Определения основных терминов платформы .NET Framework

# пп	Термин	Определение
1	**Application** (приложение)	Выполняющийся процесс, состоящий из одного или нескольких доменов приложения, каждый из которых состоит из одной или более сборок (**см. п. 4**). По крайней мере одна сборка должна иметь внешнюю точку доступа, с помощью которой она инициирует (или побуждает основную среду выполнения инициировать) процесс создания исходного домена приложений. Приложения могут также содержать неуправляемый код (**см. п. 81**) и ресурсы, например растровые изображения.
2	**Application Domain** (домен приложения)	Приложение состоит из одного или более доменов, которые действуют как подчиненные процессы в заранее определенных границах внутри основного процесса. Домен приложения изолирует объекты, созданные одним доменом, от объектов, созданных другим доменом приложения. Сборка (**см. п. 4**) загружается в домен приложения, а домен является наименьшей частью процесса, которая может быть выгружена.
3	**Abstraction** (абстракция)	Процесс упрощения сложной задачи. Когда программист приступает к решению какой-либо задачи, он не позволяет себе останавливаться на рассмотрении каждой детали. Наоборот, он обращает внимание лишь на детали, имеющие определяющее значение для решения задачи. "Рабочий стол – desktop" на экране монитора компьютера – пример абстракции. Он полностью скрывает детали устройства файловой системы
4	**Assembly** (сборка)	Единица развертывания и контроля версий в **.NET Framework**. Она задает пространство имен (**namespace**) для разрешения запросов по отношению к типам (**см. п. 75**) и определяет, какие типы и ресурсы будут предоставлены для внешнего доступа, а какие — только для внутреннего доступа в рамках данной сборки. Сборка включает манифест (**assembly manifest – см. п. 5**), который описывает ее содержание.
5	**Assembly Manifest** (манифест сборки)	Метаданные (**см. п. 46**), которые описывают модули (**см. п. 48**) и ресурсы, входящие в состав данной сборки, экспортируемые типы (**см. п. 75**) и упоминаемые в ней другие сборки. Эти метаданные также могут содержать необходимые, дополнительные и недопустимые для этой сборки разрешения.
6	**Base Framework**	Основная часть библиотеки Framework Class Library, которая содержит базовые классы, например Object и String. **Base Framework** и среда **CLR** (**см. п. 16**) образуют часть инфраструктуры **CLI** (**см. п. 15**).
7	**Boxing** (**см. п. 80**) (упаковка)	Преобразование базового типа значения (**см. п. 83**) в объектный (**см. п. 52**), что означает сохранение в нем всей информации о типах во время выполнения и выделение памяти в куче, в которой организована сборка мусора (**см. п. 25**). Инструкция **box** промежуточного языка **IL** (**см. пп. 14, 27**) преобразует тип значения в объектный тип за счет создания копии типа значения и вставки его во вновь созданный объект .
8	**Built-in Data Types** (встроенные типы данных)	Типы **(см. п. 75)**, предусмотренные в системе типов CLR **(см. п. 16)**. Они включают типы значений, например Intl6 и Int32 (**см. п. 83**), а также ссылочные типы, например Object и String (**см. п. 64**). Промежуточный язык IL **(см. пп. 14, 27)** поддерживает работу с этими типами с помощью специальных инструкций.
9	**CLS**	См. **Common Language Specification**, см. **п. 17**
10	**CLS Compliant** (**CLS**-совместимость)	Элемент среды CLR (**см. п. 16**) называется CLS-совместимым, если он содержит открытыми только те языковые компоненты, которые описаны в спецификации CLS (Common Language Specification, **см. п. 17**). "Открытыми" называются элементы, видимые за пределами своей сборки (**см. п. 4**). Например, метод (**см. п. 47**) называется CLS-совместимым, если он использует в своем списке параметров только CLS-совместимые типы. Этот метод может также содержать CLS-несовместимые, но скрытые в своей сборке типы. CLS-совместимыми могут быть классы, интерфейсы (**см. п. 32**), компоненты, инструменты и многое другое. (Описание всех CLS-правил, включая этот пример, вы найдете в спецификации ECMA – **E**uropean **C**omputer **M**anufacturer's **A**ssociation.)

11	**CLS Frameworker** **(CLS**-создатель платформ**)**	Язык или инструмент, который позволяет разработчикам создавать целые CLS-совместимые платформы для использования в CLI-инфраструктуре (**см. п. 15**).
12	**Coercion** **(**приведение типа**)**	Попытка преобразования значения одного типа (**см. п. 75**) в значение другого типа. Приведение к более широкому типу, например значения типа Int32 к значению типа Int64, сохраняет информацию. Приведение к более узкому типу, например значения типа Int64 к значению типа Int32, иногда приводит к утрате информации.
13	**COM Interoperability – COM Interop** (способность к взаимодействию)	Механизм в среде выполнения, который позволяет взаимодействовать технологиям COM (**C**omponent **O**bject **M**odel – модель компонентных объектов Microsoft) и CLR (**см. п. 16**).
14	**Common Intermediate Language – CIL** **(**общий промежуточный язык**)**	Язык, используемый для выходных данных компиляторов и для входных данных JIT-компилятора (**см. пп. 35, 36, 37**). Он определяет абстрактную архитектуру выполнения на основе стека (**см. п. 67*а***). Среда CLR может включать несколько JIT-компиляторов для преобразования CIL-языка в собственный машинный код (**см. п. 49**) используемой платформы.
15	**Common Language Infrastructure – CLI** **(**общая языковая инфраструктура**)**	Эта инфраструктура является комбинацией среды Common Language Runtime (**см. п. 16**), базовой платформы Base Framework (**см. п. 6**) и других библиотек. CLI-инфраструктура — это основной компонент стандарта ECMA – **E**uropean **C**omputer **M**anufacturer's **A**ssociation.
16	**Common Language Runtime – CLR** (общеязыковая исполняющая среда)	Система типов (**см. п. 79**), метаданных (**см. п. 46**) и выполнения кода в рамках платформы .NET Framework, которая предлагает управляемый код (**см. п. 43**) и данные вместе с такими службами, как межъязыковая интеграция, безопасность доступа кода, управление выполняемыми объектами, а также отладка и оптимизация кода. С помощью CLR компиляторы и другие инструменты могут предлагать эти сервисы разработчикам.
17	**Common Language Specification – CLS** **(**общеязыковая спецификация**)**	Подмножество компонентов платформы .NET Framework, которые поддерживаются широким набором совместимых языков и инструментов. CLS-совместимость применима только к открытым аспектам типов (**см. п. 75**). CLS-совместимые языки и инструменты могут гарантированно взаимодействовать с другими CLS-совместимыми языками и инструментами. Например, тип Int32 является CLS-совместимым, поэтому другим CLS-совместимым языкам и инструментам известны корректные способы использования этого типа.
18	**Contract** **(**контракт**)**	Обязательство между поставщиком и пользователем, например объявление метода в интерфейсном типе (**см. п. 32**). При добавлении типа в интерфейсный тип он поддерживает контракт, заданный в интерфейсном типе. Контракты могут иметь строгую форму, например требовать строгого соответствия сигнатуры и контракта (т.е. корректность синтаксиса, **см. п. 47**), или нестрогую форму, например предполагать последовательное поведение всех классов, которые реализуют общий контракт (т.е. корректность семантики). Семантические контракты гораздо сложнее создавать и проверять.
19	**Delegate** **(**делегат**)**	Делегат является классом, вследствие этого, ссылочным типом (**см. п. 64**) и происходит от базового класса **System.Delegate**. Как и любой другой класс, делегат**: а)** должен быть определен и **б)** впоследствии может быть порожден. **Порождение класса**, происходящее во время исполнения программы, называется **объектом**. Для термина “делегат” не существует подобного разграничения. **И само определение (а) и его порожденный объект (б)** обозначаются одним и тем же словом — " delegate **–** делегат". Для создания и определения производного делегата не используется привычный синтаксис образования классов (знак двоеточия " **:** "). Вместо этого применяется ключевое слово **delegate,** как показано ниже: public delegate double Calculation(int x, int y); Здесь определен делегат Calculation**,** который может инкапсулировать *любой* метод (в том числе и метод – обработчик событий), имеющий два параметра типа **int** и возвращаемое значение метода типа **double.** Делегаты в C# применяются по двум причинам: **1)** они поддерживают события (**см. п. 20**); **2)** они предоставляют возможность выбора вызываемого метода во время выполнения программы (то есть динамически), а не во время компилирования. **Делегат**, как **и** наследование, интерфейс и полиморфизм, также **помогает отложить решение о реализации методов, до времени исполнения**.

20	**Event** **(**событие**)**	Событие – это нечто, инициируемое действием (z.B., "click" мышкой) пользователя при работе с **GUI.** Этим нечто является реакция системы, вызванная "click'ом" мышкой. Алгоритм реагирования системы состоит из **четырех** шагов, которые заключаются в следующем: 1. Объявить объект (z.B., CalculateButtonHandler) стандартного класса EventHandler**:** **EventHandler CalculateButtonHandler;** 2. Создать объект класса EventHandler, который определяет имя метода – обработчика событий (z.B., btnCalculate_Click) в качестве единственного аргумента конструктора этого класса (объект инкапсулирует метод)**:** **CalculateButtonHandler = new EventHandler(btnCalculate_Click);** 3. Зарегистрировать объект класса EventHandler для того объекта **GUI** (z.B., btnCalculate), чьи события он должен обрабатывать (**Click** – стандартный идентификатор обрабатываемого события**;** оно происходит тогда, когда пользователь "click'нет" мышкой): **btnCalculate.Click += CalculateButtonHandler;** 4. Запрограммировать обработчик событий который оформлен в виде **метода** **btnCalculate_Click(Object sender, System.EventArgs e)** { **…** } На фундаментальном уровне **событие фактически является вызовом метода** (4-й шаг)**.**
21	**Encapsulation** **(**инкапсуляция**)**	Способ связывания атрибутов и процедур для формирования объекта (класса). Инкапсуляция позволяет программисту реализовать проверку правильности использования атрибутов и процедур, поместив концептуально идентичные атрибуты и процедуры в класс, а затем в классе определив правила для управления доступом к ним.
22	**Exception** **(**исключительная ситуация**)**	Механизм уведомления об ошибках времени выполнения, который требует осуществить их обработку или анализировать стек (**см. п. 67*a*, 68**) до тех пор, пока не будут обработаны (перехвачены) все сообщения об ошибках или поток не прекратит свою работу.
23	**Executable File** **(**выполняемый файл**)**	Файл в компактном формате Portable Executable (PE) (**см. пп. 55, 59**), который может быть загружен в память и выполнен загрузчиком операционной системы. Он может иметь вид EXE-файла или DLL-файла (**D**ynamic-**L**ink **L**ibrary – динамически подключаемая библиотека).
24	**Execution System** **(**система выполнения**)**	Система среды CLR (**см. п. 16**), которая выполняет управляемый (**см. п. 43**) и неуправляемый (**см. п. 81**) код. Система выполнения использует в качестве входных данных код на промежуточном языке (Intermediate Language — IL, **пп. 14, 27**) или предварительно откомпилированный машинный код, а также метаданные (**см. п. 46**), связанные с кодом. Система выполнения отвечает за загрузку и выгрузку кода, проверку, управление памятью, обеспечение безопасности, обработку исключительных ситуаций (**см. п. 22**), а также размещение объектов.
25	**Garbage Collection** **(**сборка мусора**)**	Процесс транзитивного перехода по всем указателям на активно используемые объекты для обнаружения всех объектов, на которые ссылаются другие объекты. Он нужен для управления повторным использованием памяти в куче, которая не занята объектами, обнаруженными во время этого поиска. Сборка мусора в среде CLR (**см. п. 16**) также упорядочивает в компактном виде используемую память для сокращения рабочего пространства кучи.
26	**GC**	См. Garbage Collection, п. **25**
27	**IL**	См. Common Intermediate Language, п. **14**
28	**Imperative Security Check** (императивная проверка безопасности)	Проверка безопасности, которая возникает при вызове метода (**см. п. 47**) внутри защищенного кода. Этот тип проверки может быть связан с использованием данных или изолирован в некотором месте объекта или метода. Например, если имя защищенного файла известно только во время выполнения, то императивная проверка безопасности может быть связана с передачей имени файла в качестве параметра метода. (**Императивные** ограничения безопасности задаются и проверяются во время выполнения, а не компиляции, как это происходит при **декларативной** проверке безопасности.)
29	**Inheritance** **(**наследование**)**	Способ получения атрибутов и поведения одного объекта (класса) другим при наличии того, что программисты называют отношением "**Is A**" («является»). **Ассоциация** (**"Has A"** – **"**содержит**"**)**:** ассоциация показывает, что **один объект** содержит **другой**.
30	**Instance Fields** (**см. п. 69**) **(**поля экземпляра**)**	Поля, которые в каждом отдельном экземпляре типа (то есть класса) содержат собственное значение. В других языках и средах программирования они также называются членами данных (data members).

31	**Instance Methods** (**см. п. 70**) (методы экземпляра)	Методы, вызываемые по отношению к экземпляру, а не к типу (то есть классу). Для доступа к полям данного экземпляра эти методы используют указатель **this** (**см. п. 73**), который иногда может иметь неопределенное значение (null).
32	**Interface** (интерфейс)	Синтаксически интерфейсы аналогичны классам, но методы (а также свойства, индексаторы и события) интерфейсов объявляются без тела. Интерфейс определяет, что должен делать класс, но не конкретизирует, как он это должен делать. Класс реализует интерфейс путем создания набора методов (а также свойств, индексаторов и событий) , определенных в интерфейсе. Интерфейс это контракт, который **обязан** реализовать класс.
33	**Intermediate Language (IL)**	См., Common Intermediate Language, **см. 14**
34	**Interoperability** (взаимодействие)	Механизм среды выполнения, который позволяет среде CLR сотрудничать с другими моделями приложений. Эти другие приложения могут быть компонентами неуправляемого кода (**см. п. 81**) с доступом через PInvoke (**см. п. 57**) или компонентами COM (**C**omponent **O**bject **M**odel – модель компонентных объектов Microsoft) с доступом через COM Interop (**см. п. 13**).
35	**JIT**	См. Just in time **см. 36**
36	**Just in time** (по мере необходимости)	Характеристика действия, которое происходит только в случае необходимости, например компиляция в случае необходимости, или JIT-компиляция (JIT compilation), либо активация в случае необходимости объекта, или JIT-активация (JIT object activation). Иногда сокращение JIT также используется для обозначения JIT-компилятора (JIT compiler, **см. п. 37**).
37	**JIT Compiler**	(JIT-компилятор). Компилятор, который преобразует код на языке IL (**см. пп. 14, 27**) в машинный код только в случае необходимости во время выполнения. Его также называют JITter. Среда CLR (**см. п. 16**) может содержать несколько JIT-компиляторов.
38	**Library** (библиотека)	Множество типов (то есть классов и др.), которые обеспечивают абстракцию и реализацию службы. Такие библиотеки часто называются библиотеками классов (**см. п. 6**), но они могут содержать не только классы. На самом деле они могут включать любые типы среды CLR (**см. п. 16**), в том числе классы, интерфейсы (**см. п. 32**) и типы значений **struct** (**см. п. 83**). Набор библиотек образует профиль (**см. п. 61**).
39	**Lifetime** (жизненный цикл)	Промежуток времени, который начинается в момент выделения памяти для объекта и заканчивается в момент удаления последней ссылки на этот объект, т.е. момент времени, когда сборщик мусора (**см. пп. 25, 26**) может удалить его из памяти.
40	**Load Instruction** (инструкция загрузки)	Инструкция языка IL (**см. пп. 14, 27**), которая загружает значения, например локальные переменные, в стек выполнения (**см. п. 67*a***).
41	**Local** (локальное расположение)	Место в стеке (**см. п. 67*a***), выделенное методом (**см. п. 47**) для собственных нужд, например для объявленной пользователем локальной переменной или для временного хранения промежуточных значений (т.е. когда компилятор исчерпает все регистры).
42	**Location** (расположение)	Область в памяти, в которой может быть сохранено значение и контракт расположения (location contract), указывающий сохраняемый тип. Контракт расположения указывает как может использоваться хранимое в этом месте значение. Например, расположение может указывать, что в нем хранится ссылка на интерфейсный тип. Это значит, что хранимое в этом расположении значение может быть доступно, только имея интерфейсный тип, даже если его точный тип (**см. п. 52**) может обладать большей функциональностью. В каждый момент времени в расположении может храниться только одно значение.
43	**Managed Code** (управляемый код)	Код, который поставляет метаданные (**см. п. 46**), необходимые среде CLR (**см. п. 16**) для предоставления таких служб, как управление памятью, межъязыковая интеграция, обеспечение безопасности доступа кода, а также автоматическое управление объектами. Весь код на языке IL (**см. пп. 14, 27**) исполняется как управляемый.
44	**Managed Pointer** (**см. пп. 58, 73**) (управляемый указатель)	Указатель на поле внутри объекта, который может быть обработан сборщиком мусора, обычно созданный компилятором для временного использования (например, указатель на отдельный символ внутри строки, которая может быть обработана сборщиком мусора). Сборщик мусора (**см. п. 25**) должен иметь возможность

		обновлять все внутренние указатели во время компактной упаковки места в куче. Для корректной сборки мусора каждый внутренний указатель на объект должен иметь видимую сборщику мусора ссылку на весь этот объект.
45	**Member** (член)	Конструктор, поле, метод, свойство или событие типа (**см. п. 75**). Существуют статические члены (члены класса, (**см. п. 69, 70**)) или члены экземпляра (**см. п. 30, 31**). Вложенный класс не является членом объемлющего его класса.
46	**Metadata** (метаданные)	Данные (или информация) о данных. В среде CLR (**см. п. 16**) метаданные используются для описания сборок (**см. п. 4**) и типов (**см. п. 75**). Эта информация хранится в выполняемых файлах (**см. п. 23**) и используется компиляторами, инструментами и системой выполнения для предоставления широкого диапазона сервисов. Метаданные имеют очень большое значение: **1)** для получения информации о типах во время выполнения; **2)** и для динамического вызова методов (**см. п. 60**). Во многих архитектурах и системах также используются метаданные, например типы библиотек в технологии COM (**C**omponent **O**bject **M**odel) также содержат метаданные.
47	**Method** (метод)	Набор выполняемых инструкций, который может быть вызван извне. Каждый метод имеет свой **контракт**, т.е. имя, несколько параметров, а также возвращаемый тип (**см. п. 18**). Клиенты, которым требуется вызвать этот метод, должны удовлетворять контракту в момент вызова метода. Методы могут быть разных видов, например статические методы (**см. п. 70**) или методы экземпляра (**см. п. 31**).
48	**Module** (модуль)	Один PE-файл (**см. пп. 55, 59**). Если этот файл содержит манифест (**см. п. 5**), то в таком случае он образует сборку (**см. п. 4**) и может быть загружен загрузчиком типов среды CLR (**см. п. 16**). Если этот файл не имеет манифеста, то для загрузки он должен быть частью сборки. Созданные на разных языках модули могут быть собраны в одной сборке.
49	**Native Code** (машинный код)	Код, откомпилированный в специализированном виде для данного процессора.
50	**Object-Oriented Programming** (объектно-ориентированное программирование)	Стиль программирования, основанный на концепциях идентичности (см. п. **21**), классификации, наследования (см. п. **29**), полиморфизма (см. п. **60**) и сокрытия ненужной информации (см. п. **21**). Самые первые попытки использовать этот стиль программирования были реализованы в языке Simula 67.
51	**Object Reference** (объектная ссылка)	Ссылка на объект в куче, в которой проводится сборка мусора. Информация об объектных ссылках передается сборщику мусора (**см. пп. 25, 26**), чтобы они могли быть изменены при перемещении их объектов по куче.
52	**Object Type** (**см. п. 64**) (объектный тип)	Подмножество ссылочного типа. В некоторых языках программирования (z.B, C#) объектные типы называются классами. Значения (экземпляры) объектного типа называются объектами. Объектные типы являются точными типами, т.е. полностью удовлетворяют всем функциональным требованиям (контрактам), которые реализует тип.
53	**Override** (переопределение)	В системе безопасности это означает динамическое изменение оценки последующих требований безопасности. Например, утверждение разрешения может вызвать удачный исход проверки требования, а отказ от разрешения или его исключительный доступ могут вызвать неудачный исход проверки требования, который в противном случае не имел бы места.
54	**Parameter Area** (область параметров)	В системе выполнения это часть фрейма стека метода (**см. п. 67**), т.е. записи активизации (activation record), используемой для хранения **параметров, которые передаются методу при его вызове** (**см. п. 47**).
55	**PE file**	См. Portable Executable file, **см. п. 59**
56	**Platform Invocation Service** (служба платформенных вызовов)	Служба, которая позволяет управляемому коду (**см. п. 43**) вызывать неуправляемый машинный код (**см. п. 81**).
57	**Platform Invoke** (PInvoke)	См. Platform Invocation Services, **см. п. 56**
58	**Pointer Type** (**см. п. 44, 52, 64**) (указательный тип)	Представитель подмножества ссылочных типов, который может указывать на данные или код. Он имеет вид адреса и может представлять собой управляемый указатель (**см. п. 44**), неуправляемый указатель или неуправляемый указатель функции.

59	**Portable Executable File** (компактный выполняемый файл)	Формат файла, используемый для выполняемых программ, а также для файлов, связанных вместе для образования выполняемых программ.
60	**Polymorphism** (полиморфизм).	Полиморфизм означает, что что-то может существовать во многих формах – это что-то представляет собой метод в объектно-ориентированном языке программирования. Полиморфизм реализуется с помощью переопределения **(см. п. 53)** методов (позднее связывание, динамическое связывание или полиморфизм времени выполнения программы, **см. п. 85**) или с помощью перегрузки методов (раннее связывание или полиморфизм времени компиляции).
61	**Profile** **(**профиль**)**	Спецификация набора библиотек (**см. п. 38**), каждая из которых представляет одну или несколько служб, а также набора требований для языка **IL** (**см. п 14, 27**), которые обеспечивают программную базу.
62	**Profile** **(**профилирование**)**	Процесс измерения характеристик выполнения программы, обычно выполняемый для повышения производительности за счет обнаружения источника потери производительности.
63	**Property** **(**свойство**)**	На фундаментальном уровне методы **set** и **get** часто позволяют только получить доступ к содержанию поля объекта. Использование свойств позволяет в некоторых языках предложить синтаксическую конструкцию, которая предназначена для ссылки на значение поля (или присвоения значения полю), но фактически приводит к вызову метода. Использование индексированных свойств позволяет применять синтаксические конструкции, которые выглядят как обращение к индексированному массиву с последующим вызовом методов с аргументами. Но в этом массиве, в отличие от традиционных массивов, индексы могут иметь любой тип (а не только целочисленный).
64	**Reference Type** (**см. пп. 52, 83**) **(**ссылочный тип**)**	Один из двух основных типов системы типов. Он является комбинацией адреса в памяти и последовательности битов, расположенных по этому адресу.
65	**Reflection** **(**отражение**)**	Способность предоставлять информацию о типе (**см. п. 75**) во время выполнения, обычно для того, чтобы клиент мог обнаружить члены (**см. п. 45**) этого типа (методы, поля, свойства, события и вложенные типы). Отражение также позволяет ссылаться на эти члены во время выполнения и таким образом определять новые типы. Вновь определенные типы можно сохранить на диске, а также создать и разместить в памяти их новые экземпляры.
66	**Runtime**	См. Execution System, **см. п. 24**
67***a***	**Stack** (стек)	Стек – это зона временной памяти (ОЗУ) типа **LIFO** (**L**ast **I**n – **F**irst **O**ut **:** Последний Входит – Первый Выходит). Обычно каждой команде загрузки в стек (PUSH – поместить) позже будет соответствовать команда извлечения из стека (POP – извлечь). Так как стек является памятью типа LIFO, данные должны извлекаться из стека в порядке, обратном загрузке.
67	**Stack Frame** **(**фрейм стека**)**	В архитектуре механизма выполнения это часть стека, связанная с отдельным вызовом метода (**см. п. 47**).
68	**Stack Walk** **(**обход стека**)**	Процесс просмотра стека вызовов с проверкой каждого фрейма. На платформе .NET обход стека может выполняться в разных ситуациях, включая сборку мусора (**см. п. 25**) и проверку безопасности.
69	**Static Field** (**см. п. 30**) (статическое поле или поле класса)	Поле, которое может иметь только одно значение для всех экземпляров типа (**см. п. 75**), в отличие от поля экземпляра (**см. п. 30**), которое может иметь отдельное значение для каждого экземпляра. В некоторых языках программирования оно называется полем класса (class field) или переменной класса (class variable).
70	**Static Method** (**см. п. 31**) (статический метод или метод класса)	Метод, который вызывается по отношению к типу (**см. п. 75**), а не значению (экземпляру). В некоторых языках программирования он называется методом класса (class method). **Статический метод не может получить указатель this (см. п. 73).**
71	**Store Instruction** **(**инструкция сохранения**)**	Инструкция языка IL (**см. п. 14, 27**), которая сохраняет значения из стека (**см. п. 67*a***) выполнения в других местах, например в локальных переменных.
72	**TCB**	См. Trusted Computing Base, **см. п. 74**
73	**this Pointer** (**см. п. 44**) (указатель **this**)	Указатель на текущий экземпляр объекта. Позволяет получить доступ к полям экземпляра (**см. п. 30**), для которого в данный момент вызван метод (**см. п. 31**). Не все методы могут получить указатель this (см. Static Method, **см. п. 70**). Указатель **this** может иметь неопределенное значение (**null**) (**см. п. 31**).

74	**Trusted Computing Base** (доверенная вычислительная база)	Аппаратное и программное обеспечение, которое защищает компьютерную систему за счет принудительного применения политики безопасности. Для систем на основе платформы .NET эта база включает весь код безопасности (например, классы Permission и диспетчер безопасности), а также некоторые части системы выполнения.
75	**Type** (**см. пп. 52, 64, 83**) (тип). Типы в C#: **простые типы, классы, интерфейсы, массивы, делегаты, перечисления, структуры**	Определение, на основании которого создаются значения (экземпляры). В системе типов среды CLR (**см. п. 16**) предусмотрены два фундаментально разных типа: тип по значению (**см. п. 83**) и ссылочный тип (**см. п. 64**). Типы могут возвращаться статическими (**см. п. 70**) и виртуальными (**см. п. 85**) методами, а также методами экземпляра (**см. п. 31**); их могут иметь статические поля (**см. п. 69**) и поля экземпляра (**см. п. 30**), события (**см. п. 20**) и свойства (**см. п. 63**).
76	**Typed Reference** (типизированная ссылка)	Встроенный тип данных, который представляет пару тип/значение. Типизированная ссылка описывает только локальные переменные и параметры.
77	**Type Loader** (загрузчик типов)	Компонент, который загружает реализацию типа (класса) в память, проверяет ее непротиворечивость и подготавливает к выполнению.
78	**Type Safety** (безопасность типов)	Технология, которая гарантирует, что полный доступ к памяти осуществляется таким образом, чтобы не нарушалась система типов. Обеспечение безопасности типов осуществляется за счет того, что каждая ссылка типизируется, причем каждая такая типизированная ссылка всегда ссылается только на тот тип, который совместим по операции присвоения с ее типом. Использование только строго типизированных типов позволяет механизму выполнения подтверждать все операции присвоения и гарантирует, что ссылка и объект, на который она ссылается, строго совместимы. Примером безопасного по отношению к использованию типов доступа является применение только открытых методов для доступа к значениям. А примером доступа, опасного по отношению к использованию типов, является применение указателей для непосредственного изменения полей. Хотя в среде CLR предусмотрены инструменты для гарантированной безопасности применения типов, они могут использоваться по желанию разработчиков.
79	**Type System** (**см. пп. 52, 64, 83**) (система типов)	Система, которая предоставляет разработчикам несколько типов для определения собственных типов. Она состоит их двух фундаментально разных типов: типов значения и ссылочных типов. Примерами встроенных типов являются тип-значение Int32 и ссылочный тип Object. Разработчики могут определять их собственные типы-значения и ссылочные типы. Система типов используется всеми совместимыми языками, инструментами и CLI-интерфейсами.
80	**Unboxing (см. п. 7)** (распаковка)	Преобразование объектного типа (**см. пп. 52**) в тип значения (**см. пп. 83**). Каждый тип значения может быть упакован (т.е. преобразован в объектный тип), но не все объектные типы могут быть преобразованы в тип значения.
81	**Unmanaged Code** (неуправляемый код)	Код, созданный без учета соглашений и требований среды CLR (**см. с. 16**). Неуправляемый код выполняется в среде .NET с минимальным обслуживанием (например, без сборки мусора, с ограниченной отладкой, без декларативной системы безопасности). Неуправляемый код не имеет метаданных (**см. п. 46**), которые описывают его.
82	**Value** (значение)	Экземпляр любого CLR-совместимого типа (**см. с. 16**).
83	**Value Type** (тип-значение)	Тип данных, который полностью описывает значение за счет указания последовательности битов его представления. Во время выполнения информация о типе экземпляра хранится не в самом экземпляре, а в метаданных (**см. п. 46**). Тип значения аналогичен типу int в языках C++, Java и C#. С помощью упаковки (**см. п. 7**) экземпляры типа-значения можно представлять, в виде объектов (**см. п. 52**).
84	**Verification** (верификация)	Процесс проверки IL-кода для его полного гарантированного соответствия специально заданному набору правил, который задан для доказательства безопасности кода с контролем типов. В среде CLR предусмотрены инструменты проверки IL-кода.
85	**Virtual Method** (виртуальный метод)	Метод, реализация которого неизвестна вплоть до времени выполнения и зависит от типа значения (**см. пп. 83**), получаемого методом в момент активизации во время выполнения (**см. п. 60**).
86	**Visibility** (видимость)	Логическая проверка возможности разрешения компилятором имени элемента, используемого в некой точке программы, т.е. возможности найти тот элемент, на который ссылаются в некой точке программы. Видимость – это первое условие, которое должно быть удовлетворено для определения доступности элемента программы.