Плывите сосиски (сосите плывиски))))

[**===1. Системы контроля версий. Классификация. Git, основные возможности**](#_7amqgnhnumnw) **5**

[**===2. Состав. NET Framework. Структура среды выполнения CLR.**](#_274wg0q08mu4) **7**

[**===3. Структура управляемого модуля - portable executable (PE). Понятие и исполнение сборки. CIL.**](#_7tvslk3ztz37) **8**

[**===4. CTS (Common Type System). Типы данных C#. Ссылочные и типы значений.**](#_xgzgdd6jsabh) **9**

[**===5. Понятие упаковки и распаковки типов. Типы Nullable: преобразование, проверка, null-объединение**](#_3882iqo8upx) **11**

[**===6. Тип данных String: операции, литералы, пустые и нулевые строки, форматированный вывод.**](#_h6bohmrzr536) **13**

[**===7. Неявная типизация – назначение и использование.**](#_8sgya1fc66n) **16**

[**===8. Массивы C# одномерные, прямоугольные и ступенчатые.**](#_bnh17vquy2nd) **17**

[**===9. Понятие кортежей. Свойства, создание**](#_rweq1rux0lyi) **18**

[**===10. Принципы объектно-ориентированного программирования.**](#_7rzj8rakmwfi) **19**

[**===11. Класс. Элементы класса. Свойства и индексаторы.**](#_otg05e818ij8) **20**

[**===12. Класс. Константы. Поля только для чтения. Инициализаторы класса.**](#_uni0yd1f0j2) **22**

[**===13. Спецификаторы доступа C#. Видимость типов. Доступ к членам типов.**](#_5kxncni0hefs) **22**

[**===14. Класс. Конструкторы и их свойства. Деструкторы**](#_ywnpeel3ri9s) **24**

[**===15. Класс и методы System.Object.**](#_57bzabi2jrq) **25**

[**===16. Статические методы и статические конструкторы класса.**](#_71yb8lmg42hk) **26**

[**===17. Статические классы. Методы расширения и правила их определения.**](#_qv9nofwk0t69) **27**

[**===18. Анонимные типы.**](#_60qkucfr2yl5) **28**

[**===19. Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы.**](#_1u69n72sibuu) **29**

[**===20. Перегрузка методов и операторов. Правила перегрузки операторов.**](#_8miqlfn98c30) **31**

[**===21. Операции преобразования типа. Явная и неявная форма. Ограничения.**](#_9aobmmwrtqvk) **32**

[**===22. Вложенные типы. Вложенные объекты**](#_m0pq2d1s4obw) **34**

[**===23. Правила наследования C#.**](#_frerxjds3t6b) **35**

[**===24. Сокрытие имен при наследовании. Обращение к скрытым членам**](#_laeumnhr3wop) **36**

[**===25. Использование операций is и as**](#_h88s66qicypi) **37**

[**===26. Полиморфизм. Виртуальные методы, свойства и индексаторы. Правила переопределения.**](#_2sa4szet7kui) **38**

[**===27. Понятие раннего и позднего связывания.**](#_kdxxum623v4e) **39**

[**===28. Абстрактные классы и методы. Бесплодные классы.**](#_atk643nh56jv) **40**

[**===29. Структуры в C#.**](#_4b1h4s712ypy) **41**

[**===30. Интерфейсы. Свойства интерфейсов. Реализация интерфейсов.**](#_d5ybnbgh036x) **43**

[**===31. Явная и неявная реализация интерфейсов. Работа с объектами через интерфейсы.**](#_thhomwlifctz) **44**

[**===32. Ковариантность интерфейсов. Контравариантность интерфейсов**](#_j96ls140y8wt) **46**

[**===33. Стандартные интерфейсы .NET. Назначение и применение.**](#_t48awy2ddmxv) **48**

[**===34. Исключительные ситуации. Генерация и повторная генерация исключений.**](#_ewec33w6eul4) **50**

[**===35. Исключительные ситуации. Варианты обработки исключений. Фильтры исключений**](#_ndn4ndy97317) **51**

[**===36. Обобщения (generics). Свойства обобщений.**](#_gfngvgxe1tr1) **53**

[**===37. Концепция ограничений обобщений. Статические члены обобщений.**](#_w7qbmyvasi5y) **55**

[**===38. Делегаты. Определение, назначение и варианты использования. Обобщенные делегаты.**](#_7zkw8kq8i699) **58**

[**===39. Анонимные функции. Лямбда-выражения.**](#_jr1ez2j2k1ns) **60**

[**===40. Обобщённые делегаты .NET. Action, Func, Predicate**](#_ckt8hlvf6d4) **62**

[**===41. События и делегаты.**](#_f1wf07wd8vjp) **63**

[**===42. Стандартные коллекции .NET. Типы коллекций.**](#_1u6hl884n2tp) **65**

[**===43. Стандартные интерфейсы коллекций.**](#_3tucywn9cyni) **67**

[**===44. IEnumerable и IEnumerator**](#_fi8us788c9bv) **69**

[**===45. LINQ to Objects. Синтаксис. Форма. Возврат результата. Грамматика выражений запросов. Отложенные и не отложенные операции.**](#_r63o2b7f2pm8) **69**

[**===46. LINQ to Objects. Операции Where, Select, Take, OrderBy, Join, GroupBy**](#_ubdve0sx3jgf) **73**

[**===47. Рефлексия. System Type.**](#_41ecczxgjq35) **77**

[**===48. Классы для работы с файловой системой.**](#_lhpsvmqqyxh8) **78**

[**===49. Синтаксическая конструкция using. Чтение и запись файлов. Потоковые классы.**](#_3zh4cicqfn0l) **80**

[**===50. Классы адаптеры потоков.**](#_4qy9nlw3s8wu) **82**

[**===51. Сериализация. Форматы сериализации.**](#_bl7a1epyecag) **85**

[**===52. Сериализация контрактов данных. интерфейс ISerializable.**](#_kmh5rs4e93o8) **88**

[**===53. Атрибуты. Создание собственного атрибута.**](#_8zkew264v0mm) **91**

[**===54. Процесс. Домен приложений. Поток выполнения.**](#_949jjdrjbsql) **93**

[**===55. Создание потоков , классы приоритетов. Состояния потоков**](#_hyvcaqg54ntl) **96**

[**===56. Синхронизация потоков. Lock. Monitor. Мutex. Semaphore**](#_ampxa5g702ix) **99**

[**===57. Библиотека параллельных задач TPL. Класс Task. Состояние задачи.**](#_oluv7xv50tyg) **104**

[**===58. Способы создания Task. Возврат результата. Отмена выполнения задач. Продолжения.**](#_xttpk6gc4db0) **106**

[**===59. Параллелизм при императивной обработке данных. Класс Parallel**](#_cis53sjzzzlj) **108**

[**===60. Асинхронные методы. async и await**](#_h9j54ilahosy) **111**

[**===61. Проектирование отношений. Агрегация, композиция и ассоциация**](#_ayhzpsv76cru) **112**

Вопросы

# ===51. Сериализация. Форматы сериализации.

**Сериализация** представляет процесс преобразования какого-либо объекта в поток байтов. После преобразования мы можем этот поток байтов или записать на диск или сохранить его временно в памяти. А при необходимости можно выполнить обратный процесс - **десериализацию**, то есть получить из потока байтов ранее сохраненный объект.

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом Serializable:

[Serializable]

class Person

{

}

При отсутствии данного атрибута объект Person не сможет быть сериализован, и при попытке сериализации будет выброшено исключение *SerializationException*.

Сериализация применяется к свойствам и полям класса. Если мы не хотим, чтобы какое-то поле класса сериализовалось, то мы его помечаем атрибутом *NonSerialized*:

[NonSerialized]

public string accNumber;

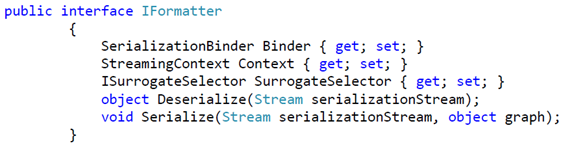
При наследовании подобного класса атрибут Serializable автоматически не наследуется.

Форматы сериализации:

**BinaryFormatter**, для формата SOAP - класс **SoapFormatter**, для xml - **XmlSerializer**, для json - **DataContractJsonSerializer**.

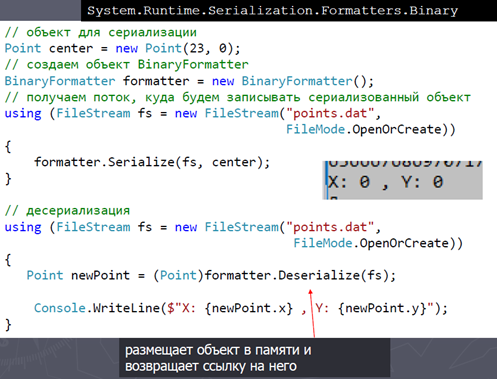
* Бинарный (класс *BinaryFormatter*) (это же и есть двоичная сериализация *мистер кэп*)
* SOAP (класс *SoapFormatter*)
* xml (класс *XmlSerializer*)
* JSON (класс *DataContractJsonSerializer*)

Для классов *BinaryFormatter* и *SoapFormatter* сам функционал сериализации определен в интерфейсе **IFormatter**:



BinaryFormatter сериализация:

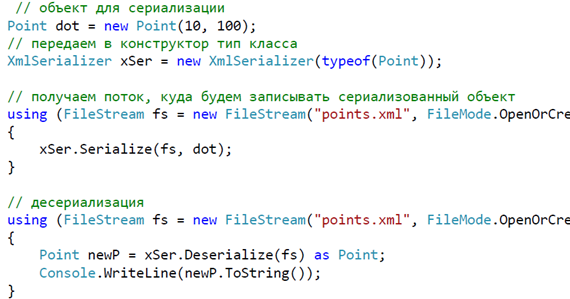
сохраняются данные полей объектов, полное квалифицированное имя каждого типа и полное имя определяющей его сборки (имя, версия, маркер общедоступного ключа и культура)

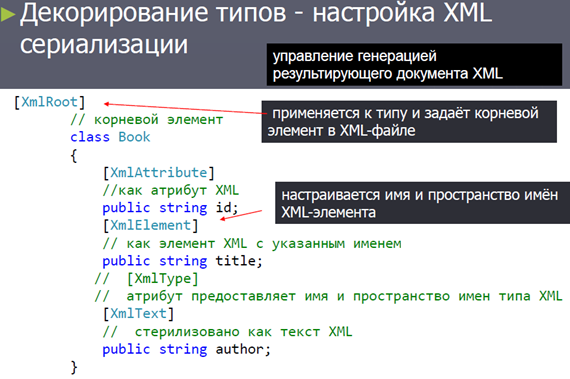


XmlSerializer сериализация

Ограничения:

* Класс должен иметь конструктор без параметров
* Сериализации подлежать только открытые члены
* Приватные данные, не представленные свойствами, игнорируются.
* Требует указания типа





# ===52. Сериализация контрактов данных. интерфейс ISerializable.

*Контракт данных* - формальное соглашение между службой и клиентом, абстрактно описывающее данные, обмен которыми происходит. Это значит, что для взаимодействия клиент и служба не обязаны совместно использовать одни и те же типы, достаточно совместно использовать одни и те же контракты данных.

*Контракт данных* – это тип (класс или структура), объект которого описывает информационный фрагмент (открытые поля и свойства).

Использование сериализатора контрактов данных:

* выбрать класс для использования: DataContractSerializer или NetDataContractSerializer
* добавить сериализуемым типам атрибут [DataContract], а их членам, которые необходимо включить в сериализацию - [DataMember]
* создать экземпляр сериализатора и вызвать его метод WriteObject или ReadObject

Существует два сериализатора контрактов данных:

* DataContractSerializer — обеспечивает слабую привязку типов .NET к типам контрактов данных. Может генерировать совместимый со стандартами XML код. Требует предварительной явной регистрации сериализуемых производных типов, чтобы иметь возможность сопоставлять имена контрактов данных с именами типов .NET
* NetDataContractSerializer — характеризуется тесной привязкой типов .NET к типам контрактов данных, не требует явной регистрации сериализуемых производных типов, т.к. самостоятельно записывает полные имена типов и сборок сериализуемых типов

*namespace SerialTest*

*{*

*[DataContract] public class Person*

*{*

*[DataMember] public string Name;*

*[DataMember] public int Age;*

*}*

*}*

// …..

*Person p = new Person { Name = "Stacey", Age = 30 };*

*var ds = new DataContractSerializer (typeof (Person));*

*using (Stream s = File.Create ("person.xml"))*

*ds.WriteObject (s, p); // Сериализация*

*Person p2;*

*using (Stream s = File.OpenRead ("person.xml"))*

*p2 = (Person) ds.ReadObject (s); // Десериализация*

*Console.WriteLine (p2.Name + " " + p2.Age); // Stacey 30*

Конструктору DataContractSerializer необходимо передать тип корневого объекта — сериализуемый объект, который в XML дереве будет корневым элементом. Конструктор NetDataContractSerializer этого не требует. В остальном их применение аналогично.

Оба типа сериализаторов по умолчанию принимаю форматер XML, поэтому с ними можно использовать XmlWriter:

*Person p = new Person { Name = "Stacey", Age = 30 };*

*var ds = new DataContractSerializer (typeof (Person));*

*// Добавить в вывод отступы:*

*XmlWriterSettings settings = new XmlWriterSettings() { Indent = true };*

*using (XmlWriter w = XmlWriter.Create ("person.xml", settings))*

*ds.WriteObject (w, p);*

*System.Diagnostics.Process.Start ("person.xml");*

Имена XML элементов в выводе соответствуют именам контрактов данных, которые по умолчанию совпадают с именами типов в .NET, однако это можно переопределить с помощью атрибутов, задав альтернативное имя для элемента:

*[DataContract (Name="Candidate")]*

*public class Person { ... }*

*Также можно изменить пространство имен по умолчанию для корневого элемента:*

*[DataContract (Namespace="http://somedomain.com/contacts")]*

*public class Person { ... }*

*Можно также переопределить имена членов данных:*

*[DataContract (Name="Candidate", Namespace="http://oreilly.com/nutshell")]*

*public class Person*

*{*

*[DataMember (Name="FirstName")] public string Name;*

*[DataMember (Name="ClaimedAge")] public int Age;*

*}*

**Интерфейс ISerializable**

Реализация интерфейса **ISerializable** предоставляет типу полный контроль над тем, как производится его двоичная сериализация и десериализация. Определение интерфейса выглядит следующим образом:

*public interface ISerializable*

*{*

*void GetObjectData (SerializationInfo info, StreamingContext context);*

*}*

Метод **GetObjectData** запускается при сериализации и наполняет объект SerializationInfo (словарь пар имя-значение) данными из всех полей, подлежащих сериализации.

Рекомендуется объявлять метод GetObjectData как *virtual*, чтобы производные классы могли расширять сериализацию не реализуя заново интерфейс ISerializable.

Параметры GetObjectData:

* SerializationInfo - представляет из себя нечто похожее на коллекцию "Dictionary". Сюда с помощью метода "AddValue()" добавляется ключ и значение. А при десериализации при помощи “GetValue()” это значение будет вытаскиваться по ключу.
* StreamingContext - это
  + структура, помимо прочего содержащая значения enum, указывающие откуда поступает сериализованный экземпляр
  + объект, который содержит дополнительную информацию о потоке, в который записываются эти данные

*(выбирайте сами определение, первое из ответов на билеты, но второе мне больше нравится)*

Помимо реализации интерфейса ISerializable, тип, управляющий собственной сериализацией *должен содержать конструктор десериализации*, принимающий те же два параметра, что и метод GetObjectData. Он может быть объявлен с любым модификатором доступа.

*[Serializable]*

*public class Team : ISerializable*

*{*

*public string Name;*

*public List<Person> Players;*

*public virtual void GetObjectData (SerializationInfo si, StreamingContext sc)*

*{*

*si.AddValue ("Name", Name);*

*si.AddValue ("PlayerData", Players.ToArray());*

*}*

*public Team() {}*

*// Конструктор десериализации*

*protected Team (SerializationInfo si, StreamingContext sc)*

*{*

*Name = si.GetString ("Name");*

*Person[] a = (Person[]) si.GetValue ("PlayerData", typeof (Person[]));*

*Players = new List<Person> (a);*

*}*

*}*

# ===53. Атрибуты. Создание собственного атрибута.

**Атрибуты** позволяют добавлять *пользовательскую информацию* к метаданным элементов кода: сборкам, типам, членам, возвращаемым значениям и параметрам. Например, атрибуты могут быть использованы для *сериализации* (а именно [Serializable] и [NonSerialized]) — процесс преобразования произвольных объектов в определенный формат. В этом случае атрибуты полей могут точно устанавливать как преобразовывать поля.

В .NET атрибуты представляют собой типы классов, которые расширяют абстрактный базовый класс System.Attribute.

Для создания собственного атрибута нужно:

* создать класс, производный от System.Attribute или его потомка; по соглашению имя класса должно заканчиваться словом Attribute (но не обязательно)
* применить к этому классу описанный выше атрибут AttributeUsage. *Если атрибут не требует свойств или аргументов конструктора, на этом его создание завершено*
  + Тут же указываются ограничения для атрибута (т.е. где он может применяться). Их задает перечисление AttributeTargets, значения которого могут быть:
    - All: используется всеми типами
    - Assembly: атрибут применяется к сборке
    - Constructor: атрибут применяется к конструктору
    - Delegate: атрибут применяется к делегату
    - Enum: применяется к перечислению
    - Event: атрибут применяется к событию
    - Field: применяется к полю типа
    - Interface: атрибут применяется к интерфейсу
    - Method: применяется к методу
    - Property: применяется к свойству
    - Struct: применяется к структуре

Пример задавания ограничения: [AttributeUsage (AttributeTargets.Method)]

* добавить в класс один или несколько публичных конструкторов. Параметры конструктора определяют позиционные параметры конструктора и становятся обязательными при использовании атрибута.

Параметры атрибута могут быть:

* + примитивные типы: bool, byte, char, double, float, int, long, short или string
  + Type
  + enum
  + одномерный массив любого из указанных выше типов
* объявить публичное свойство или поле для каждого именованного параметра атрибута

Пример:

*[AttributeUsage (AttributeTargets.Method)] // <= атрибут с ограничением*

*public sealed class TestAttribute : Attribute*

*{*

*public int Repetitions; // публичное свойство для int repetitions*

*public string FailureMessage;*

*public TestAttribute () : this (1) { } // конструктор*

*public TestAttribute (int repetitions) // еще один конструктор с позиционными параметрами*

*{*

*Repetitions = repetitions;*

*}*

*}*Использование атрибута:

*class Foo*

*{*

*[Test]*

*public void Method1() { ... }*

*[Test(20)] // атрибут с параметром int repetitions*

*public void Method2() { ... }*

*[Test(20, FailureMessage="Debugging Time!")]*

*public void Method3() { ... }*

*}*

# ===54. Процесс. Домен приложений. Поток выполнения.

*вопрос на самом деле небольшой, просто свойств и методов в классах много*

При запуске приложения операционная система создает для него отдельный **процесс**, которому выделяется определённое адресное пространство в памяти и который изолирован от других процессов. Процесс может иметь несколько потоков. Как минимум, процесс содержит один - главный поток, создаваемый при вызове метода Main. А из главного потока могут запускаться вторичные потоки.

В .NET процесс представлен классом **Process** из пространства имен **System.Diagnostics**. Этот класс позволяет управлять уже запущенными процессами, а также запускать новые. В данном классе определено ряд свойств и методов, позволяющих получать информацию о процессах и управлять ими:

* Свойство *Handle*: возвращает дескриптор процесса
* Свойство *Id*: получает уникальный идентификатор процесса в рамках текущего сеанса ОС
* Свойство *MachineName*: возвращает имя компьютера, на котором запущен процесс
* Свойство *Modules*: получает доступ к коллекции ProcessModuleCollection, которая хранит набор модулей (файлов dll и exe), загруженных в рамках данного процесса
* Свойство *ProcessName*: возвращает имя процесса, которое нередко совпадает с именем приложения
* Свойство *StartTime*: возвращает время, когда процесс был запущен
* Свойство *VirtualMemorySize64*: возвращает объем памяти, который выделен для данного процесса
* Метод *CloseMainWindow()*: закрывает окно процесса, который имеет графический интерфейс
* Метод *GetProcesses()*: возвращающий массив всех запущенных процессов
* Метод *GetProcessesByName()*: возвращает процессы по его имени. Так как можно запустить несколько копий одного приложения, то возвращает массив
* Метод *Kill()*: останавливает процесс
* Метод *Start()*: запускает новый процесс

**Домен приложения** — это своеобразный контейнер, внутри которого запускается программа, и который изолирует ее во время выполнения. Домен задает границы управляемой памяти для приложения и является контейнером для загруженных сборок и параметров конфигурации приложения. Каждый процесс *обычно размещает только один домен приложения* — стандартный домен, автоматически созданный CLR при запуске процесса. Однако в рамках одного процесса возможно создавать дополнительные домены приложений.

Домены:

1) существуют внутри процессов

2) содержат загруженные сборки

3) процесс запускает при старте домен по умолчанию (AppDomain.CurrentDomain)

4) домены могут создаваться и уничтожаться в ходе работы в рамках процесса (менее затраты по сравн. с процессами)

5) обеспечивают уровень изоляции кода

Для управления домена платформа .NET предоставляет класс **AppDomain**. Рассмотрим некоторые основные методы и свойства данного класса:

* Свойство *BaseDirectory*: базовый каталог, который используется для получения сборок (как правило, каталог самого приложения)
* Свойство *CurrentDomain*: домен текущего приложения
* Свойство *FriendlyName*: имя домена приложения
* Свойство *SetupInformation*: представляет объект AppDomainSetup и хранит конфигурацию домена приложения
* Метод *CreateDomain()*: статический метод, позволяющий создать новый домен приложения в рамках текущего процесса
* Метод *CreateInstance()*: создает экземпляр типа из внешней сборки после ее загрузки в домен приложения
* Метод *ExecuteAssembly()*: запускает сборку exe в рамках текущего домена приложения
* Метод *GetAssemblies()*: получает набор сборок .NET, загруженных в домен приложения
* Метод *Load()*: загружает сборку домена приложения
* Метод *Unload()*: выгружает домен приложения из определенного процесса

**Потоки процесса**

Свойство Threads представляет коллекцию потоков процесса - объект ProcessThreadCollection, каждый поток в которой является объектом **ProcessThread**. В данном классе можно выделить следующие свойства:

* *CurrentPriority*: возвращает текущий приоритет потока
* *Id*: идентификатор потока
* *IdealProcessor*: позволяет установить процессор для обработки потока
* *PriorityLevel*: уровень приоритета потока
* *StartAddress*: адрес в памяти функции, запустившей поток
* *StartTime*: время запуска потока

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен **System.Threading**. В нем определен класс, представляющий отдельный поток - класс **Thread**.

Класс **Thread** определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства и методы класса:

* Статическое свойство *CurrentContext* позволяет получить контекст, в котором выполняется поток
* Статическое свойство *CurrentThread* возвращает ссылку на выполняемый поток
* Свойство *IsAlive* указывает, работает ли поток в текущий момент
* Свойство *IsBackground* указывает, является ли поток фоновым
* Свойство *Name* содержит имя потока
* Свойство *Priority* хранит приоритет потока - значение перечисления ThreadPriority
* Свойство *ThreadState* возвращает состояние потока - одно из значений перечисления ThreadState
* Статический метод *GetDomain()* возвращает ссылку домен приложения
* Статический метод *GetDomainId()* возвращает id домена приложения, в котором выполняется текущий поток
* Статический метод *Sleep()* останавливает поток на определенное количество миллисекунд
* Метод *Abort()* уведомляет среду CLR о том, что надо прекратить поток, однако прекращение работы потока происходит не сразу, а только тогда, когда это становится возможно. Для проверки завершенности потока следует опрашивать его свойство ThreadState
* Метод *Interrupt()* прерывает поток на некоторое время
* Метод *Join()* блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
* Метод *Resume()* возобновляет работу ранее приостановленного потока
* Метод *Start()* запускает поток
* Метод *Suspend()* приостанавливает поток

# ===55. Создание потоков , классы приоритетов. Состояния потоков

Используя класс Thread, мы можем выделить в приложении несколько потоков, которые будут выполняться одновременно. Во-первых, для запуска нового потока нам надо определить задачу в приложении, которую будет выполнять данный поток. Для этого мы можем добавить новый метод, производящий какие-либо действия.

Для создания нового потока используется делегат ThreadStart, который получает в качестве параметра метод, который мы определил выше.

И чтобы запустить поток, вызывается метод Start. Рассмотрим на примере:

using System.Threading;

class Program

{

static void Main(string[] args)

{

// создаем новый поток

Thread myThread = new Thread(new ThreadStart(Count));

myThread.Start(); // запускаем поток

for (int i = 1; i < 9; i++)

{

Console.WriteLine("Главный поток:");

Console.WriteLine(i \* i);

Thread.Sleep(300);

}

Console.ReadLine();

}

public static void Count()

{

for (int i = 1; i < 9; i++)

{

Console.WriteLine("Второй поток:");

Console.WriteLine(i \* i);

Thread.Sleep(400);

}

}

}

Здесь новый поток будет производить действия, определенные в методе Count. В данном случае это возведение в квадрат числа и вывод его на экран. И после каждого умножения с помощью метода Thread.Sleep мы усыпляем поток на 400 миллисекунд.

Чтобы запустить этот метод в качестве второго потока, мы сначала создаем объект потока: Thread myThread = new Thread(new ThreadStart(Count));. В конструктор передается делегат ThreadStart, который в качестве параметра принимает метод Count. И следующей строкой myThread.Start() мы запускаем поток. После этого управление передается главному потоку, и выполняются все остальные действия, определенные в методе Main.

Таким образом, в нашей программе будут работать одновременно главный поток, представленный методом Main, и второй поток. Кроме действий по созданию второго потока, в главном потоке также производятся некоторые вычисления. Как только все потоки отработают, программа завершит свое выполнение.

Существует еще одна форма создания потока: Thread myThread = new Thread(Count); Хотя в данном случае явным образом мы не используем делегат ThreadStart, но неявно он создается. Компилятор C# выводит делегат из сигнатуры метода Count и вызывает соответствующий конструктор.

Приоритеты потоков располагаются в перечислении ThreadPriority:

· Lowest

· BelowNormal

· Normal

· AboveNormal

· Highest

По умолчанию потоку задается значение Normal. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет Highest. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

Состояния потоков

Свойство ThreadState возвращает состояние потока - одно из значений перечисления ThreadState

# ===56. Синхронизация потоков. Lock. Monitor. Мutex. Semaphore

Нередко в потоках используются некоторые разделяемые ресурсы, общие для всей программы. Это могут быть общие переменные, файлы, другие ресурсы. Например:

class Program

{

static int x=0;

static void Main(string[] args)

{

for (int i = 0; i < 5; i++)

{

Thread myThread = new Thread(Count);

myThread.Name = "Поток " + i.ToString();

myThread.Start();

}

Console.ReadLine();

}

public static void Count()

{

x = 1;

for (int i = 1; i < 9; i++)

{

Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

X++;

Thread.Sleep(100);

}

}

}

Здесь у нас запускаются пять потоков, которые работают с общей переменной x. И мы предполагаем, что метод выведет все значения x от 1 до 8. И так для каждого потока. Однако в реальности в процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым.

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком. Для этого используется ключевое слово lock. Оператор lock определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока. И мы можем переделать предыдущий пример следующим образом:

class Program

{

static int x=0;

static object locker = new object();

static void Main(string[] args)

{

for (int i = 0; i < 5; i++)

{

Thread myThread = new Thread(Count);

myThread.Name = "Поток " + i.ToString();

myThread.Start();

}

Console.ReadLine();

}

public static void Count()

{

lock (locker)

{

x = 1;

for(int i = 1; i < 9; i++)

{

Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

X++;

Thread.Sleep(100);

}

}

}

}

Для блокировки с ключевым словом lock используется объект-заглушка, в данном случае это переменная locker. Когда выполнение доходит до оператора lock, объект locker блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток. После окончания работы блока кода, объект locker освобождается и становится доступным для других потоков.

Monitor

Наряду с оператором lock для синхронизации потоков мы можем использовать мониторы, представленные классом System.Threading.Monitor. Фактически конструкция оператора lock из прошлой темы инкапсулирует в себе синтаксис использования мониторов. А рассмотренный в прошлой теме пример будет эквивалентен следующему коду:

class Program

{

static int x=0;

static object locker = new object();

static void Main(string[] args)

{

for (int i = 0; i < 5; i++)

{

Thread myThread = new Thread(Count);

myThread.Name = "Поток " + i.ToString();

myThread.Start();

}

Console.ReadLine();

}

public static void Count()

{

try

{

Monitor.Enter(locker);

x = 1;

for(int i = 1; i < 9; i++)

{

Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

X++;

Thread.Sleep(100);

}

}

Finally {

Monitor.Exit(locker);

}

}

}

Метод Monitor.Enter блокирует объект locker так же, как это делает оператор lock. А в блоке try...finally с помощью метода Monitor.Exit Происходит освобождение объекта locker, и он становится доступным для других потоков.

Кроме блокировки и разблокировки объекта класс Monitor имеет еще ряд методов, которые позволяют управлять синхронизацией потоков. Так, метод Monitor.Wait освобождает блокировку объекта и переводит поток в очередь ожидания объекта. Следующий поток в очереди готовности объекта блокирует данный объект. А все потоки, которые вызвали метод Wait, остаются в очереди ожидания, пока не получат сигнала от метода Monitor.Pulse или Monitor.PulseAll, посланного владельцем блокировки. Если метод Monitor.Pulse отправлен, поток, находящийся во главе очереди ожидания, получает сигнал и блокирует освободившийся объект. Если же метод Monitor.PulseAll отправлен, то все потоки, находящиеся в очереди ожидания, получают сигнал и переходят в очередь готовности, где им снова разрешается получать блокировку объекта.

Мьютекс

System.Threading.Mutex - позволяет организовать критическую секцию для нескольких процессов

WaitOne() - входа в критическую секцию

ReleaseMutex() – для выхода из неё (выход может быть произведен только в том же потоке выполнения, что и вход).

Семафор

объект синхронизации, позволяющий войти в заданный участок кода не более чем N потокам (N – ёмкость семафора) получение и снятие блокировки в случае семафора может выполняться из разных потоков

**классы**

System.Threading.Semaphore (между процессами) и SemaphoreSlim(в рамках одного процесса)

Wait()-получение блокировки

Release()–снятие блокировки

# ===57. Библиотека параллельных задач TPL. Класс Task. Состояние задачи.

Библиотека параллельных задач TPL (Task Parallel Library) позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах (для создания многопоточных приложений)

using System.Threading.Tasks

Планировщик библиотеки выполняет диспетчеризацию задач, а также предоставляет единообразный механизм отмены задач и обработки исключительных ситуаций.

В основе библиотеки TPL лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию. В библиотеке классов .NET задача представлена специальным классом - классом Task, который находится в пространстве имен System.Threading.Tasks. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке.

Для определения и запуска задачи можно использовать различные способы. Первый способ создание объекта Task и вызов у него метода Start:

1 Task task = new Task(() => Console.WriteLine("Hello Task!"));

2 task.Start();

В качестве параметра объект Task принимает делегат Action, то есть мы можем передать любое действие, которое соответствует данному делегату, например, лямбда-выражение, как в данном случае, или ссылку на какой-либо метод. То есть в данном случае при выполнении задачи на консоль будет выводиться строка "Hello Task!", а метод Start() собственно запускает задачу.

Второй способ заключается в использовании статического метода Task.Factory.StartNew(). Этот метод также в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться. При этом этот метод сразу же запускает задачу:

1 Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello Task!"));

В качестве результата метод возвращает запущенную задачу.

Третий способ определения и запуска задач представляет использование статического метода Task.Run():

1 Task task = Task.Run(() => Console.WriteLine("Hello Task!"));

Метод Task.Run() также в качестве параметра может принимать делегат Action - выполняемое действие и возвращает объект Task.

Свойства класса Task

Класс Task имеет ряд свойств, с помощью которых мы можем получить информацию об объекте. Некоторые из них:

· AsyncState: возвращает объект состояния задачи

· CurrentId: возвращает идентификатор текущей задачи

· Exception: возвращает объект исключения, возникшего при выполнении задачи

· Status: возвращает статус задачи

# ===58. Способы создания Task. Возврат результата. Отмена выполнения задач. Продолжения.

*Способы создания расписаны в 57 вопросе*

**Возврат результата:**

Задача может возвращать значение. Это очень удобно по двум причинам:

* с помощью задачи можно вычислить некоторый результат. Подобным образом *поддерживаются параллельные вычисления.*
* вызывающий процесс окажется заблокированным до тех пор, пока не будет получен результат. Это означает, что для *организации ожидания результата не требуется никакой особой синхронизации*.

Для того, чтобы возвратить результат из задачи, достаточно создать эту задачу, используя обобщенную форму Task<TResult> класса Task. Ниже приведены два конструктора этой формы класса Task:

*public Task(Func<TResult> функция)*

*public Task(Func<Object, TResult> функция, Object состояние)*

Пример возврата значения из задачи:

*static bool MyTask() { /\* какой-то код функции \*/ }* // определение метода MyTask

// … далее пишется в Main()

*Task<bool> tsk = Task<bool>.Factory.StartNew(MyTask); // в tsk результат задачи*

// … это в самом конце Main()

*tsk.Dispose();* // уничтожение объекта

**Отмена выполнения задач:**

*лучше сразу смотреть пример имо. Так непонятно. Еще лучше проверить в вижле.*

Признак *отмены* является экземпляром объекта типа **CancellationToken**, т.е. структуры, определенной в пространстве имен **System.Threading**. В структуре CancellationToken определено несколько свойств и методов, но мы рассмотрим два из них:

* доступное только для чтения свойство **IsCancellationRequested** - возвращает true, если отмена задачи была запрошена для вызывающего признака, иначе false
* Метод **ThrowIfCancellationRequested()** - выкидывает ошибку, если задача была отменена

*CancellationTokenSource cancelTokenSource = new CancellationTokenSource(); // создаем объект CancellationTokenSource*

*CancellationToken token = cancelTokenSource.Token; // берез токен объекта*

*Task task1 = new Task(() =>*

*{*

*while (true)*

*{*

*if (token.IsCancellationRequested) // проверка, если потребовалась отмена*

*{*

*Console.WriteLine("Операция прервана");*

*return; // задача прерывается*

*}*

*Console.WriteLine("...");*

*Thread.Sleep(500); // многоточие пишется каждые полсекунды*

*}*

*});*

*task1.Start(); // задача начинается*

*while (true) {*

*ConsoleKeyInfo s = Console.ReadKey(); // если ввести “q”, то задача прекратится*

*if (s.KeyChar == 'q')*

*{*

*cancelTokenSource.Cancel(); // отправляет запрос на отмену задачи*

*break;*

*}*

*}*

*Console.WriteLine("Введите клавишу, чтобы завершить программу...");*

*Console.ReadKey();*

**Продолжение задачи** (continuation task): сообщает задаче, что после её завершения она должна продолжить делать что-то другое

static void Main(string[] args)

{

Task task1 = new Task(() => {

Console.WriteLine("1 задача");

});

// задача продолжения

Task task2 = task1.ContinueWith(Display);

task1.Start();

// ждем окончания второй задачи

task2.Wait();

Console.WriteLine("Выполняется работа метода Main");

Console.ReadLine();

}

static void Display(Task t)

{

Console.WriteLine("2 задача");

Thread.Sleep(3000);

}

}

# ===59. Параллелизм при императивной обработке данных. Класс Parallel

Класс Parallel также является частью TPL и предназначен для упрощения параллельного выполнения кода. Parallel имеет ряд методов, которые позволяют распараллелить задачу.

Одним из методов, позволяющих параллельное выполнение задач, является метод Invoke:

static void Main(string[] args)

{

Parallel.Invoke(Display,

() => {

Console.WriteLine("Выполняется задача {0}", Task.CurrentId);

Thread.Sleep(3000);

},

() => Factorial(5));

Console.ReadLine();

}

static void Display()

{

Console.WriteLine("Выполняется задача {0}", Task.CurrentId);

Thread.Sleep(3000);

}

static void Factorial(int x)

{

int result = 1;

for (int i = 1; i <= x; i++)

{

result \*= i;

}

Console.WriteLine("Выполняется задача {0}", Task.CurrentId);

Thread.Sleep(3000);

Console.WriteLine("Результат {0}", result);

}

Метод Parallel.Invoke в качестве параметра принимает массив объектов Action, то есть мы можем передать в данный метод набор методов, которые будут вызываться при его выполнении. Количество методов может быть различным, но в данном случае мы определяем выполнение трех методов. Опять же как и в случае с классом Task мы можем передать либо название метода, либо лямбда-выражение. И таким образом, при наличии нескольких ядер на целевой машине данные методы будут выполняться параллельно на различных ядрах.

**Parallel.For**

Метод Parallel.For позволяет выполнять итерации цикла параллельно. Он имеет следующее определение: For(int, int, Action<int>), где первый параметр задает начальный индекс элемента в цикле, а второй параметр - конечный индекс. Третий параметр - делегат Action - указывает на метод, который будет выполняться один раз за итерацию.

**Parallel.ForEach**

Метод Parallel.ForEach осуществляет итерацию по коллекции, реализующей интерфейс IEnumerable, подобно циклу foreach, только осуществляет параллельное выполнение перебора. Он имеет следующее определение: ParallelLoopResult ForEach<TSource>(IEnumerable<TSource> source, Action<TSource> body), где первый параметр представляет перебираемую коллекцию, а второй параметр - делегат, выполняющийся один раз за итерацию для каждого перебираемого элемента коллекции.

Здесь метод Factorial, обрабатывающий каждую итерацию, принимает дополнительный параметр - объект ParallelLoopState. И если счетчик в цикле достигнет значения 5, вызывается метод Break. Благодаря чему система осуществит выход и прекратит выполнение метода Parallel.For при первом удобном случае.

Методы Parallel.ForEach и Parallel.For возвращают объект ParallelLoopResult, наиболее значимыми свойствами которого являются два следующих:

· IsCompleted: определяет, завершилось ли полное выполнение параллельного цикла

· LowestBreakIteration: возвращает индекс, на котором произошло прерывание работы цикла

# ===60. Асинхронные методы. async и await

Операторы async и await используются вместе для создания асинхронного метода. Такой метод, определенный с помощью модификатора async и содержащий одно или несколько выражений await, называется **асинхронным методом.**

При асинхронном вызове поток выполнения разделяется на две части:

* в одной выполняется метод,
* а в другой –процесс программы.

Асинхронный вызов методов реализуется средой исполнения при помощи пула

Потоков

**Назначение:**

* Доступ к web HttpClient
* Работа с файлами
* Работа с изображениями
* WCF программирование

Ключевое слово **async** указывает, что метод или лямбда-выражение являются асинхронными. А оператор **await** применяется к задаче в асинхронных методах, чтобы приостановить выполнение метода до тех пор, пока эта задача не завершится. При этом выполнение потока, в котором был вызван асинхронный метод, не прерывается.

class Program

{

static void Main(string[] args)

{

DisplayResultAsync();

Console.ReadLine();

}

static async void DisplayResultAsync()

{

int num = 5;

int result = await FactorialAsync(num);

Thread.Sleep(3000);

Console.WriteLine("Факториал числа {0} равен {1}", num, result);

}

static Task<int> FactorialAsync(int x)

{

int result = 1;

return Task.Run(() =>

{

for (int i = 1; i <= x; i++)

{

result \*= i;

}

return result;

});

}

}

Итак, здесь определен метод, возвращающий объект Task - задачу, которая выполняется асинхронно. И так так мы получаем факториал целого числа, то объект задачи типизирован типом int.

Сам факториал мы получаем в асинхронном методе DisplayResultAsync. Асинхронным он является потому, что он объявлен с модификатором async и содержит использование ключевого слова **await.**

По негласным правилам в названии асинхроннных методов принято использовать суффикс Async.

# ===61. Проектирование отношений. Агрегация, композиция и ассоциация

UML (Unified Modeling Language) - унифицированный язык моделирования. Он определил следующие типы связей:

* **Ассоциации** - отношение, где два класса как-то связаны между собой, т.е. один включен в другой. Типы ассоциаций:
  + **Агрегация** - это ассоциация между классом A (часть) и классом B (целое), которая означает, что экземпляры (один или несколько) класса A ходят в состав экземпляра класса B.

Объекты могут быть равноправными; один *включает* в себя другой; целое хоть и содержит свою составную часть, *время их жизни не связано*

*Агрегация* может быть частью нескольких объектов.

* + **Композиция** - это ассоциация между классом A (часть) и классом B (целое), которая дополнительно накладывает более сильные ограничения в сравнении с агрегацией: композиционно часть A может входить только в одно целое B, часть существует, только пока существует целое и прекращает свое существование вместе с целым.

Объекты *не равноправны*; один *состоит* из другого; целое *контролирует время жизни* своей составной части (часть не может существовать без целого)

*Композиция* может быть частью одного и только одного целого.

* **зависимости**
* **обобщения** (наследования)

Отношение *обобщения* часто применяется на диаграмме классов. Действительно, трудно представить себе ситуацию, когда между классами в одной системе нет ничего общего. Как правило, общее есть, и это общее целесообразно выделить в отдельный класс. При этом общие составляющие, собранные в суперклассе, автоматически наследуются подклассами.

Хороший пример для понимания различия агрегации и композиции.

“Приведем простой пример: У нас есть наш *Car* — машина. У каждой машины есть двигатель. Кроме того, у каждой машины есть пассажиры внутри.

В чем же принципиальная разница между полями *Engine engine* и *Passenger [] passengers*? Если у машины внутри сидит пассажир А, это не значит, что в ней не могут находиться пассажиры B и C. Одна машина может соответствовать нескольким пассажирам. Кроме того, если всех пассажиров высадить из машины, она продолжит спокойно функционировать.

Связь между классом *Car* и массивом пассажиров *Passenger [] passengers* менее строгая. Она называется **агрегацией**.

**Композиция** — более строгий тип связи. При использовании композиции объект не только является частью какого-то объекта, но и не может принадлежать другому объекту того же типа.

Самый простой пример — двигатель автомобиля. Двигатель является частью автомобиля, но не может быть частью другого автомобиля. Как видишь, их связь гораздо более строгая, чем у *Car* и *Passengers*.”

Чаще всего в моделях используются бинарные ассоциации, отражающие связи между объектами двух классов. В UML определены также **многополюсные ассоциации**, отражающие связи между бóльшим числом объектов. С формальной точки зрения многополюсные ассоциации излишни, поскольку их можно выразить через комбинацию бинарных ассоциаций введением дополнительных сущностей. Действительно, упорядоченную тройку объектов (a, b, c) ‒ элемент трехполюсной ассоциации ‒ можно представить как упорядоченную пару (a, d), где d ‒ новый объект, представляющий упорядоченную пару (b, c).

Диаграммы классов (принципы):

* Описывать структуру удобнее параллельно с описанием поведения.
* Каждая итерация должна быть небольшим уточнением
* Не обязательно включать в модель все классы сразу
* Не обязательно определять все составляющие класса сразу.
* Не обязательно определять все отношения между классами сразу