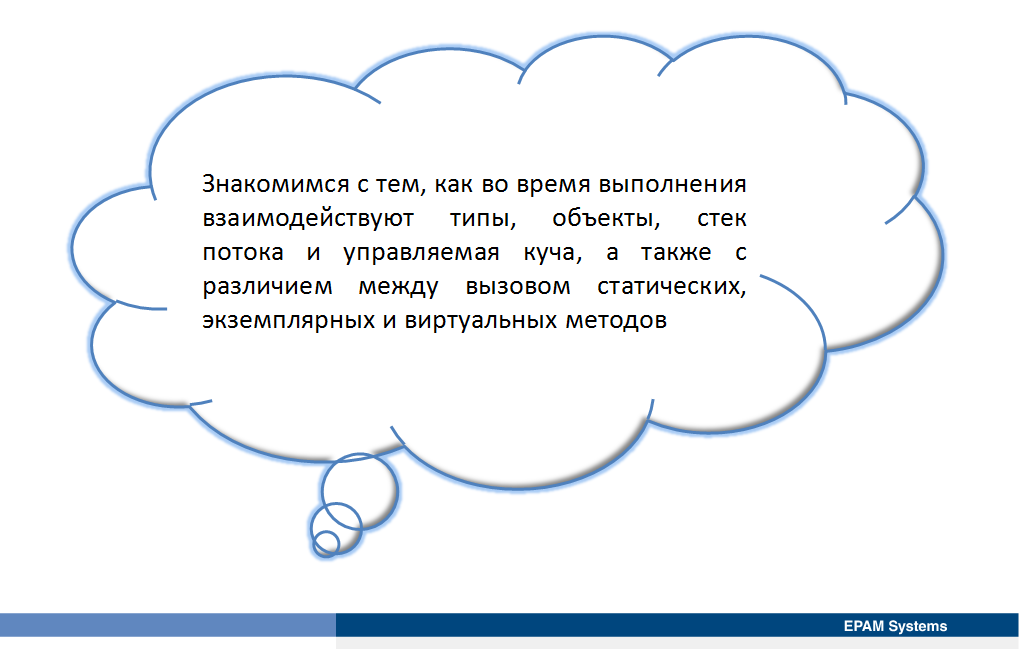
// value = 99

http://go.microsoft.com/fwlink/?LinkId=192910

## Демонстрация: Необязательные и выходные параметры, именованные аргументы



# Урок 3. Взаимодействие типов, объектов, стека потока и управляемой кучи во время выполнения



В уроке описано, как во время выполнения взаимодействуют типы, объекты, стек[[2]](#footnote-2) потока и управляемая куча, объясняется различие между вызовом статических, экземплярных и виртуальных методов.

У процесса Microsoft Windows с загруженной в него исполняющей средой CLR может быть много потоков. После создания потоку выделяется стек размером в 1 Мбайт. Выделенная для стека память используется для передачи параметров в методы и хранения определенных в пределах методов локальных переменных. Стеки заполняются от области верхней памяти к области нижней памяти (то есть от старших к младшим адресам). На рис. 3 поток уже выполняет какой-то код, и в его стеке уже находятся какие-то данные (отмечены областью более темного оттенка вверху стека). В какой-то момент времени поток выполняет код, вызывающий метод M1.

void M1()

{

string name = "Joe";

M2(name);

. . .

return;

}

Стек потока

Рис. 3. Стек потока перед вызовом метода M1

Все методы, кроме самых простых, содержат некоторый входной код (prologue code), инициализирующий метод до начала его работы. Такие методы также содержат выходной код (epilogue code), выполняющий очистку после того, как метод выполнит свою основную работу, чтобы возвратить управление вызывающей программе. В начале выполнения метода M1 его входной код выделяет в стеке потока память для локальной переменной name (Рис. 4).

void M1()

{

string name = "Joe";

M2(name);

. . .

return;

}

Стек потока

name(string)

локальные переменные метода M1

Рис. 4. Размещение локальной переменной метода M1 в стеке потока

Далее метод M1 вызывает метод М2, передавая в качестве аргумента локальную переменную name. При этом адрес локальной переменной name помещается в стек. Внутри метода М2 местоположение переменной-параметра s будет храниться в стеке. При вызове метода адрес возврата в вызывающий метод помещается в стек (Рис. 5).

void M1()

{

string name = "Joe";

M2(name);

. . .

return;

}

Стек потока

name(string)

локальные переменные метода M1

void M2(string s)

{

int length = s.Length;

int tally;

. . .

return;

}

s(string)

[return address]

парамеры

метода M2 Params

Рис. 5. При вызове М2 метод M1 помещает аргументы и адрес возврата в стек потока

В начале выполнения метода М2 его входной код выделяет в стеке потока память для локальных переменных length и tally (Рис. 6). Затем выполняется код метода М2. В конце концов выполнение М2 доходит до оператора возврата, который записывает в указатель команд процессора адрес возврата из стека, и стековый фрейм[[3]](#footnote-3) М2 возвращается в предыдущее состояние (Рис.7). С этого момента продолжается выполнение кода M1, следующего сразу за вызовом М2, а стековый фрейм метода находится в состоянии, необходимом для работы M1.

void M1()

{

string name = "Joe";

M2(name);

. . .

return;

}

Стек потока

name(string)

void M2(string s)

{

int length = s.Length;

int tally;

. . .

return;

}

s(string)

[return address]

length(int)

tally (int)

локальные переменные метода M1

парамеры

метода M2 Params

локальные переменные метода M2

Рис. 6. Выделение в стеке потока памяти для локальных переменных метода М2

void M1()

{

string name = "Joe";

M2(name);

. . .

return;

}

Стек потока

name(string)

локальные переменные метода M1

Рис. 7. Стековый фрейм М2 возвращается в первоначальное состояние

В конечном счете метод M1 возвращает управление вызывающей программе, устанавливая указатель команд процессора на адрес возврата (на рисунках не показан, но в стеке он находится прямо над аргументом name), и стековый фрейм M1 возвращается в состояние до вызова метода М2. С этого момента продолжается выполнение кода вызвавшего метода, причем начинает выполняться код, непосредственно следующий за вызовом M1, а стековый фрейм вызвавшего метода находится в состоянии, необходимом для его работы (Рис. 8).

void M1()

{

string name = "Joe";

M2(name);

. . .

return;

}

Стек потока

Рис. 8. M1 возвращает управление вызывающей функции, устанавливая указатель команд процессора на адрес возврата.

Далее поток выполняет код, из которого вызывается метод МЗ, который содержит код, призванный продемонстрировать, как работает CLR (к этому моменту управляемая куча инициализирована) (Рис. 9). При этом определены два класса

internal class Employee

{

public int GetYearsEmployed { ... }

public virtual string GetProgressReport { ... }

public static Employee Lookup(string name) { ... }

}

internal sealed class Manager : Employee

{

public override string GetProgressReport { ... }

}

**void M3()**

**{**

**Employee e;**

**int year;**

**e = new Manager();**

**e = Employee.Lookup("Joe");**

**year = e.GetYearsEmployeed();**

**e.GetProgressReport();**

**}**

**Стек потока**

**Управляемая куча**

Рис. 9. CLR загружена в процесс, его куча инициализирована, готовится вызов стека потока, в который загружен метод МЗ

В процессе преобразования IL-кода метода МЗ в машинные команды JIT-компилятор выявляет все типы, на которые есть ссылки в МЗ – Employee, int, Manager и string (из-за наличия строки «Joe»). На этом этапе CLR обеспечивает загрузку в домен AppDomain всех сборок, в которых определены все эти типы. Затем, используя метаданные сборки, CLR получает информацию о типах и создает структуры данных, собственно представляющие эти типы (структуры данных для объектов типа Employee и Manager, Рис. 10). Поскольку до вызова МЗ поток уже выполнил какой-то код, допускается, что объекты типа int и string уже созданы (что вполне возможно, так как это часто используемые типы), и они не показаны на рисунке.

**void M3()**

**{**

**Employee e;**

**int year;**

**e = new Manager();**

**e = Employee.Lookup("Joe");**

**year = e.GetYearsEmployeed();**

**e.GetProgressReport();**

**}**

**Стек потока**

**Управляемая куча**

**Manager Type Object**

**Sync block index**

**Static fields**

**Employee Type Object**

**Sync block index**

**Static fields**

**Type object ptr**

**Type object ptr**

**GetProgressReport**

**GetYearsEmployed**

**GetProgressReport**

**Lookup**

Рис. 10. При вызове МЗ создаются объекты типа Employee и Manager

Все объекты в куче содержат два дополнительных члена: указатель на объект-тип (TypeObjectPtr) и индекс блока синхронизации (SyncBlocklndex). У объектов типа Employee и Manager оба эти члена есть. Определяя тип, можно создать в них статические поля данных. Байты для этих статических полей выделяются в составе самих объектов-типов. Наконец, у каждого объекта-типа есть таблица методов с входными точками всех методов, определенных в типе. Так как в типе Employee определены три метода (GetYearsEmployed, GetProgressReport и Lookup), в соответствующей таблице методов есть три записи. В типе Manager определен один метод (переопределенный GetProgressReport), представленный в таблице методов этого типа.

После того как CLR позаботилась о создании всех необходимых для метода объектов-типов и компиляции кода метода МЗ, исполняющая среда приступает к выполнению машинного кода МЗ. При выполнении входного кода МЗ в стеке потока выделяется память для локальных переменных (Рис. 11). При этом, CLR автоматически инициализирует все локальные переменные значением null или 0 (нулем) – это делается в рамках выполнения входного кода метода.

**void M3()**

**{**

**Employee e;**

**int year;**

**e = new Manager();**

**e = Employee.Lookup("Joe");**

**year = e.GetYearsEmployeed();**

**e.GetProgressReport();**

**}**

**Стек потока**

**Управляемая куча**

**e(Employee)**

**year(int)**

**= 0**

**Manager Type Object**

**Sync block index**

**Static fields**

**Employee Type Object**

**Sync block index**

**Static fields**

**Type object ptr**

**Type object ptr**

**GetProgressReport**

**GetYearsEmployed**

**GetProgressReport**

**Lookup**

**null**

Рис. 11. Выделение памяти в стеке потока для локальных переменных метода МЗ

Далее МЗ выполняет код создания объекта Manager. При этом в управляемой куче создается экземпляр типа Manager, объект Manager. У объекта Manager – также, как и всех остальных объектов – есть указатель на объект-тип и член SyncBlocklndex. У этого объекта также есть байты, необходимые для размещения всех экземплярных полей данных, определенные в типе Manager, а также всех экземплярных полей, определенных во всех базовых классах типа Manager (в данном случае это Employee и Object) (Рис. 12). Всякий раз при создании нового объекта в куче CLR автоматически инициализирует внутренний член-указатель на объект тип так, чтобы он указывал на соответствующий объект-тип объекта (в данном случае это объект-тип Manager). Кроме того, CLR инициализирует индекс блока синхронизации (SyncBlocklndex) и присваивает всем экземплярным полям объекта значение null или 0 (нуль) перед вызовом конструктора типа, который, скорее всего изменит значения некоторых экземплярных полей. Оператор new возвращает адрес в памяти объекта Manager, который хранится в переменной е (в стеке потока).

**void M3()**

**{**

**Employee e;**

**int year;**

**e = new Manager();**

**e = Employee.Lookup("Joe");**

**year = e.GetYearsEmployeed();**

**e.GetProgressReport();**

**}**

**Стек потока**

**Управляемая куча**

**e(Employee)**

**year(int)**

**= 0**

**Sync block index**

**Instance fields**

**Manager Object**

**Manager Type Object**

**Sync block index**

**Static fields**

**Employee Type Object**

**Sync block index**

**Static fields**

**Type object ptr**

**Type object ptr**

**Type object ptr**

**GetProgressReport**

**GetYearsEmployed**

**GetProgressReport**

**Lookup**

Рис. 12. Создание и инициализация объекта Manager

Следующая строка метода МЗ вызывает статический метод Lookup объекта Employee. При вызове этого метода CLR определяет местонахождение объекта-типа, соответствующего типу, в котором определен статический метод. Затем на основании таблицы методов объекта-типа среда CLR находит точку входа в вызываемый метод, обрабатывает код JIT-компилятором (при необходимости) и вызывает полученный машинный код. Можно предположить, что метод Lookup объекта Employee выполняет запрос к базе данных, чтобы найти сведения о Joe. Получая из базы данных информацию о том, что Joe занимает, например, должность менеджера, код метода Lookup создает в куче новый объект Manager, инициализирует его данными о Joe и возвращает адрес готового объекта. Адрес размещается в локальной переменной е (Рис. 13).

**void M3()**

**{**

**Employee e;**

**int year;**

**e = new Manager();**

**e = Employee.Lookup("Joe");**

**year = e.GetYearsEmployeed();**

**e.GetProgressReport();**

**}**

**Стек потока**

**Управляемая куча**

**e(Employee)**

**year(int)**

**= 0**

**Sync block index**

**Instance fields**

**Manager Object**

**Manager Type Object**

**Manager Object**

**Type object ptr**

**Sync block index**

**Instance fields**

**Sync block index**

**Static fields**

**Employee Type Object**

**Sync block index**

**Static fields**

**JITted**

**code**

**Type object ptr**

**Type object ptr**

**Type object ptr**

**Type object ptr**

**GetProgressReport**

**GetYearsEmployed**

**GetProgressReport**

**Lookup**

Рис. 13. Статический метод Lookup в Employee выделяет память и инициализирует объект Manager значением Joe

Следует отметить, что е больше не ссылается на первый созданный объект Manager. Поскольку нет переменных, ссылающихся на этот объект, он становится подходящим кандидатом для механизма сборки мусора, который в следующем проходе освободит занятую объектом память.

Следующая строка метода МЗ вызывает невиртуальный экземпляр метода GetYearsEmployed объекта Employee (Рис. 14). При этом CLR определяет местонахождение объекта-типа, соответствующего типу переменной, использованной для вызова. В данном случае переменная е определена как Employee. (Если бы вызываемый метод не был определен в типе Employee, в процессе поиска метода CLR начала бы последовательно просматривать классы иерархии – вплоть до Object.) Далее CLR находит в таблице методов объекта-типа запись о входе в вызываемый метод, обрабатывает код JIT-компилятором (при необходимости) и вызывает полученный машинный код. Метод GetYearsEmployed возвращает какое-то значение, например, 5, то есть стаж работы Joe в компании составляет пять лет. Полученное целое число размещается в локальной переменной year в стеке потока (Рис. 14).

**void M3()**

**{**

**Employee e;**

**int year;**

**e = new Manager();**

**e = Employee.Lookup("Joe");**

**year = e.GetYearsEmployeed();**

**e.GetProgressReport();**

**}**

**Стек потока**

**Управляемая куча**

**e(Employee)**

**year(int)**

**= 5**

**Sync block index**

**Instance fields**

**Manager Object**

**Manager Type Object**

**Manager Object**

**Type object ptr**

**Sync block index**

**Instance fields**

**Sync block index**

**Static fields**

**Employee Type Object**

**Sync block index**

**Static fields**

**JITted**

**code**

**JITted**

**code**

**Type object ptr**

**Type object ptr**

**Type object ptr**

**Type object ptr**

**GetProgressReport**

**GetYearsEmployed**

**GetProgressReport**

**Lookup**

Рис. 14. Невиртуальный экземплярный метод GetYearsEmployeed в Employee возвращает значение 5

Следующая строка метода МЗ вызывает виртуальный экземплярный метод GetProgressReport в Employee. При вызове виртуального экземплярного метода CLR приходится выполнять некоторую дополнительную работу. Во-первых, CLR обращается к переменной, используемой для вызова, и затем следует по адресу вызывающего объекта. В данном случае, переменная е указывает на объект Joe типа Manager. Во-вторых, CLR проверяет у объекта внутренний член-указатель на объект-тип; этот член ссылается на фактический тип объекта. Затем CLR находит в таблице методов объекта – типа запись о входе в вызываемый метод, обрабатывает код JIT-компилятором (при необходимости) и вызывает полученный машинный код. (Рис. 15).

**void M3()**

**{**

**Employee e;**

**int year;**

**e = new Manager();**

**e = Employee.Lookup("Joe");**

**year = e.GetYearsEmployeed();**

**e.GetProgressReport();**

**}**

**Стек потока**

**Управляемая куча**

**e(Employee)**

**year(int)**

**= 5**

**Sync block index**

**Instance fields**

**Manager Object**

**Manager Type Object**

**Manager Object**

**Type object ptr**

**Sync block index**

**Instance fields**

**Sync block index**

**Static fields**

**Employee Type Object**

**Sync block index**

**Static fields**

**JITted**

**code**

**JITted**

**code**

**JITted**

**code**

**Type object ptr**

**Type object ptr**

**Type object ptr**

**Type object ptr**

**GetProgressReport**

**GetYearsEmployed**

**GetProgressReport**

**Lookup**

Рис. 15. При вызове виртуального метода GetProgressReport экземпляра Employee будет вызвана переопределенная реализация этого метода в Manager

Следует отметить, если метод Lookup в Employee обнаружит, что Joe всего лишь Employee, а не Manager, то Lookup создаст объект Employee, член-указатель на объект-тип которого ссылается на объект типа Employee, это приведет к тому, что выполнится реализация GetProgressReport в Employee, а не в Manager. Объекты типа Employee и Manager содержат члены-указатели на объекты-типы. Причина в том, что объекты-типы – это, по сути, и есть сами объекты. Создавая объекты-типы, CLR должна инициализировать эти члены. При своем запуске в процессе CLR сразу же создает специальный объект-тип для типа System.Type (он определен в mscorlib.dll). Объекты типа Employee и Manager являются «экземплярами» этого типа, и по этой причине их указатели на объекты-типы инициализируются так, чтобы ссылаться на объект-тип System.Type (Рис. 16).

**void M3()**

**{**

**Employee e;**

**int year;**

**e = new Manager();**

**e = Employee.Lookup("Joe");**

**year = e.GetYearsEmployeed();**

**e.GetProgressReport();**

**}**

**Стек потока**

**Управляемая куча**

**e(Employee)**

**year(int)**

**= 5**

**Sync block index**

**Instance fields**

**Manager Object**

**Manager Type Object**

**Manager Object**

**Type object ptr**

**Sync block index**

**Instance fields**

**Sync block index**

**Static fields**

**Employee Type Object**

**Sync block index**

**Static fields**

**Type Type Object**

**Sync block index**

**Static fields**

**. . .**

**JITted**

**code**

**JITted**

**code**

**JITted**

**code**

**Type object ptr**

**Type object ptr**

**Type object ptr**

**Type object ptr**

**Type object ptr**

**GetProgressReport**

**GetYearsEmployed**

GetProgressReport

**Lookup**

Рис. 16. Объекты типа Manager и Employee как экземпляры типа System.Type

Объект-тип System.Type сам является объектом и поэтому также имеет член-указатель на объект-тип, и ссылается на самого себя, так как объект-тип System.Type сам по себе является «экземпляром» объекта-типа (Рис. 16). Теперь становится понятно, как устроена и работает вся система типов в CLR. Метод GetType типа System.Object просто возвращает адрес, хранящийся в указателе на объект-тип указанного объекта. Иначе говоря, метод GetType возвращает указатель на объект-тип объекта, и именно поэтому всегда возможно определить истинный тип любого объекта в системе (включая объекты-типы).

1. Тип возвращаемого методом значения не является частью сигнатуры метода с точки зрения перегрузки методов. В то же время он являются частью сигнатуры метода при определении совместимости между делегатом и методом, на который он указывает. [↑](#footnote-ref-1)
2. **Стек вызовов** (call stack; применительно к процессорам – просто «стек») – в теории вычислительных систем, LIFO-стек, хранящий информацию для возврата управления из подпрограмм (процедур) в программу (или подпрограмму, при вложенных или рекурсивных вызовах) и/или для возврата в программу из обработчика прерывания (в том числе при переключении задач в многозадачной среде) [↑](#footnote-ref-2)
3. **Стековый кадр** (англ. *stack frame*) – технология передачи аргументов и выделения временной памяти (в процедурах языков программирования высокого уровня) с использованием системного стека. [↑](#footnote-ref-3)