# 1. Концепция применения программных платформ. Преимущества применения программных платформ при разборе приложений (на примере Microsoft .NET)

Common Language Runtime обеспечивает среду выполнения .NET-приложений. Выполняет обработку исключительных ситуаций, обеспечение безопасности, средства. Все эти функции доступны из любого языка программирования, соответствующего спецификации Common Language Specification. Microsoft предоставляет три языка программирования, способных использовать CLR, – Visual Basic .NET, Visual C# .NET и Visual C++ With Managed Extensions.

Компилируемый компилятором код для CLR называется управляемым кодом (managed code). Управляемый код пользуется преимуществами среды выполнения и помимо собственно кода содержит метаданные, которые создаются в процессе компиляции и содержат информацию о типах, членах и ссылках, используемых в коде. Метаданные используются средой выполнения:

- для обнаружения классов;

- для загрузки классов;

- для генерации кода для конкретной платформы;

- для обеспечения безопасности.

Среда выполнения также следит за временем жизни объектов. Удаление объектов из памяти происходит с помощью процесса, называемого сборкой мусора (garbage collection).

Common Language Runtime также задает общую систему типов, используемую всеми языками программирования. Это означает, например, что все языки программирования будут оперировать целочисленными данными или данными с плавающей точкой единого формата и единой длины, а представления строк тоже будут едиными для всех языков программирования. За счет единой системы типов достигается более простая интеграция компонентов и кода, написанных на разных языках программирования.

Разработка на C# гораздо быстрее, чем на других объектно-ориентированных языках (C++), за счет полной кроссплатформенности и высокого уровня абстракции. Недостатком является скорость выполнения (как безопасного, так и небезопасного кода), отсутствие прямого доступа к памяти (только небезопасный код), а также размер потребляемой памяти и необходимость установленного .NET.

# 2. Основные операторы языка C#

|  |  |
| --- | --- |
| **Выражение** | **Описание** |
| x[.](http://msdn.microsoft.com/ru-ru/library/6zhxzbds(v=vs.90).aspx)y | Доступ к членам |
| f[(x)](http://msdn.microsoft.com/ru-ru/library/0z4503sa(v=vs.90).aspx) | Вызов метода и делегата |
| a[[x]](http://msdn.microsoft.com/ru-ru/library/a3hd7ste(v=vs.90).aspx) | Доступ к массиву и индексатору |
| x[++](http://msdn.microsoft.com/ru-ru/library/36x43w8w(v=vs.90).aspx),++x, x--,--x | Постфиксное/префиксное приращение/уменьшение |
| [new](http://msdn.microsoft.com/ru-ru/library/fa0ab757(v=vs.90).aspx) T(...) | Создание объекта и делегата |
| **new** T(...){...} | Создание объекта с инициализатором. |
| **new** {...} | Анонимный инициализатор объекта. |
| **new** T[...] | Создание массива. |
| [typeof](http://msdn.microsoft.com/ru-ru/library/58918ffs(v=vs.90).aspx)(T) | Получение объекта System.Type для T |
| [checked](http://msdn.microsoft.com/ru-ru/library/74b4xzyw(v=vs.90).aspx)(x) | Вычисление выражения в проверенном контексте |
| [unchecked](http://msdn.microsoft.com/ru-ru/library/a569z7k8(v=vs.90).aspx)(x) | Вычисление выражения в непроверенном контексте |
| [default](http://msdn.microsoft.com/ru-ru/library/25tdedf5(v=vs.90).aspx) (T) | Получение значения по умолчания для типа T |
| [delegate](http://msdn.microsoft.com/ru-ru/library/900fyy8e(v=vs.90).aspx) {} | Анонимная функция (анонимный метод) |
| x [is](http://msdn.microsoft.com/ru-ru/library/scekt9xw(v=vs.90).aspx) T | Возвращает значение true, если x относится к типу T, в противном случае возвращает значение false |
| x [as](http://msdn.microsoft.com/ru-ru/library/cscsdfbt(v=vs.90).aspx) T | Возвращает x типа T или нулевое значение, если x не относится к типу T |
| x [<](http://msdn.microsoft.com/ru-ru/library/z5wecxwa(v=vs.90).aspx) y … | Операторы отношения |
| x op= y | Составные операторы присвоения: [+=](http://msdn.microsoft.com/ru-ru/library/sa7629ew(v=vs.90).aspx), [-=](http://msdn.microsoft.com/ru-ru/library/2y9zhhx1(v=vs.90).aspx), [\*=](http://msdn.microsoft.com/ru-ru/library/s2bkaksf(v=vs.90).aspx), [/=](http://msdn.microsoft.com/ru-ru/library/d31sybc9(v=vs.90).aspx), [%=](http://msdn.microsoft.com/ru-ru/library/ydwa9zh0(v=vs.90).aspx), [&=](http://msdn.microsoft.com/ru-ru/library/e669ax02(v=vs.90).aspx), [|=](http://msdn.microsoft.com/ru-ru/library/h5f1zzaw(v=vs.90).aspx), [!=](http://msdn.microsoft.com/ru-ru/library/3tz250sf(v=vs.90).aspx), [<<=](http://msdn.microsoft.com/ru-ru/library/ayt2kcfb(v=vs.90).aspx), [>>=](http://msdn.microsoft.com/ru-ru/library/23as4533(v=vs.90).aspx) |
| Логические | x [&](http://msdn.microsoft.com/ru-ru/library/sbf85k1c(v=vs.90).aspx) y, x [^](http://msdn.microsoft.com/ru-ru/library/zkacc7k1(v=vs.90).aspx) y, x [|](http://msdn.microsoft.com/ru-ru/library/kxszd0kx(v=vs.90).aspx) y |
| Условные | x [&&](http://msdn.microsoft.com/ru-ru/library/2a723cdk(v=vs.90).aspx) y, x [||](http://msdn.microsoft.com/ru-ru/library/6373h346(v=vs.90).aspx) y |
| X [??](http://msdn.microsoft.com/ru-ru/library/ms173224(v=vs.90).aspx) y | Равно y, если x — нулевое, в противном случае равно x |
| x [?:](http://msdn.microsoft.com/ru-ru/library/ty67wk28(v=vs.90).aspx) y : z | Равно y, если x имеет значение true, z если x имеет значение false |
| (T)x | Явное преобразование |
| Арифметические | \*, +, - , /, % |
| x >> y | Сдвига |

# 3. Строгая и нестрогая типизация. Характеристики типов данных

В С# используется статическая сильная явная типизация.

Статическая типизация означает то, что конкретные типы данных переменных и функций устанавливаются на этапе компиляции. В динамической типизации все типы выясняются по время выполнения.

Сильная/слабая (строгая/нестрогая) типизация подразумевает, может ли язык смешивать в выражении разные типы и выполнять неявные преобразования автоматически, без учета возможной потери точности или неоднозначности преобразования.

Явная/неявная типизация подразумевает, нужно ли явно задавать тип новых переменных, функций, их аргументов.

Каждый тип данных характеризует набор возможных действий с ними, их внутреннее представление, диапазон допустимых значений, размер.

Типы данных могут быть значимыми (простые типы, структуры, перечисления) и ссылочными (классы, массивы, делегаты, интерфейсы).

# 4. Основные типы данных в языке C#. Приведение типов в языке C#

|  |  |  |
| --- | --- | --- |
| Тип | Область значений | Размер |
| sbyte | -128 до 127 | Знаковое 8-бит целое |
| byte | 0 до 255 | Беззнаковое 8-бит целое |
| char | U+0000 до U+ffff | 16-битовый символ Unicode |
| bool | true или false | 1 байт |
| short | -32768 до 32767 | Знаковое 16-бит целое |
| ushort | 0 до 65535 | Беззнаковое 16-бит целое |
| int | -2147483648 до 2147483647 | Знаковое 32-бит целое |
| uint | 0 до 4294967295 | Беззнаковое 32-бит целое |
| long | -9223372036854775808 до 9223372036854775807 | Знаковое 32-бит целое |
| ulong | 0 до 18446744073709551615 | Беззнаковое 32-бит целое |
| float | ±1,5\*10-45 до ±3,4\*1033 | 4 байта, точность — 7 разрядов |
| double | ±5\*10-324 до ±1,7\*10306 | 8 байт, точность —16 разрядов |
| decimal |  | 12 байт, точность — 28 разрядов |

Типом данных также является перечисление, структура, класс. Неявное преобразование возможно от менее мощного к более мощному типу, но не наоборот. Все типы данных явно или неявно унаследованы от **Object**. Любой встроенный тип данных можно использовать как **System.Uint64**, **System.Single**, …

# 5. Роль CLR в архитектуре .NET. Задачи, решаемые .NET

**Common Language Runtime** обеспечивает среду выполнения **.NET**-приложений. Выполняет обработку исключительных ситуаций, обеспечение безопасности, средства. Все эти функции доступны из любого языка программирования, соответствующего спецификации Common Language Specification. Microsoft предоставляет три языка программирования, способных использовать CLR, – Visual Basic .NET, Visual C# .NET и Visual C++ With Managed Extensions.

Компилируемый компилятором код для **CLR** называется управляемым кодом (managed code). Управляемый код пользуется преимуществами среды выполнения и помимо собственно кода содержит метаданные, которые создаются в процессе компиляции и содержат информацию о типах, членах и ссылках, используемых в коде. Метаданные используются средой выполнения:

- для обнаружения классов;

- для загрузки классов;

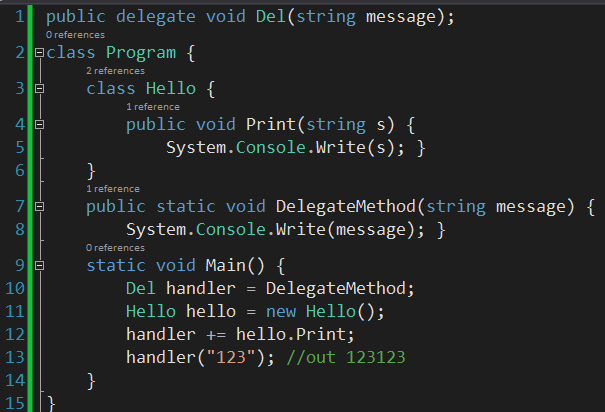
- для генерации кода для конкретной платформы;

- для обеспечения безопасности.

Среда выполнения также следит за временем жизни объектов. Удаление объектов из памяти происходит с помощью процесса, называемого сборкой мусора (**garbage collection**).

**Common Language Runtime** также задает общую систему типов, используемую всеми языками программирования. Это означает, например, что все языки программирования будут оперировать целочисленными данными или данными с плавающей точкой единого формата и единой длины, а представления строк тоже будут едиными для всех языков программирования. За счет единой системы типов достигается более простая интеграция компонентов и кода, написанных на разных языках программирования.

# 6. Делегаты

Делегат — это тип, который определяет подпись метода и его можно связать с любым методом с совместимой подписью. Метод можно запустить (или вызвать) с помощью делегата. Делегаты используются для передачи методов в качестве аргументов к другим методам.

Делегату можно назначить любой метод из любого доступного класса или структуры, соответствующий подписи делегата, которая состоит из типа возвращаемого значения и параметров. Этот метод должен быть статическим методом или методом экземпляра. Это позволяет программно изменять вызовы метода, а также включать новый код в существующие классы. Если подпись делегата известна, то можно назначить собственный метод.

Благодаря возможности ссылаться на метод как на параметр делегаты оптимально подходят для задания функций обратного вызова. Например, алгоритм сортировки можно передать как ссылку на метод, сравнивающий два объекта. Отделение кода сравнения позволяет составить алгоритм в более общем виде.

Можно использовать как замену интерфейсам и как механизм передачи метода как аргумента метода.

Делегаты можно складывать и вычитать. Как только произведено сложение делегата, его базовым классом становится не **System.Delegate**, а **System.MultiCastDelegate**. **System.Delegate** – **sealed** класс, имеет свойство **Method** – возвращает связанный метод, **Target** – экземпляр связанного класса. Если необходимо узнать методы, связанные с делегатом, можно вызвать метод **GetInvocationList().**

Основная функция – инкапсуляция методов.

Отличия и сходства с интерфейсами:

- и делегат, и интерфейс позволяет отделить заголовок от реализации

- делегаты не наследуются

- и на делегат, и на интерфейс можно получить ссылку

- делегаты предпочтительнее использовать там, где нужно инкапсулировать статический метод или там, где необходимо несколько реализаций

- интерфейсы лучше использовать там, где нужна связка методов

|  |  |
| --- | --- |
| Ковариация | Обратный вызов |
| C:\Users\Borsch\Dropbox\Screenshots\Screenshot_63.png | C:\Users\Borsch\Dropbox\Screenshots\Screenshot_65.png |

# 7. Библиотека классов .NET

**System** – является корневым пространством имен для основных типов в **.NET Framework**. Это пространство имен включает классы, представляющие собой базовые типы данных, используемые всеми приложениями: **Object** (корень иерархии наследования), **Byte**, **Char**, **Array**, **Int32**, **String** и т. д. Многие из этих типов соответствуют простым типам данных, которые используются в языке программирования. При написании кода с применением типов **.NET Framework** можно использовать соответствующее ключевое слово языка для базового типа данных **.NET Framework**.

**System.Collections** – Интерфейсы и классы, определяющие различные коллекции объектов: списки, очереди, битовые массивы, хэш-таблицы и словари. **ArrayList**, **BitArray**, **Hashtable**, **Queue**, **SortedList**, **Stack**.

**System.Collections.Generic** – Интерфейсы и классы, определяющие универсальные коллекции, которые позволяют пользователям создавать строго типизированные коллекции, обеспечивающие повышенную безопасность типов и производительность по сравнению с не универсальными строго типизированными коллекциями. **Dictionary**, **HashSet**, **LinkedList**, **List**, **Queue**, **SortedDictionary**, **SortedList**, **Stack.**

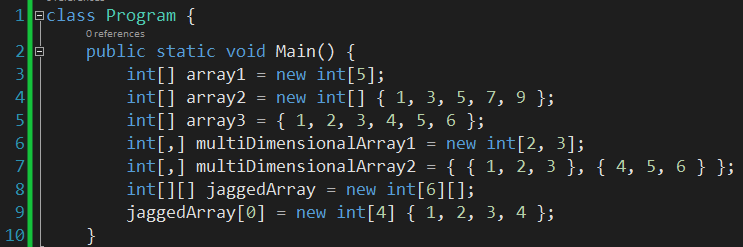
**System.Collections.Specialized** – Специализированные и строго типизированные коллекции, например словарь в виде связного списка, битовый вектор или коллекция, которая может содержать только строки.

**System.Data.Linq** – Классы для доступа к реляционным данным как к объектам. Класс **DataContext** и другие связанные с ним классы могут использоваться для чтения, создания, обновления и удаления объектов, соответствующих элементам базы данных; при этом соответствие задается с помощью атрибутов объектной модели или в отдельном внешнем XML-файле.

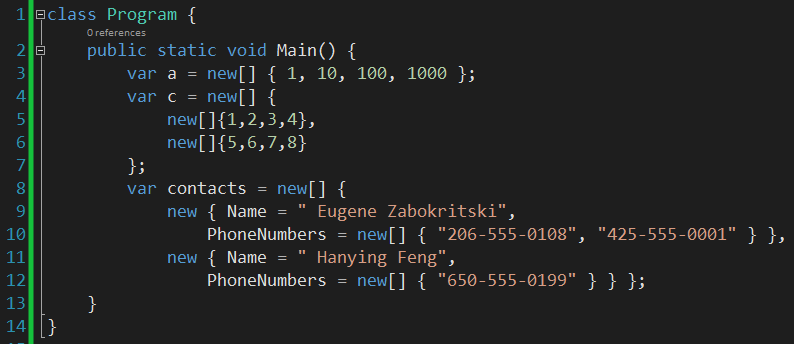
**System.Text** – Классы, предоставляющие кодировки знаков ASCII, Unicode, UTF-7 и UTF-8, абстрактные базовые классы для преобразования блоков знаков в блоки байтов и обратно, а также вспомогательный класс, преобразующий и форматирующий объекты строк (String) без создания промежуточных экземпляров String.

**System.Threading** – Классы и интерфейсы для многопотокового программирования. В дополнение к классам синхронизации работы потоков и доступа к данным (Mutex, Monitor, Interlocked, AutoResetEvent и т. д.), это пространство имен включает класс ThreadPool, который позволяет использовать пул системных потоков, и класс Timer, выполняющий методы обратного вызова для потоков пула.

# 8. Массивы

Массив – ссылочный тип данных, содержащий несколько переменных одного типа. Массивы бывают одномерными, двумерными, массивами массивов. Последние могут реализовывать ступенчатые массивы. Любой массив унаследован от **System.Array.** При передачи массива в качестве аргумента метода, важно помнить, что при передаче по значению в метод передается копия ссылки на массив, следовательно его элементы могут быть изменены, не может быть изменена только ссылка на массив. Для любого массива можно использовать **foreach**. Значение по умолчанию числовых элементов массива задано равным нулю, а элементы ссылок имеют значение **NULL**.

## Нетипизированные массивы

Можно создать неявно типизированный массив, в котором тип экземпляра массива получается из элементов, указанных в инициализаторе массива. Правила для неявно типизированной переменной также применяются к неявно типизированным массивам.

# 9. Пространства имен

Пространства имен представляют собой способ организации различных типов, присутствующих в программах C#. Их можно сравнить с папкой в компьютерной файловой системе. Подобно папкам, пространства имен определяют для классов уникальные полные имена. Программа C# содержит одно или несколько пространств имен, каждое из которых либо определено программистом, либо определено как часть написанной ранее библиотеки классов.

Например, пространство имен **System** содержит класс **Console**, который включает методы для чтения и записи в окне консоли. В пространство имен System входит ряд других пространств, например **System.IO** и **System.Collections**. Только в платформе **.NET Framework** представлено более 80 пространств имен, в каждом из которых находятся тысячи классов, поскольку пространства имен используются для того, чтобы избежать путаницы с типами и методами, имеющими одинаковые имена.

При написании класса вне объявления пространства имен компилятор предоставит ему заданное по умолчанию пространство имен.

Удобно использовать для создания разделяемых классов, методов и типов.

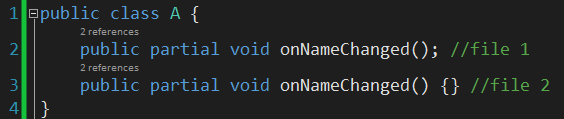
## C:\Users\Borsch\Dropbox\Screenshots\Screenshot_68.pngРазделяемые классы

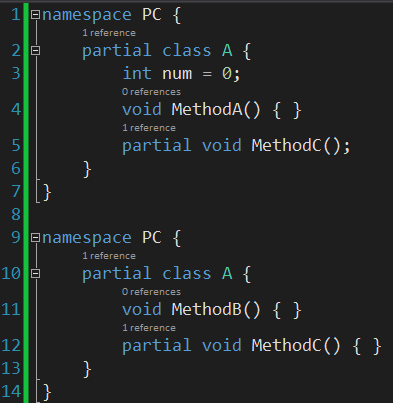
Ключевое слово **partial** указывает на то, что другие части класса, структуры или интерфейса могут быть определены в пространстве имен. Все части должны использовать ключевое слово **partial**. Для формирования окончательного типа все части должны быть доступны во время компиляции. Все части должны иметь одинаковые специальные возможности, например **public**, **private** и т.д.

Если какая-либо из частей объявлена абстрактной, то весь тип будет считаться абстрактным. Если какая-либо из частей объявлена запечатанной, то весь тип будет считаться запечатанным. Если какая-либо из частей объявляет базовый тип, то весь тип будет наследовать данный класс. Можно делать вложенные разделяемые классы.

Разделяемыми могут также быть структуры и интерфейсы.

## Разделяемые методы

Разделяемый класс или структура могут содержать разделяемый метод. Одна часть класса содержит подпись метода. В той же или в другой части можно определить дополнительную реализацию. Если реализация не предоставлена, то метод и все вызовы метода удаляются во время компиляции.

Разделяемые методы позволяют разработчику одной части класса определить метод, схожий с событием. Разработчик другой части класса может решить, реализовывать этот метод или нет. Если метод не реализован, то компилятор удаляет подпись метода и все вызовы этого метода. Таким образом, любой код в разделяемом классе может свободно использовать разделяемый метод, даже если реализация не предоставлена. Во время компиляции и выполнения программы не возникнут никакие ошибки, если метод будет вызван, но не реализован.

Объявления разделяемого метода должны начинаться с контекстно-зависимого ключевого слова **partial**, а метод должен возвращать значение типа **void**.

Разделяемые методы могут иметь параметры **ref**, но не могут иметь параметры **out**.

Разделяемые методы неявно имеют модификатор **private** и поэтому не могут иметь модификатор **virtual**.

## Разделяемые типы

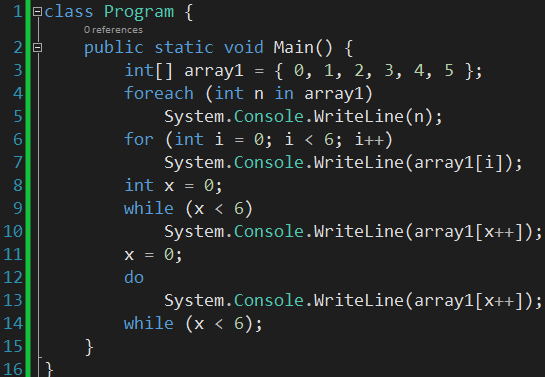
Определения разделяемых типов позволяют разделять определения для классов, структур и интерфейсов на несколько файлов.

# 10. Структура программы. Методы

Метод представляет собой блок кода, содержащий набор инструкций. Программа инициирует выполнение операторов, вызывая метод и задавая необходимые аргументы метода. В C# все инструкции выполняются в контексте метода. Метод **Main** является точкой входа для каждого приложения C#, и вызывается он средой **CLR** при запуске программы.

Методы объявляются в классе или в структуре путем указания уровня доступа, например **public** или **private**, необязательных модификаторов, например **abstract** или **sealed**, возвращаемого значения, имени метода и списка параметров этого метода. Все вместе эти элементы образуют подпись метода.

# 11. Виды циклов



# 12. Объектно-ориентированное программирование. Классы. Объекты

C# является объектно-ориентированным языком программирования и аналогично другим современным языкам группирует связанные поля, методы, свойства и события в структуры данных, которые называются классами.

Класс, по сути, является чертежом для пользовательского типа данных. Определив класс, его можно использовать, загрузив в память. Класс, загруженный в память, называется объектом или экземпляром. Экземпляр класса создается с помощью ключевого слова C# **new**.

Статический член представляет собой метод или поле, доступ к которым можно получить без ссылки на определенный экземпляр класса. Самым общим статическим методом является Main, который представляет точку входа для всех программ C#; следует обратить внимание, что создавать экземпляр содержащего класса для вызова метода **Main** не нужно. Еще одним часто используемым статическим методом является **WriteLine** в классе **Console**. При доступе к статическим методам необходимо обратить внимание на отличие в синтаксисе; с левой стороны оператора **dot** вместо имени экземпляра используется имя класса: **Console.WriteLine**.

Поле класса, объявляемое как статическое, будет общим для всех экземпляров класса.

В статическом классе все элементы также статические. Использование статических классов, методов и полей целесообразно в ряде случаев для повышения производительности и эффективности. Однако если предполагается, что поле является полем экземпляра, хотя на самом деле оно статическое, могут возникнуть неявные ошибки.

# 13. Способы передачи параметров вызова

Параметры методов могут быть переданы по значению, по ссылке или как выходные параметры. Когда параметр передается по значению, метод получает копию данных вызывающего объекта и не может изменить эту копию. При передаче параметра по ссылке метод получает указатель на данные вызывающего объекта. Эти данные используются совместно с вызывающим объектом. Если метод производит изменение значения параметра, передаваемого по ссылке, изменяются данные вызывающего объекта. При использовании параметра, передаваемого по ссылке, метод может использовать начальное значение данных. Выходные параметры похожи на параметры, передаваемые по ссылке, за исключением того что выходные параметры используются только для возврата данных вызывающему объекту, в то время как параметры, передаваемые по ссылке, могут использоваться как для передачи данных методу, так и для получения данных из метода.

Класс или другой ссылочный тип при передаче по значению он передается по копии ссылки. При передаче структуры или другого значимого типа по значению он передается по копии объекта, при этом если полем объекта является ссылочный тип, скопируется только значение ссылки на него.

# 14. Классы. Хранение объектов в стеке и куче

Класс — это логическая структура, позволяющая создавать свои собственные пользовательские типы путем группирования переменных других типов, методов и событий. Класс подобен чертежу. Он определяет данные и поведение типа. Если класс не объявлен статическим, то клиентский код может его использовать, создав объекты или экземпляры, назначенные переменной. Переменная остается в памяти, пока все ссылки на нее не выйдут из области видимости. В это время среда CLR помечает ее пригодной для сборщика мусора. Если класс объявляется статическим, то в памяти остается только одна копия и клиентский код может получить к ней доступ только посредством самого класса, а не переменной экземпляра.

Все значимые объекты и ссылки на ссылочные объекты хранятся в стеке. Ссылочные объекты хранятся в куче. При этом если значимый объект является частью ссылочного, то он тоже хранится в куче. Любой значимый тип можно явно привести к типу **Object.** Данная операция называется упаковкой и позволяет использовать значимый тип как ссылочный, например, для передачи параметра в метод. Обратное действие явного приведения **Object** к значимому типу называется распаковкой. Если в процессе распаковки выяснилось, что данная распаковка некорректна, вызывается **InvalidCastException**. Если значимый тип содержит ссылочный, то в стеке хранится только ссылка на ссылочный, а сам ссылочный хранится в куче.

# 15. Операторы ветвления

В качестве оператора ветвления можно использовать конструкции **if**, **switch**, **goto**. В качестве параметра оператора **switch** можно использовать любую целочисленную константу, а также **string**, **bool**, **char** и перечислимый тип. В отличие от C++, C# не допускает, чтобы выполнение, начавшись в одном разделе, продолжалось в следующем. Приведённый ниже код вызывает ошибку. **Goto** позволяет перейти к любой метке в коде, в том числе в операторе **switch**.

# 16. Исключения

Функции обработки исключений на языке C# помогают обрабатывать любые непредвиденные или исключительные ситуации, происходящие при выполнении программы. При обработке исключений используются ключевые слова **try**, **catch** и **finally** для попыток применения действий, которые могут не достичь успеха, для обработки ошибок, если предполагается, что это может быть разумным, и для последующего освобождения ресурсов. Исключения могут генерироваться средой **CLR**, платформой **.NET** **Framework** или внешними библиотеками, либо кодом приложения. Исключения создаются при помощи ключевого слова **throw**.

Во многих случаях исключение может инициироваться не методом, вызванным непосредственно кодом, а другим методом, расположенным ниже в стеке вызовов. Когда это происходит, среда **CLR** выполняет откат стека в поисках метода с блоком **catch** для определенного типа исключения. При обнаружении первого такого блока **catch** этот блок выполняется. Если среда **CLR** не находит соответствующего блока catch где-либо в стеке вызовов, она завершает процесс и отображает пользователю сообщение.

Исключения имеют типы, в конечном счете являющиеся производными от **System.Exception**.

При возникновении исключения в блоке **try** поток управления немедленно переходит к первому соответствующему обработчику исключений, присутствующему в стеке вызовов. В языке **C#** ключевое слово **catch** используется для определения обработчика исключений.

Если обработчик для определенного исключения не существует, выполнение программы завершается с сообщением об ошибке.

Если в блоке **catch** определяется переменная исключения, ее можно использовать для получения дополнительной информации о типе произошедшего исключения.

Исключения могут явно генерироваться программной с помощью ключевого слова **throw**.

Объекты исключения содержат подробные сведения об ошибке, такие как состояние стека вызовов и текстовое описание ошибки.

Код в блоке **finally** выполняется, даже при возникновении исключения. Блок **finally** используется для освобождения ресурсов, например для закрытия потоков или файлов, открытых в блоке **try**.

# 17. Интерфейсы

Интерфейсы описывают группу связанных функциональных возможностей, которые могут принадлежать к любому классу или структуре. Интерфейсы в могут содержать методы, свойства, события, индексаторы или любое сочетание этих перечисленных типов членов. Интерфейсы не могут содержать поля. Члены интерфейсов автоматически являются открытыми.

Когда говорят, что класс или структура наследует интерфейс, это означает, что класс или структура предоставляет реализацию для всех членов, определяемых интерфейсом. Сам интерфейс не предоставляет функциональных возможностей, которые класс или структура могут наследовать таким же образом, каким могут наследоваться функциональные возможности базового класса. Однако если базовый класс реализует интерфейс, производный класс наследует эту реализацию.

Классы и структуры могут быть унаследованы от интерфейсом таким же образом, как классы могут быть унаследованы от базового класса или структуры, но есть два исключения:

Класс или структура может наследовать несколько интерфейсов.

Когда класс или структура наследует интерфейс, наследуются только имена и подписи методов, поскольку сам интерфейс не содержит реализаций.

Если два интерфейса имеют метод с одной и той же сигнатурой, то можно одним методом реализовать оба интерфейса. Если определить раздельную реализацию, то вызывать тот или иной метод просто по имени невозможно, нужно приводить экземпляр класса к типу интерфейса.

Интерфейс подобен абстрактному базовому классу: любой неабстрактный тип, наследующий интерфейс, должен реализовать все его члены.

Невозможно создать экземпляр интерфейса напрямую.

Интерфейсы могут содержать методы, свойства, индексаторы и события в качестве членов.

Интерфейсы не содержат реализации методов.

Как классы, так и структуры способны наследовать от нескольких интерфейсов.

Интерфейс может быть унаследован от нескольких интерфейсов.

# 18. Виртуальные методы

Ключевое слово **virtual** используется для изменения объявлений методов, свойств, индексаторов и событий и разрешения их переопределения в производном классе.

При вызове виртуального метода тип времени выполнения объекта проверяется на переопределение члена. Вызывается переопределение члена в самом дальнем классе. Это может быть исходный член, если никакой производный класс не выполнял переопределение этого члена.

По умолчанию методы не являются виртуальными. Такой метод нельзя переопределить.

Модификатор **virtual** нельзя использовать с модификаторами **static**, **abstract**, **private** или **override**.

Действие виртуальных свойств аналогично абстрактным методам, за исключением отличий в синтаксисе объявлений и вызовов.

Использование модификатора **virtual** в статическом свойстве является недопустимым.

Виртуальное наследуемое свойство может быть переопределено в производном классе путем включения объявления свойства, которое использует модификатор **override**.

**Virtual** позволяет использовать переопределенные методы классов-наследников при приведении к типу базового класса. Если просто скрыть методы, то будут вызваны методы базового класса.

Совместное использование модификатора **new** и модификатора переопределения в одном члене является недопустимым, поскольку два модификатора имеют взаимоисключающие значения. Модификатор **new** создает новый элемент с таким же именем и приводит к скрытию исходного члена. Модификатор **override** расширяет реализацию для наследуемого члена.

При использовании модификатора **new** в объявлении, которое не скрывает наследуемого члена, возникает предупреждение.

# 19. Переопределение операторов

|  |  |
| --- | --- |
| **Операторы** | **Возможность перегрузки** |
| +, -, !, ~, ++, --, | Эти унарные операторы можно перегрузить. |
| +, -, \*, /, %, &, | , ^, <<, >> | Эти бинарные операторы можно перегрузить. |
| ==, !=, <, >, <=, >= | Операторы сравнения можно перегрузить (но см. примечание после таблицы). |
| &&, || | Условные логические операторы нельзя перегрузить, но они вычисляются с помощью операторов **&** и **|** , допускающих перегрузку. |
| [] | Оператор индексирования массива нельзя перегрузить, но можно определить индексаторы. |
| (T)x | Оператор приведения нельзя перегрузить, но можно определить новые операторы преобразования (explicit и implicit). |
| +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>= | Операторы присвоения нельзя перегрузить, но, например, **+=** вычисляется с помощью **+**, допускающего перегрузку. |
| =, ., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof | Эти операторы нельзя перегрузить. |

Операторы сравнения можно перегружать, но только парами: если перегружен оператор ==, то != также должен быть перегружен. Обратный принцип также действителен и действует для операторов < и >, а также для <= и >=. **Explicit** – явное преобразование. **Implicit** – неявное. (public static explicit operator to(from f)).

# 20. Абстрактные классы

Ключевое слово **abstract** позволяет создавать классы и члены классов, которые являются неполными и должны быть реализованы в производном классе.

Ключевое слово **sealed** позволяет предотвратить наследование класса или определенных членов класса, помеченных ранее как **virtual**.

Создавать экземпляры абстрактного класса нельзя. Назначение абстрактного класса заключается в предоставлении общего определения для базового класса, которое могут совместно использовать несколько производных классов. Например, в библиотеке классов может быть определен абстрактный класс, используемый в качестве параметра для многих из ее функций, поэтому программисты, использующие эту библиотеку, должны задать свою реализацию этого класса, создав производный класс.

Абстрактные классы могут определять абстрактные методы. Для этого перед типом возвращаемого значения метода необходимо поместить ключевое слово **abstract.**

Классы, производные от абстрактного класса, должны реализовывать все абстрактные методы. Если абстрактный класс наследует виртуальный метод из базового класса, абстрактный класс может переопределить виртуальный метод с помощью абстрактного метода.

Если метод **virtual** объявляется **abstract override**, он все равно считается виртуальным по отношению к любому классу, наследующему от абстрактного класса. Класс, наследующий от абстрактного метода, не может получить доступ к исходной реализации. Таким образом абстрактный класс может принуждать производные классы предоставлять новые реализации метода для виртуальных методов.

# 21. Полиморфизм. Наследование

Наследование, вместе с инкапсуляцией и полиморфизмом, является одной из трех основных характеристик (или базовых идей) объектно-ориентированного программирования. Наследование позволяет создавать новые классы, которые повторно используют, расширяют и изменяют поведение, определенное в других классах. Класс, члены которого наследуются, называется базовым классом, а класс, который наследует эти члены, называется производным классом.

При определении класса для наследования от другого класса, производный класс явно получает все члены базового класса, за исключением его конструкторов и деструкторов. Производный класс может таким образом повторно использовать код в базовом классе без необходимости в его повторной реализации. В производном классе можно добавить больше членов. Таким образом, производный класс расширяет функциональность базового класса.

## Абстрактные и виртуальные методы

Когда базовый класс объявляет метод как виртуальный, производный класс может переопределить метод с помощью своей собственной реализации. Если базовый класс объявляет член как абстрактный, то этот метод должен быть переопределен в любом неабстрактном классе, который прямо наследует от этого класса. Если производный класс сам является абстрактным, то он наследует абстрактные члены, не реализуя их. Абстрактные и виртуальные члены являются основой для полиморфизма, который является второй основной характеристикой объектно-ориентированного программирования.

## Абстрактные базовые классы

Можно объявить класс как абстрактный, если необходимо предотвратить прямое создание экземпляров посредством нового ключевого слова. При таком подходе класс можно использовать, только если новый класс является производным от него. Абстрактный класс может содержать один или несколько подписей методов, которые сами объявлены в качестве абстрактных. Эти подписи задают параметры и возвращают значение, но не имеют реализации (тела метода). Абстрактному классу необязательно содержать абстрактные члены; однако, если класс все же содержит абстрактный член, то сам класс должен быть объявлен в качестве абстрактного. Производные классы, которые сами не являются абстрактными, должны предоставить реализацию для любых абстрактных методов из абстрактного базового класса.

## Интерфейсы

Интерфейс является ссылочным типом, в чем-то схожим с абстрактным базовым классом, который состоит только из абстрактных членов. Когда класс наследует от интерфейса, он должен предоставить реализацию для всех членов интерфейса. В классе может быть реализовано несколько интерфейсов, хотя производным он может быть только от одного прямого базового класса.

## Доступ производного класса к членам базового класса

Из производного класса можно получить доступ к открытым, защищенным, внутренним и защищенным внутренним членам базового класса. Хотя производный класс и наследует закрытые члены базового класса, он не может получить доступ к этим членам. Однако все эти закрытые члены все же присутствуют в производном классе и могут выполнять ту же работу, что и в самом базовом классе. Например, предположим, что защищенный метод базового класса имеет доступ к закрытому полю. Это поле должно присутствовать в производном классе для надлежащей работы унаследованного метода базового класса.

## Предотвращение дальнейшего наследования

Класс может предотвратить наследование от него других классов или наследование от любых его членов, объявив себя или члены запечатанными.

## Скрытие производного класса членов базового класса

Производный класс может скрывать члены базового класса путем объявления членов с тем же именем и подписью. Модификатор new может использоваться, чтобы явно указать, что член не предназначен, чтобы быть переопределением базового члена. Использование new не является обязательным, но при отсутствии использования new будет сгенерировано предупреждение компилятора.

## Полиморфизм

О полиморфизме часто говорят, как о третьем базовом элементе объектно-ориентированного программирования, после инкапсуляции и наследования.

1. Во время выполнения объекты производного класса могут рассматриваться как объекты базового класса в таких местах как параметры метода и коллекции массивов. При этом объявленный тип объекта больше не идентичен его типу времени выполнения.
2. Базовые классы могут определять и реализовывать виртуальные методы, а производные классы могут переопределять их. Это означает, что они предоставляют свои собственные определение и реализацию. Во время выполнения, когда клиентский код вызывает метод, среда CLR ищет тип времени выполнения объекта и вызывает это переопределение виртуального метода. Таким образом, в исходном коде можно вызвать метод в базовом классе и вызвать выполнение метода с версией производного класса.
3. Создание иерархии классов, в которой класс каждой конкретной формы производится от общего базового класса.
4. Использование виртуального метода для вызова соответствующего метода в каком-либо производном классе одним вызовом метода базового класса.

# 22. Инкапсуляция

Инкапсуляция — это возможность спрятать от конечного пользователя внутреннее устройство объекта и предоставить доступ только к тем методам, которые необходимы.

Объект — это механизм, который должен работать автономно и иметь инструменты, с помощью которых им можно управлять. Например, если представить автомобиль как объект, то водителю не нужно иметь доступ к двигателю или коробке передач, чтобы управлять автомобилем. Если будет прямой доступ, то водитель может нарушить работу. Чтобы работа не была нарушена, водителю предоставля­ются специальные методы:

- педаль газа для управления оборотами двигателя;

- ручка переключения передач для управления коробкой передач;

- руль для управления колесами и т. д.

Но есть компоненты, к которым водитель должен иметь доступ. Например, чтобы открыть дверь или багажник, не нужно выдумывать дополнительные механизмы, потому что тут нарушить работу автомобиля достаточно сложно.

Если ваши объекты будут автономными и смогут работать без дополнительных данных, то вы получаете выигрыш не только в удобстве программирования текущего проекта, но и в будущем. Допустим, что вы создали объект в C#, который рисует на экране определенную фигуру в зависимости от заданных параметров. Если вам понадобятся подобные возможности в другом проекте, то достаточно воспользоваться уже существующим кодом.

# 23. Строки

Тип данных **string** — это последовательность, содержащая ни одного или любое число знаков Юникода. В платформе.NET Framework **string** является псевдонимом для **String**.

Несмотря на то, что тип **string** является ссылочным типом, операторы равенства (== и !=) определены для сравнения значений объектов типа string, а не ссылок. Это упрощает проверку равенства строк.

Оператор + служит для объединения строк.

Строки являются неизменяемыми: содержимое строкового объекта невозможно изменить после создания объекта.

Оператор [] служит для доступа только для чтения к отдельным знакам объекта **string**.

Строковые литералы имеют тип **string** и могут быть написаны в двух формах: в кавычках и в кавычках с @.

Все встроенные типы данных C# предоставляют метод **ToString**, преобразующий значение в строку. Этот метод может быть использован для преобразования числовых значений в строки.

Чтобы изменить регистр букв в строке (сделать их заглавными или строчными) следует использовать **ToUpper()** или **ToLower()**.

Чтобы найти строку внутри другой строки следует использовать **IndexOf().**

Класс **StringBuilder** создает строковый буфер, который позволяет повысить производительность, если в программе обрабатывается много строк. Класс **StringBuilder** также позволяет заново присваивать отдельные знаки, что не поддерживается встроенным строковым типом данных.

# 24. Структуры

Структуры используют большую часть того же синтаксиса, что и классы, однако они более ограничены по сравнению с ними.

- В объявлении структуры поля не могут быть инициализированы до тех пор, пока они будут объявлены как постоянные или статические.

- Структура не может объявлять используемый по умолчанию конструктор (конструктор без параметров) или деструктор.

- Структуры копируются при присваивании. При присваивании структуры к новой переменной выполняется копирование всех данных, а любое изменение новой копии не влияет на данные в исходной копии.

- Структуры являются типами значений, а классы — ссылочными типами.

- В отличие от классов, структуры можно создавать без использования оператора new.

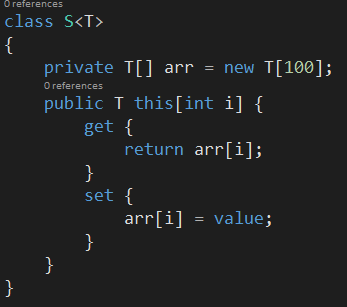
- Структуры могут объявлять конструкторы, имеющие параметры.

- Структура не может быть унаследованной от другой структуры или класса и не может быть основой для других классов. Все структуры наследуют непосредственно от **System.ValueType**, который наследует от **System.Object**.

- Структуры могут реализовывать интерфейсы.

- Структура может использоваться как тип, допускающий значение **NULL**, и ей можно назначить значение **NULL**.

# 25. Индексаторы

Индексаторы позволяют индексировать экземпляры класса или структуры так же, как массивы. Индексаторы напоминают свойства, но их методы доступа принимают параметры.

Индексаторы не обязательно должны использовать в качестве индекса целочисленное значение; конкретный механизм поиска определяет разработчик.

Индексаторы можно перегружать.

Индексаторы могут иметь более одного формального параметра, например при доступе к двухмерному массиву.

# 26. Классы коллекций

См. билет №7.

# 27. Операторы циклов

См. билет №11.

# 28. Свойства

Свойство — это член, предоставляющий гибкий механизм для чтения, записи или вычисления значения частного (**private**) поля. Свойства можно использовать, как если бы они являлись открытыми членами данных, хотя в действительности они являются специальными методами, называемыми методами доступа. Это обеспечивает простой доступ к данным и позволяет повысить уровень безопасности и гибкости методов.

# 29. Конструкторы. Деструкторы. Сборщик мусора

Классы: конструктор нестатичного класса вызывается каждый раз, когда создается экземпляр класса. Конструктор статичного класса вызывается один раз при создании первого экземпляра класса, либо при первом обращении к статичному методу или полю. При определении конструктора с параметрами конструктор по умолчанию становится более недоступен.

Структуры: Конструктор по умолчанию всегда доступен, но его нельзя определить самостоятельно.

Статический конструктор может быть определен как в статичном, так и не в статичном классе. Не имеет параметров и модификаторов доступа. Его нельзя вызвать напрямую.

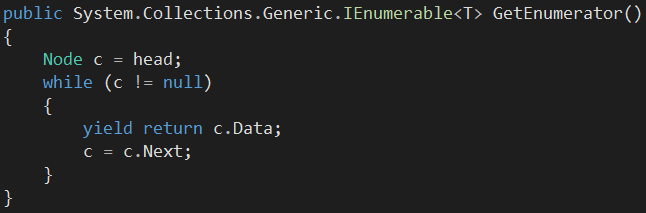
Конструктор копирования реализуется вручную.

Деструкторы: невозможно определить для структур. Не наследуются, не перегружаются, не имеют параметров и модификаторов доступа. При вызове конструктора в блоке **try** вызывается метод **Finalize**, в блоке **finally** находится **Finalize** для его прямого родителя.

Сборщик в автоматическом режиме высвобождает память неиспользуемых объектов. Можно запускать вручную **GC.Collect**. Если необходимо реализовать свой сборщик мусора, можно реализовать интерфейс **IDisposible**, а точнее, метод **Dispose**. Можно использовать метод **GC.SuppressFinalize(this)**, который сообщает системе, что она не должна вызывать метод завершения для указанного объекта.

# Фишечки от Повышева

## Создание экземпляра интерфейса для работы foreach



## Object

Является базовым типом для всех типов данных. Имеет методы **Equals** – можно переопределить для пользовательского класса, **Finalize** – можно переопределить для пользовательского класса, **GetHashCode** – можно переопределить для пользовательского класса, **GetType**, **MemberClone**, **ReferenceEquals**, **ToSring** – можно переопределить для пользовательского класса.

## Возвращаемые значения

- Если тип является вложенным, то, будучи возвращенным, должен быть доступен.

- Закрытые операторы пользовательских типов недопустимы.

- Конструкторы могут быть как открытыми, так и закрытыми.

- Деструкторы не могут содержать модификаторы доступа.

- Перечисления всегда имеют открытый модификатор доступа.

- Базовый класс должен иметь уровень доступа не ниже уровня производного.

- Делегаты должны принимать и возвращать методы одного уровня доступа.

- Тип свойства должен иметь тот же уровень доступа, что и само свойство.