Оглавление

[CLR 3](#_Toc158817875)

[Общее 3](#_Toc158817876)

[Сборка 3](#_Toc158817877)

[Создание сборки 5](#_Toc158817878)

[Глобальный кэш сборок (GAC) 6](#_Toc158817879)

[LOH (Large object heap) // TODO 6](#_Toc158817880)

[CTS, CLS 6](#_Toc158817881)

[Этапы выполнения программы 7](#_Toc158817882)

[Типы 7](#_Toc158817883)

[GC //TODO 8](#_Toc158817884)

[Интерфейсы, абстрактные классы // TODO 9](#_Toc158817885)

[Class vs Structure 9](#_Toc158817886)

[Boxing / Unboxing 10](#_Toc158817887)

[Делегаты 12](#_Toc158817888)

[Action 13](#_Toc158817889)

[Predicate 13](#_Toc158817890)

[Func 13](#_Toc158817891)

[Ковариантность 14](#_Toc158817892)

[Контрвариантность 14](#_Toc158817893)

[Событие (Event) 14](#_Toc158817894)

[Обработка событий //TODO 15](#_Toc158817895)

[Коллекции 15](#_Toc158817896)

[IEnumerable и Yeld return 17](#_Toc158817897)

[Yeld return //TODO 18](#_Toc158817898)

[Сравнение объектов. IEquatable 18](#_Toc158817899)

[IDisposable 18](#_Toc158817900)

[Ref, out, in 19](#_Toc158817901)

[Ковариантность, контрвариантность, инвариантность 20](#_Toc158817902)

[Ковариантные интерфейсы 20](#_Toc158817903)

[Контравариантные интерфейсы 20](#_Toc158817904)

[Совмещение ко/контрвариантности 21](#_Toc158817905)

[Неизменяемые данные (Immutable types) 21](#_Toc158817906)

[Как добиться Immutability? 22](#_Toc158817907)

[Что еще почитать? 22](#_Toc158817908)

[.NET CORE 23](#_Toc158817909)

[.Net standard 23](#_Toc158817910)

[.NET CLI 23](#_Toc158817911)

[Framework-dependent, framework-dependent executable, self-contained 24](#_Toc158817912)

CLR

Общее

.NET Framework предоставляет Common Language Runtime, которая выполняет код и предлагает службы, облегчающие процесс разработки.

Код, который обращается к CLR, называют управляемым кодом.

Фишки:

* объединение языков программирования
* поддержка отслеживания версий и развертывания
* службы отладки и профилирования.

Среда выполнения обеспечивает автоматическое размещение объектов и управление ссылками на них, а также освобождение объектов, когда они больше не используются. Объекты, время жизни которых управляется подобным образом, называются \_\_управляемыми данными\_\_.

неуправляемый код не выполняется в среде CLR. Следовательно, до появления среды .NET Framework во всех программах для Windows применялся неуправляемый код

Объекты, написанные на разных языках, могут взаимодействовать друг с другом, а их поведение может быть тесно интегрировано. Например, разработчик может определить класс, а затем на другом языке создать производный от него класс или вызвать метод из исходного класса. Можно также передать экземпляр класса в метод класса, написанного на другом языке.

Когда компилируется C# код в \*.exe в студии, получается файл с инструкциями для виртуальной машины, которой является CLR. Код с инструкциями отдается на растерзание JIT-компилятору. Он его компилирует и выполняет уже на понятном процессору языке. Этим достигается кросплатформенность и намного уменьшенные исполнительные файлы.

Сборка

* коллекция типов и ресурсов, собранных для совместной работы
* образуют логическую функциональную единицу.
* создаются в виде (EXE) или файла библиотеки динамической компоновки (DLL) и являются стандартными блоками приложений .NET. Они предоставляют сведения для среды CLR, которые нужны для распознавания реализаций типов.

Сборки создаются из управляемых модулей:

1. Код -> IL-код -> управляемый модуль (УМ)
2. УМ - переносимый исполняемый файл

Части УМ:

* заголовок (PE32/PE32+)
* Заголовок CLR - инфа для работы CLR
* IL-код
* Метаданные (что хранится, как интерпритировать, связи с другими сборками)

Модули группируются в сборки.

Сборки имеют следующие свойства.

* Сборки реализованы как файлы EXE или DLL (есть еще файлы .netmodule, но они используются только в процессе разработке, и в конце входят в состав EXE или DLL.
* Сборки загружаются в память только в том случае, если они реально используются.

ДЛЯ CLR

Сборки предоставляют сведения, которые нужны для распознавания реализаций типов.

Для среды выполнения тип не существует вне контекста сборки.

Сборка определяет следующие сведения:

* Код, выполняемый средой CLR.

каждая сборка может иметь только одну точку входа: DllMain, WinMain или Main.

* Манифест

Указывает типы и ресурсы, предоставляемые за пределами сборки, а также перечисляет другие сборки, от которых она зависит. При отсутствии связанного манифеста сборки IL-код выполняться не будет.

Each assembly's manifest performs the following functions:

* Enumerates the files that make up the assembly.
* Governs how references to the assembly's types and resources map to the files that contain their declarations and implementations.
* Enumerates other assemblies on which the assembly depends.
* Provides a level of indirection between consumers of the assembly and the assembly's implementation details.
* Renders the assembly self-describing.

Information contained in the assembly manifest:

**Assembly name**A text string specifying the assembly's name.

**Version number**A major and minor version number, and a revision and build number. The common language runtime uses these numbers to enforce version policy.

**Culture**Information on the culture or language the assembly supports. This information should be used only to designate an assembly as a satellite assembly containing culture- or language-specific information. (An assembly with culture information is automatically assumed to be a satellite assembly.)

**Strong name information**The public key from the publisher if the assembly has been given a strong name.

**List of all files in the assembly**A hash of each file contained in the assembly and a file name. Note that all files that make up the assembly must be in the same directory as the file containing the assembly manifest.

**Type reference information**Information used by the runtime to map a type reference to the file that contains its declaration and implementation. This is used for types that are exported from the assembly.

**Information on referenced assemblies**  
A list of other assemblies that are statically referenced by the assembly. Each reference includes the dependent assembly's name, assembly metadata (version, culture, operating system, and so on), and public key, if the assembly is strong named.

* Граница версий.

Сборка является наименьшей единицей с поддержкой версий в среде CLR.

Версия для всех типов и ресурсов в одной сборке назначается как единому целому.

* Единица развертывания.

При запуске приложения могут присутствовать лишь сборки, первоначально вызываемые приложением.

Другие сборки, например, содержащие ресурсы локализации или вспомогательные классы, могут извлекаться по требованию.

* Ресурсы

Ресурс представляет собой любые неисполняемые данные, которые логически развертываются вместе с приложением. Ресурсы могут отображаться в приложении в виде сообщений об ошибках либо как часть интерфейса пользователя. Ресурсы могут содержать данные различных видов, включая символьные строки, изображения и объекты.

(Для записи сохраняемых объектов в файл ресурсов объекты должны быть сериализуемыми.)

Благодаря хранению данных в файле ресурсов сами данные можно изменять без перекомпиляции всего приложения. Это также позволяет хранить данные в одном месте и исключает необходимость в использовании жестко закодированных данных, которые хранятся в нескольких местах.

Создание сборки

Сборки могут быть статическими или динамическими.

Статическая - такой двоичный модуль .NET, который загружается прямо из хранилища на диске, т.е. на момент запроса CLR-средой он находится в физическом файле (или файлах, если сборка многофайловая) где-то на жестком диске. Как не трудно догадаться, при каждой компиляции исходного кода C# в результате всегда получается статическая сборка.

Динамическая - создается в памяти "на лету" за счет использования типов из пространства имен System.Reflection.Emit. System.Reflection.Emit позволяет сделать так, чтобы сборка с ее модулями, определениями типов и логикой реализации на CIL создавалась во время выполнения. После этого сборку в памяти можно сохранять в дисковый файл, превращая ее в статическую.

Глобальный кэш сборок (GAC)

кэш кода в среде CLR на уровне компьютера.

В GAC сохраняются сборки, предназначенные для совместного использования на компьютере несколькими приложениями.

Прежде чем включить сборки в глобальный кэш сборок, нужно назначить им строгие имена.

Организовывать совместное использование сборок путем установки их в глобальный кэш следует только при необходимости.

Как правило, зависимости между сборками следует сохранять закрытыми, а сами сборки нужно размещать в папке приложения, если они не предназначены для совместного использования.

LOH (Large object heap) // TODO

CTS, CLS

CLS - command laanguage specification- спецификация для компилятора. В CLS определены 2 типа: поле и метод. Через них реализуются все остальные.

Система общих типов определяет способ объявления, использования и управления типами в среде CLR а также является важной составной частью поддержки межъязыковой интеграции в среде выполнения. Система общих типов выполняет следующие функции:

CTS поддерживает пять категорий типов:

1. Классы
2. Структуры
3. Перечисления
4. Интерфейсы
5. Делегаты

Любой тип содержит 0 или более элементов.

Основные элементы:

* Поле — переменная, часть состояния объекта. Идентифицируются по имени и типу.
* Метод — функция, выполняющая действие над объектом. Имеет имя, сигнатуру(число параметров, последовательность, типы параметров, возвр. значение функции) и модификаторы.
* Свойство — в реализации выглядит как метод (get/set) а для вызывающей стороны как поле ( = ). Свойства позволяют типу, в котором они реализованы, проверить входные параметры и состояние объекта.
* Событие — обеспечивает механизм взаимного уведомления объектов.

Модификаторы доступа:

* Public — член доступен любому коду из любой сборки
* Private — член доступен только внутри типа
* Family (protected) — член доступен производныи типам независимо от сборки
* Assembly (internal) — метод вызывается любым кодом из той же сборки
* Family or Assembly (protected internal) — метод вызывается производными типами из любой сборки и + любыми типами из той же сборки.

Этапы выполнения программы

1. Старт процесса CLR
2. Старт главного потока
3. Выделение стека для потока
4. Заполнение стека

Return-address метода - указатель на ячейку, на которую надо переместить указатель стека после выполнения метода (или записать знчение, если есть return).

Типы

Все унаследовано от Object

Примитивы + Enum Унаследованы от ValueType. Все остальное - от Object.

ValueType- по сути структура => от нее нельзя наследоваться.

Любой тип хранит:

* Указатель на объект-тип (только у ссылочных типов)
* Блок синхронизации - для многопоточности (толлько у ссылочных типов)
* Заничение(я)

Static и Virtual поля и методы хранятся в объекте-типе.

При вызове new происходит:

1. выделение памяти
2. инициализация полей значениями 0 и null
3. присвоение значений из конструктора
4. Размещение в памяти

Поля класса размещаются по усмотрению CLR.

Structure - в порядке, написанном программистом.

Для управления размещением в памяти используется

[StructLayout(LayoutKind.<Auto>)] - в ссылочных типах

Explicit - вручную

Sequental - в значимых типах

При Explicit 2 поля могут ссылаться на одну область памяти, но тогда другая область не используется. (просто чтоб знать)

GC //TODO

Объекты делятся на 3 поколения:

К поколению 0 относятся новые объекты, которые еще ни разу не подвергались сборке мусора.

К поколению 1 относятся объекты, которые пережили одну сборку,

к поколению 2 - объекты, прошедшие более одной сборки мусора.

Cборщик мусора анализирует объекты из поколению 0.

Те объекты, которые остаются актуальными после очистки, повышаются до поколения 1.

Если после обработки объектов поколения 0 все еще необходима дополнительная память, то сборщик мусора приступает к объектам из поколения 1. Те объекты, на которые уже нет ссылок, уничтожаются, а те, которые по-прежнему актуальны, повышаются до поколения 2.

Поскольку объекты из поколения 0 являются более молодыми и нередко находятся в адресном пространстве памяти рядом друг с другом, то их удаление проходит с наименьшими издержками.

GC сам вызывает метод Finalize

public class SomeClass: IDisposable

{

private bool disposed = false;

// реализация интерфейса IDisposable.

public void Dispose()

{

Dispose(true);

// подавляем финализацию

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)

{

if (!disposed)

{

if (disposing)

{

// Освобождаем управляемые ресурсы

}

// освобождаем неуправляемые объекты

disposed = true;

}

}

// Деструктор

~SomeClass()

{

Dispose (false);

}

}

//TODO

finalization qeue

воскрешение объектов

Интерфейсы, абстрактные классы // TODO

Абстрактный класс - создает функционал, который могут реализовывать или переопределять наследники.

Интерфейс - определяет функционал, но не реализует его. (Архитектор может определить методы (сами интерфейсы), а реализуют разные разработчики)

Можно наследовать только 1 класс, но много интерфейсов.

1. В abstract может быть реализация методово, свойств, индексаторов. В interface только сигнатура метода.

2. Abstract может содержать поля

3. Abstract может наследоваться от abstract и interface. Interface - только от Interface

Абстрактные свойства, методы, индексаторы могут быть только в абстрактном классе

Interface vs Abctract, Implicit implementation

<https://www.infoworld.com/article/2928719/when-to-use-an-abstract-class-vs-interface-in-csharp.html>

Таблица виртуальных методов

специальная структура в памяти, генерируемая автоматически, в которой перечислены указатели на виртуальные методы. В случае, если где-то в коде вызывается метод function() применительно к указателю на класс A, вместо вызова непосредственно функции A::function() будет произведён вызов функции, находящейся в таблице виртуальных методов по нужному смещению – это поведение реализует полиморфизм.

Когда тип загружается в CLR, для него создается и инициализируется таблица

методов (см. главу 1). Она содержит по одной записи для каждого нового, представляемого только этим типом метода, а также записи для всех виртуальных методов,

унаследованных типом. Унаследованные виртуальные методы включают методы,

определенные в базовых типах иерархии наследования, а также все методы, определенные интерфейсными типами

Для того, чтобы компилятор не генерировал лишний код, ему можно дать указание в виде \_\_declspec(novtable)

class \_\_declspec(novtable) A { .... } ------------- для сокращения затрат памяти

если вызываемый метод невиртуальный, то он ищется в таблице методов compile-time типа, то есть среди методов того типа, которым является ссылка, через которую мы вызываем метод.

Если же вызывается виртуальный метод, то конкретная его реализация ищется в таблице методов run-time типа, то есть того типа объекта, на который реально указывается ссылка:

для вызова метода в языке CIL среды CLR, есть три инструкции: call, calli и callvirt. Последняя инструкция реализует принцип позднего связывания. То есть, говоря простым языком, решение о вызове определённого метода принимается во время выполнения, а не компиляции. Тем самым реализуется идея полиморфизма.

Если класс-наследник имеет модификатор sealed, то для него таблица виртуальных методов не строится, поэтому работает быстрее.

//TODO

Частично реализуемые интерфейсы

Конструкторы абстрактных классов

Class vs Structure

Класс — это ссылочный тип, который может быть прямым производным от другого класса и является неявным производным от типа System.Object. В классе определяются операции, которые объект (являющийся экземпляром класса) может выполнять (методы, события или свойства), и данные, которые объект содержит (поля). Обычно класс включает и определение, и реализацию (в отличие, например, от интерфейса, который содержит только определение без реализации), однако один или несколько членов класса могут не иметь реализации.

Структура — это тип значения, неявно производный от типа System.ValueType, который, в свою очередь, является производным от типа System.Object. Структуры полезны для представления значений с небольшими требованиями к памяти и для передачи параметров по значению в методы со строгой типизацией параметров. В .NET все типы-примитивы (Boolean, Byte, Char, DateTime, Decimal, Double, Int16, Int32, Int64, SByte, Single, UInt16, UInt32 и UInt64) определяются как структуры.

**Первое правило Структуры**:

Всегда все переменные должны быть инициализированы! Нельзя использовать конструктор по умолчанию (без параметров) для структуры, потому что компилятор всегда генерирует его сам. Сгенерированный конструктор для структуры всегда устанавливает поля в 0, false или null – как и для классов

**Второе правило Структуры:**

нельзя инициализировать переменные в месте их объявления!

В общем случае классы используются для моделирования более сложного поведения или для таких данных, которые будут изменяться после создания объекта класса. Структуры лучше подходят для небольших структур данных, информация в которых не должна изменяться после создания структуры.

Только классы:

Поддерживают наследование

Могут быть null (т.к. ссылочные)

Новый экземпляр = новая память

Только структуры:

Передаются по значению

Не могут быть null, пока не используется Nullable

Не выделяется память под экземпляр, пока не 'boxed'

Оба:

Могут содержать методы и события

Поддерживают интерфейсы

Why structs don't have inheritance?

Inheritance implies the existence of a base class and a derived class. With value types like structs, the concept of inheritance

becomes more complex because it would require dynamic allocation, reference semantics, and additional memory management,

which contradicts the nature of value types.

Boxing / Unboxing

Упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа int) к типу object. При упаковке общеязыковая среда CLR обертывает значение в объект типа System.Object и сохраняет его в управляемой куче (хипе).

Распаковка (unboxing), наоборот, предполагает преобразование объекта типа object к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования.

Кроме того, существует другая проблема - проблема безопасности типов. Так, мы получим ошибку во время выполнения программы, если напишем следующим образом:

Account account2 = new Account { Sum = 4000 };

account2.Id = "4356";

int id2 = (int)account2.Id; // Исключение InvalidCastException

Мы можем не знать, какой именно объект представляет Id, и при попытке получить число в данном случае мы столкнемся с исключением InvalidCastException.

class Account<T>

{

public T Id { get; set; }

public int Sum { get; set; }

}

Account<int> account1 = new Account<int> { Sum = 5000 };

Account<string> account2 = new Account<string> { Sum = 4000 };

account1.Id = 2; // упаковка не нужна

account2.Id = "4356";

int id1 = account1.Id; // распаковка не нужна

string id2 = account2.Id;

Console.WriteLine(id1);

Console.WriteLine(id2);

Иногда возникает необходимость присвоить переменным универсальных параметров некоторое начальное значение,

в том числе и null. Но напрямую мы его присвоить не можем:

T id = null;

В этом случае нам надо использовать оператор default(T).

Он присваивает ссылочным типам в качестве значения null, а типам значений - значение 0:

You can cast an int to any numeral type, to Nullable<int>, to ValueType, to Object, and to any type where a explicit or implicit conversion has been defined.

What is the use of boxing/unboxing?

Say you have an integer and you want to put it into an ArrayList.

ArrayList.Add expects an object parameter.

You need to put your integer into an object variable.

An object variable is a reference.

So you need to create an object to which you can have a reference.

Creating that object and taking a reference to it is boxing.

When you retrieve the value from the ArrayList, you can only retrieve an object.

If you want to use it as an integer, you need to retrieve the value from the object.

Taking the value from the object and putting it on the stack is unboxing.

Подвержены боксингу и анбоксингу

Делегаты

Делегат – класс, который наследуется…

MSFT:

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Делегат представляет собой объект, который может ссылаться на метод.

delegate возвращаемый\_тип имя (список\_параметров);

Когда компилятор C# обрабатывает тип делегата, он автоматически генерирует запечатанный (sealed) класс, унаследованный от System.MulticastDelegate. Этот класс (в сочетании с его базовым классом System.Delegate) предоставляет необходимую инфраструктуру для делегата, чтобы хранить список методов, подлежащих вызову в более позднее время.

Метооды делегатов:

* Invoke() — возможно, главный из них, поскольку он используется для синхронного вызова каждого из методов, поддерживаемых объектом делегата; это означает, что вызывающий код должен ожидать завершения вызова, прежде чем продолжить свою работу. Может показаться странным, что синхронный метод Invoke() не должен вызываться явно в коде C#. Invoke() вызывается "за кулисами", когда применяется соответствующий синтаксис C#.
* BeginInvoke() и EndInvoke() предлагают возможность вызова текущего метода асинхронным образом, в отдельном потоке выполнения.

public abstract class Delegate : ICloneable, ISerializable

{

// Методы Для взаимодействия со списком функций.

public static Delegate Combine(params Delegate[] delegates);

public static Delegate Combine(Delegate a, Delegate b) ;

public static Delegate Remove(Delegate source, Delegate value);

public static Delegate RemoveAll(Delegate source, Delegate value);

// Перегруженные операции.

public static bool operator ==(Delegate d1, Delegate d2) ;

public static bool operator != ( Delegate d1, Delegate d2) ;

// Свойства, показывающие цель делегата.

public Methodlnfo Method { get; }

public object Target { get; }

}

public abstract class MulticastDelegate : Delegate

{

// Возвращает список методов, на которые "указывает" делегат.

public sealed override Delegate[] GetlnvocationList ();

// Перегруженные операции.

public static bool operator ==(MulticastDelegate d1, MulticastDelegate d2) ;

public static bool operator !=(MulticastDelegate d1, MulticastDelegate d2) ;

// Используются внутренне для управления списком методов, поддерживаемых делегатом.

private IntPtr \_invocationCount;

private object \_invocationList;

}

От этих классов нельзя наследоваться. Будет ошибка компиляции.

|  |  |
| --- | --- |
| **Член** | **Назначение** |
| Method | Это свойство возвращает объект System.Reflection.Method, который представляет детали статического метода, поддерживаемого делегатом |
| Target | Если метод, подлежащий вызову, определен на уровне объекта (т.е. не является статическим), то Target возвращает объект, представляющий метод, поддерживаемый делегатом. Если возвращенное Target значение равно null, значит, подлежащий вызову метод является статическим |
| Combine() | Этот статический метод добавляет метод в список, поддерживаемый делегатом. В C# этот метод вызывается за счет использования перегруженной операции += в качестве сокращенной нотации |
| GetlnvokationList() | Этот метод возвращает массив типов System.Delegate, каждый из которых представляет определенный метод, доступный для вызова |
| Remove() RemoveAll() | Эти статические методы удаляют метод (или все методы) из списка вызовов делегата. В C# метод Remove () может быть вызван неявно, посредством перегруженной операции -= |

В делегат можно добавлять и удалять методы. Несколько методов в делегате выполняются с одним набором входных параметров.

Action

Делегат Action является обобщенным, принимает параметры и возвращает значение void:

public delegate void Action<T>(T obj)

Predicate

Делегат Predicate<T>, как правило, используется для сравнения, сопоставления некоторого объекта T определенному условию. В качестве выходного результата возвращается значение true, если условие соблюдено, и false, если не соблюдено:

Predicate<int> isPositive = delegate (int x) { return x > 0; };

Console.WriteLine(isPositive(20));

Console.WriteLine(isPositive(-20));

Func

Он возвращает результат действия и может принимать параметры. Он может принимать до 16 параметров, последний (17) - выходной.

TResult Func<out TResult>()

TResult Func<in T, out TResult>(T arg)

Func<T1, T2, Tn, ReturnType>

Action<T1...>

Ковариантность позволяет возвращать из метода объект, тип которого является производным от типа, возвращаемого делегатом

Контрвариантность предполагает возможность передавать в метод объект, тип которого является более универсальным по отношению к типу параметра делегата.

Событие (Event)

События основаны на делегатах и предоставляют им механизм публикации/подписки. События являются членами класса и объявляются с помощью ключевого слова event. Чаще всего для этой цели используется следующая форма:

event делегат\_события имя\_события;

События поддерживают групповую адресацию -> несколько объектов могут реагировать на уведомление о событии.

Методы экземпляра и статические методы могут быть использованы в качестве обработчиков событий. Когда статический метод используется в качестве обработчика, уведомление о событии распространяется на весь класс. А когда в качестве обработчика используется метод экземпляра, то события адресуются конкретным экземплярам объектов.

namespace ConsoleApplication1

{

delegate void UI ();

class MyEvent

{

// Объявляем событие

public event UI UserEvent;

// Используем метод для запуска события

public void OnUserEvent()

{

UserEvent();

}

}

class UserInfo

{

string uiName, uiFamily;

int uiAge;

public UserInfo(string Name, string Family, int Age)

{

this.Name = Name;

this.Family = Family;

this.Age = Age;

}

public string Name { set { uiName = value; } get { return uiName; } }

public string Family { set { uiFamily = value; } get { return uiFamily; } }

public int Age { set { uiAge = value; } get { return uiAge; } }

// Обработчик события

public void UserInfoHandler()

{

Console.WriteLine("Событие вызвано!\n");

Console.WriteLine("Имя: {0}\nФамилия: {1}\nВозраст: {2}",Name,Family,Age);

}

}

class Program

{

static void Main()

{

MyEvent evt = new MyEvent();

UserInfo user1 = new UserInfo(Name: "Alex", Family: "Erohin", Age: 26);

// Добавляем обработчик события

evt.UserEvent += user1.UserInfoHandler;

// Запустим событие

evt.OnUserEvent();

Console.ReadLine();

}

}

}

Событие C# в действительности развертывается в два скрытых метода, один из которых имеет префикс add\_, а другой — remove\_. За этим префиксом следует имя события C#. Например, событие UserEvent превращается в два скрытых метода CIL с именами add\_UserEvent() и remove\_UserEvent(). Если заглянуть в CIL-код метода add\_UserInfoHandler(), можно обнаружить там вызов метода Delegate.Combine()

Обработка событий //TODO

<https://professorweb.ru/my/csharp/charp_theory/level10/10_9.php>

Коллекции

Необобщенные коллекции

Необобщенные или универсальные коллекции определены в пространстве имен System.Collection.Generic. Самые популярные из этого пространства имен – в списке ниже:

List – класс коллекции, предоставляющий доступ к элементам по индексу. Содержит методы изменения коллекции, сортировки элементов и поиска по элементам.

Dictionary – класс коллекции, в основе которой лежат пара ключ/значение. Элементы упорядочены по ключу. Доступ к элементам производится через ключ.

Queue – класс коллекции, реализующий модель FIFO (первым пришел – первым ушел).

Stack – класс коллекции, реализующий модель LIFO (последним пришел – первым вышел).

SortedList – класс коллекции, дополняющий функциональность типа List возможностью реализовать свой механизм сортировки путем реализации класса интерфейса IComparer.

Любой из типов, перечисленных выше применяется в случае, если все элементы коллекции одного типа данных.

Обобщенные коллекции

Классы обобщенных коллекций определены в пространстве имен System.Collections библиотеки базовых типов. Самые часто используемые представлены в списке ниже:

Queue - класс коллекции, реализующий модель FIFO (первым пришел – первым ушел)

Stack – класс коллекции, реализующий модель LIFO (последним пришел – первым вышел)

ArrayList – динамический массив, способен увеличивать размер при добавлении нового элемента.

HashTable – класс коллекции, в основе которой лежат пара ключ/значение. Значения упорядочены по хэш-коду.

Queue

Enqueue(T) Adds an item into the queue.

Dequeue Returns an item from the beginning of the queue and removes it from the queue.

Peek(T) Returns an first item from the queue without removing it.

Contains(T) Checks whether an item is in the queue or not

Clear() Removes all the items from the queue.

Stack

Push(T) Inserts an item at the top of the stack.

Peek() Returns the top item from the stack.

Pop() Removes and returns items from the top of the stack.

Contains(T) Checks whether an item exists in the stack or not.

Clear() Removes all items from the stack.

Array vs ArrayList

Array Declaration & Initialization:

int[] arr = new int[5]

int[] arr = new int[5]{1, 2, 3, 4, 5};

int[] arr = {1, 2, 3, 4, 5};

ArrayList Declaration & Initialization:

ArrayList arList = new ArrayList();

arList.Add(1);

arList.Add("Two");

arList.Add(false);

Array stores a fixed number of elements. The size of an Array must be specified at the time of initialization.

ArrayList grows automatically and you don't need to specify the size.

Array is strongly typed. This means that an array can store only specific type of items\elements.

ArrayList can store any type of items\elements.

No need to cast elements of an array while retrieving because it is strongly typed and stores a specific type of items only.

The items of ArrayList need to be cast to an appropriate data type while retrieving. So, boxing and unboxing happens.

Array Performs faster than ArrayList because it is strongly typed.

ArrayListPerforms slows because of boxging and unboxing.

Hashtable vs Dictionary

Hashtable is included in the System.Collections namespace.

Dictionary is included in the System.Collections.Generic namespace.

Hashtable is a loosely typed (non-generic) collection, this means it stores key-value pairs of any data types.

Dictionary is a generic collection. So it can store key-value pairs of specific data types.

Hashtable is thread safe.

Only public static members are thread safe in Dictionary.

Hashtable returns null if we try to find a key which does not exist.

Dictionary throws an exception if we try to find a key which does not exist.

Data retrieval is slower than dictionary because of boxing-unboxing.

Data retrieval is faster than Hashtable.

**использовать пользовательский класс как ключ в Dictionary. Что для этого надо поменять (добавить) в классе? -** override GetHashCode() override Equals()

class Foo

{

public string Name { get; set;}

public int FooID {get; set;}

public override int GetHashCode()

{

return FooID;

}

public override bool Equals(object obj)

{

return Equals(obj as Foo);

}

public bool Equals(Foo obj)

{

return obj != null && obj.FooID == this.FooID;

}

}

IEnumerable и Yeld return

Интерфейс IEnumerable имеет метод, возвращающий ссылку на другой интерфейс - перечислитель:

|  |
| --- |
| public interface IEnumerable  {      IEnumerator GetEnumerator();  } |

А интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере:

|  |
| --- |
| public interface IEnumerator  {      bool MoveNext(); // перемещение на одну позицию вперед в контейнере элементов      object Current {get;}  // текущий элемент в контейнере      void Reset(); // перемещение в начало контейнера  } |

Метод MoveNext() перемещает указатель на текущий элемент на следующую позицию в последовательности. Если последовательность еще не закончилась, то возвращает true. Если же последовательность закончилась, то возвращается false.

Свойство Current возвращает объект в последовательности, на который указывает указатель.

Метод Reset() сбрасывает указатель позиции в начальное положение.

Каким именно образом будет осуществляться перемещение указателя и получение элементов зависит от реализации интерфейса. В различных реализациях логика может быть построена различным образом.

using System.Collections;

string[] people = {"Tom", "Sam", "Bob"};

IEnumerator peopleEnumerator = people.GetEnumerator(); // получаем IEnumerator

while (peopleEnumerator.MoveNext())   // пока не будет возвращено false

{

    string item = (string)peopleEnumerator.Current; // получаем элемент на текущей позиции

    Console.WriteLine(item);

}

peopleEnumerator.Reset(); // сбрасываем указатель в начало массива

Yeld return //TODO

Сравнение объектов. IEquatable

.Equals(object obj) – по умолчанию сравнивает по ссылке (под капотом вызывается ReferenceEquals), но можно перегрузить. После перегрузки == будет все равно сравнивать по ссылке, поэтому вызывать надо явно:

ob1.Equals(ob2)

Подробнее:

<https://bool.dev/blog/detail/equals-i-i-kak-rabotaet-sravnenie-tipov-v-c>

<https://habr.com/ru/post/137680/>

IEquatable<T>

Ответ и вопрос тут:  
<https://ru.stackoverflow.com/questions/841939/Зачем-мы-реализовываем-iequatablet-если-equals-есть-в-object>

и еще тут

<https://professorweb.ru/my/csharp/charp_theory/level11/11_12.php>

Самое главное: IEquatable<t> позволяет переопределить метод Equals для объекта конкретного класса, а не для object. Т.е. выпиливается приведение типа.

IDisposable

Метод финализации применяться для освобождения неуправляемых ресурсов при активизации процесса сборки мусора. Когда действительно реализуется поддержка интерфейса IDisposable, то предполагается, что после завершения работы с объектом метод Dispose() должен вручную вызываться пользователем этого объекта, прежде чем объектной ссылке будет позволено покинуть область действия.

Интерфейс IDisposable может быть реализован как в классах, так и в структурах (в отличие от метода Finalize(), который допускается переопределять только в классах), потому что метод **Dispose() вызывается пользователем объекта** (а не сборщиком мусора).

Конструкция

using (DisposableObject obj = new DisposableObject(4))

{

// Необходимые действия

}

компилятором заменяется на

DisposableObject obj = new DisposableObject(4);

try

{

// Выполнение необходимых операций

}

finally

{

obj.Dispose();

}

Ref, out, in

* ref - передает аргументы по ссылке. Любые изменения, внесенные в этот аргумент в методе, будут отражены в этой переменной, когда управление вернется к вызывающему методу.
* out - параметр должен быть проинициализирован или изменен методом.
* in – параметр не может изменяться методом.

|  |  |
| --- | --- |
| Ref | Out |
| Параметр или аргумент должен быть сначала инициализирован, прежде чем он будет передан в ref. | Инициализация параметра или аргумента перед передачей его в out не является обязательной. |
| Не требуется присваивать или инициализировать значение параметра (который передается по ref) перед возвратом в вызывающий метод. | Вызываемый метод обязан присвоить или инициализировать значение параметра (который передается в out) перед возвратом в вызывающий метод. |
| Передача значения параметра по Ref полезна, когда вызываемый метод также должен модифицировать передаваемый параметр. | Объявление параметра в методе out полезно, когда из функции или метода необходимо вернуть несколько значений. |
| Инициализация значения параметра перед его использованием в вызывающем методе не обязательна. | Значение параметра должно быть инициализировано в вызывающем методе перед его использованием. |
| Когда мы используем REF, данные могут передаваться двунаправленно. | Когда мы используем OUT, данные передаются только однонаправленно (от вызываемого метода к вызывающему методу). |
| И ref, и out по-разному обрабатываются во время выполнения программы, а во время компиляции они обрабатываются одинаково. | |
| Свойства не являются переменными, поэтому они не могут быть переданы в качестве параметра out или ref. | |

**Ключевое слово Ref / Out и перегрузка методов**

И ref, и out обрабатываются по-разному во время выполнения программы, и одинаково во время компиляции, поэтому методы не могут быть перегружены, если один метод принимает аргумент как ref, а другой — как out.

Ковариантность, контрвариантность, инвариантность

**Ковариантность**: позволяет использовать более конкретный тип, чем заданный изначально

**Контравариантность**: позволяет использовать более универсальный тип, чем заданный изначально

**Инвариантность**: позволяет использовать только заданный тип

Ковариантные интерфейсы

Обобщенные интерфейсы могут быть ковариантными, если к универсальному параметру применяется ключевое слово out. Такой параметр должен представлять тип объекта, который возвращается из метода. Например:

class Message

{}

class EmailMessage : Message

{}

interface IMessenger<out T>

{

    T WriteMessage(string text);

}

class EmailMessenger : IMessenger<EmailMessage>

{

    public EmailMessage WriteMessage(string text)

    {

        return new EmailMessage($"Email: {text}");

    }

}

Здесь обобщенный интерфейс IMessenger представляет интерфейс мессенджера и определяет метод WriteMessage() для создания сообщения. При этом на момент определения интерфейса мы не знаем, объект какого типа будет возвращаться в этом методе. Ключевое слово out в определении интерфейса указывает, что данный интерфейс будет ковариантным.

Теперь можно сделать так:

IMessenger<Message> outlook = new EmailMessenger();

Если бы не слово out, возникла бы ошибка. Поскольку в этом случае невозможно было бы привести объект IMessenger<EmailMessage> к типу IMessenger<Message>.

Контравариантные интерфейсы

Для создания контравариантного интерфейса надо использовать ключевое слово in. Например:

interface IMessenger<in T>

{

    void SendMessage(T message);

}

class SimpleMessenger: IMessenger<Message>

{

    public void SendMessage(Message message)

{

Console.WriteLine($"Отправляется сообщение: {message.Text}");

}

}

В итоге можно сделать так:

IMessenger<EmailMessage> outlook = new SimpleMessenger();

и так:

IMessenger<Message> telegram = new SimpleMessenger();

То есть объект интерфейса с более универсальным типом приводится к объекту интерфейса с более конкретным типом.

У контрвариантного интерфейса универсальный параметр контрвариантного типа может применяться только к аргументам метода, но не может применяться к возвращаемому результату метода.

Совмещение ко/контрвариантности

interface IMessenger<in T, out K>

{

    void SendMessage(T message);

    K WriteMessage(string text);

}

class SimpleMessenger : IMessenger<Message, EmailMessage>

{

    public void SendMessage(Message message)

    {

        Console.WriteLine($"Отправляется сообщение: {message.Text}");

    }

    public EmailMessage WriteMessage(string text)

    {

        return new EmailMessage($"Email: {text}");

    }

}

Благодаря ковариантности/контравариантности объект класса SimpleMessenger может представлять типы IMessenger<EmailMessage, Message>, IMessenger<Message, EmailMessage>, IMessenger<Message, Message> и IMessenger<EmailMessage, EmailMessage>.

Неизменяемые данные (Immutable types)

Данные не могут менять свои начальные значения.

Плюсы таких данных:

Проще прослеживать поведение кода, т.к. известно значение (по крайней мере его наличие).

Потокобезопасность – не надо париться за значение, которое никогда не изменится.

Не нужно создавать резервные копии, т.к. данные не изменятся.

Всегда валидное значение – инициализируется при создании и все. Дальше оно всегда есть.

Как добиться Immutability?

* Убрать сеттеры.
* private поля сделать readonly
* В конструкторе все параметры сделать обязательными (без дефолтных значений), чтоб они использовались при инициализации приватных полей.

В итоге данные станут доступны для изменения только из класса.

Начиная с C# 6.0 можно сделать «readonly autoimplemented properties»

{

    public int Red { get; }

    public int Green { get; }

    public int Blue { get; }

    public RGBColor(int red, int green, int blue)

    {

        Red = red;

        Green = green;

        Blue = blue;

    }

}

Что еще почитать?

14 постов про это дело - <https://ericlippert.com/?s=immutability&submit=Search>

.NET CORE

.**Net standard**

**.NET Standard** — это набор базовых API (другое их название — BCL, библиотека базовых классов), которые должны поддерживаться во всех реализациях .NET. .NET Standard позволяет создавать библиотеки, подходящие для любых приложений .NET, вне зависимости от реализации .NET или операционной системы, в которой они выполняются.

| **Платформа** | **ОС** | **Open Source** | **Назначение** |
| --- | --- | --- | --- |
| .NET Framework | Windows | Нет | Создание классических Windows-приложений и веб-приложений ASP.NET для IIS. |
| .NET Core | Windows, Linux, macOS | Да | Создание кроссплатформенных консольных приложений, а также веб-приложений и облачных служб ASP.NET Core. |
| Xamarin | iOS, Android, macOS | Да | Создание мобильных приложений для iOS и Android, классических приложений для macOS. |
| .NET Standard | любая | Да | Создание библиотек, которые можно использовать в любых реализациях .NET, в том числе .NET Framework, .NET Core и Xamarin. |

BCL — это набор базовых API, не зависящих от инфраструктур пользовательского интерфейса и моделей приложений. В него входят простые типы, файловый ввод-вывод, сетевые API, API сериализации, XML и другое.

.NET CLI

кроссплатформенный, входить в состав .NET Core SDK.

Структура команды:

dotnet <command> <argument> <option>

new, run, build, restore

Больше тут: <https://www.tutorialsteacher.com/core/net-core-command-line-interface>

Framework-dependent, framework-dependent executable, self-contained

Публикация приложений

| **Type** | **SDK 2.1** | **SDK 3.1** | **SDK 5.0** | **Command** |
| --- | --- | --- | --- | --- |
| [framework-dependent executable](https://docs.microsoft.com/en-us/dotnet/core/deploying/#publish-framework-dependent) for the current platform. |  | ✔️ | ✔️ | [dotnet publish](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-publish) |
| [framework-dependent executable](https://docs.microsoft.com/en-us/dotnet/core/deploying/#publish-framework-dependent) for a specific platform. |  | ✔️ | ✔️ | [dotnet publish -r <RID> --self-contained false](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-publish) |
| [framework-dependent cross-platform binary](https://docs.microsoft.com/en-us/dotnet/core/deploying/#publish-framework-dependent). | ✔️ | ✔️ | ✔️ | [dotnet publish](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-publish) |
| [self-contained executable](https://docs.microsoft.com/en-us/dotnet/core/deploying/#publish-self-contained). | ✔️ | ✔️ | ✔️ | [dotnet publish -r <RID>](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-publish) |

Создание ехе-шника.

**EXE НЕ КРОССПЛАТФОРМЕННЫЕ САМИ ПО СЕБЕ.** Для них надо указывать целевую платформу.

| **Type** | **SDK 2.1** | **SDK 3.1** | **SDK 5.0** | **Command** |
| --- | --- | --- | --- | --- |
| [framework-dependent executable](https://docs.microsoft.com/en-us/dotnet/core/deploying/#publish-framework-dependent) for the current platform. |  | ✔️ | ✔️ | [dotnet publish](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-publish) |
| [framework-dependent executable](https://docs.microsoft.com/en-us/dotnet/core/deploying/#publish-framework-dependent) for a specific platform. |  | ✔️ | ✔️ | [dotnet publish -r <RID> --self-contained false](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-publish) |
| [self-contained executable](https://docs.microsoft.com/en-us/dotnet/core/deploying/#publish-self-contained). | ✔️ | ✔️ | ✔️ | [dotnet publish -r <RID>](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-publish) |

Более подробно тут: <https://docs.microsoft.com/en-us/dotnet/core/deploying/>