



Подготовка к подбелу



by Денис Землянухин. Tg: [@Sunr1seColours](https://t.me/Sunr1seColours)

Делегаты

Делегат-тип - специальный вид класса, экземпляр которого ссылается на методы с определенным типом возвращаемого значения и списком параметров

- Является ссылочным типом данных

Наследование делегатов:

`MulticastDelegate` → `Delegate` → `Object`

Особенности делегат-типов

- Делегат-тип является наследником от класса `Delegate`
- Пользовательский делегат-тип **не** может быть напрямую унаследован от классов `Delegate` и `MulticastDelegate`
- Класс `Delegate` не является делегат-типом
- Делегат-тип опечатан

Объявление делегат-типа

```
// Пример.  
internal delegate void Print(string msg)
```

Сигнатура делегат-типа включает в себя *типы параметров, их порядок, тип возвращаемого значения*.

Делегат-тип можно объявить внутри класса, пространства имен и внутри глобального пространства имен.

Делегатом или **Экземпляром делегат-типа** хранит в себе ссылки на методы, соответствующие сигнатуре делегат-типа. Их можно назначить откуда угодно.

Все делегат-типы имеют функциональные члены

```
// Возвращает массив объектов, содержащих информацию о каждом
public Delegate[] GetInvocationList()

// Получает информацию о последнем в списке вызовов методе.
public MethodInfo Method { get; }

// Получает объект, связанный с последним методом в списке вы
// если метод статический
public object? Target { get; }
```




Делегаты хранят в себе ссылку на объект, относительно которого вызывается метод, поэтому время жизни этого объекта может быть продлено

Список вызовов делегатов

Хранит в себе ссылки на все методы, связанные с делегатом. Имеет следующие особенности:

- Допустимы одновременно ссылки на статические и на экземплярные методы
- Порядок в списке вызовов соответствует порядку добавления
- Ссылку на один и тот же метод можно добавлять более 1 раза
- Если вызывать делегат с пустым списком вызовов, то выбросится `NullReferenceException`
- `Delegate.Combine(del1, del2)` возвращает новый делегат с объединенным списком вызовов

Операции

-  позволяет получить новый делегат с объединенным списком вызовов или объединить делегат и метод с одинаковыми сигнатурами

- `-` возвращает новый делегат с списком вызовом, соответствующим разности множеств списков вызовов. Однако если один и тот же метод входит в список вызова несколько раз, то будет удалено лишь последнее его вхождение!!
- `+=` и `-=` аналогичны `+` и `-`, но не создают новый объект

Встроенные делегат-типы

```
// Принимает от 0 до 16 параметров.
void Action()
void Action<in T1, in T2, .., in T16>(T1 t1, T2 t2, .., T16 t16)

// Тоже принимает от 0 до 16 параметров.
TResult Func()
TResult Func<in T1, in T2, .., in T16>(T1 t1, T2 t2, .., T16 t16)

// Используется методами классов Array и List<T>
bool Predicate<in T>(T t)
```

Неординарные приколы

```
// Описания свойств и методов класса Delegate.
public System.Reflection.MethodInfo Method { get; }

public object? Target { get; }

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

public static Delegate? Remove (Delegate? source, Delegate? v)
```

Анонимные методы и лямбда-выражения

Объявление анонимных методов

```
// Последним типом параметров указывается тип возвращаемого значения
Func<int, int, int> sum = delegate (int x, int y)
{
    return x + y;
};
```

Анонимные методы не имеют доступа к статическим полям.

Лямбда-выражения

Являются альтернативой анонимному методу.

Различия анонимных методов и лямбда-выражения:

- Анонимные методы позволяют полностью опустить список параметров
- Лямбда-выражения позволяют опускать и выводить типы параметров
- Тело лямбда-выражения может быть выражением
- Лямбда-выражения имеют преобразования к совместимым типам деревьев

```
// Примеры лямбда-выражений.
Action lambda = () => Console.WriteLine("Hi!");
Func<int, int> alpha = x => x + 1;

// Можно использовать _ для пропуска 2 и более параметров.
Func<int, int, int> beta = (_, _) => 42;
```

Синтаксис лямбда-выражений допустим для:

- Методов
- Свойств
- Конструкторов
- Деструкторов
- Индексаторов

Потоки данных

Поток данных - абстракция, обозначающая источник получения или средство приема данных, позволяющая унифицировать процессы обмена данными между программой и внешним окружением.

Классы, представляющие потоки данных наследуются от **абстрактного** класса `Stream`

- `FileStream → Stream`
- `MemoryStream → Stream`

Потоки поддерживают операции:

- Чтения
- Записи
- Поиска

FileStream

Класс нужен для создания потоков, связанных с какими-либо файлами. Для создания объекта надо указать (явно или неявно):

- Файл, с которым будет ассоциирован поток
- Способ доступа к файлу (`FileAccess`)
- Режим открытия файла (`FileMode`)
- Доступ к совместному использованию (`FileShare`)

```
FileStream fs1 = new FileStream("electric-power.csv", FileMode.Create, FileAccess.ReadWrite, FileShare.None);  
FileStream fs1 = new FileStream("electric-power.csv", FileMode.Open, FileAccess.Read, FileShare.None);
```

`FileStream` имеет методы `Write` и `Read`, которые могут записывать и считывать данные соответственно.

Кроме того, можно создавать потоки на основе методов классов `File` и `FileInfo`:

- `Create()` - Создает новый файл и поток / открывает файл и создает поток
- `Open()` - Открывает файл и возвращает ссылку на поток
- `OpenRead()`
- `OpenWrite()`

`File` имеет методы `AppendAllText`, `WriteAllText`, `ReadAllText`, `ReadAllLines`, но не имеет методов `Write` и `Read`.

События

```
public event Action<DateTime> NotificationAppeared;
```



События могут быть вызваны только в коде того класса, где они объявлены

Для подписчиков извне доступны операции `+=` и `-=`

В .NET есть библиотечный делегат-тип для реализации стандартного событийного паттерна:

```
public delegate void EventHandler<TEventArgs>(object sender,
```



Microsoft рекомендует не использовать виртуальные события (`virtual`)

Рефлексия

Рефлексия - механизм, позволяющий программе во время своего исполнения считывать метаданные сборок.

Метаданные - данные о программах и используемых в них типах, хранимые в скомпилированных сборках.

Приколы рефлексии:

- Можно создавать типы и вызывать их методы без предыдущих знаний об именах, помещаемых в этих типах (*динамическая инедтификация типов*)
- Не нужно на этапе компиляции знать информацию о типе, из которого будут получены метаданные, достаточно название типа
- Представление типов, которые извлекаются из сборки в виде удобной объектной модели

```
// Основа всего отражение в .NET
public abstract class Type : System.Reflection.MemberInfo, Sy
```

Объявить значение переменной типа `Type` можно 3 способами:

```
// 1. Метод возвращает метаданные текущего объекта типа.
System.Object.GetType()

// 2. Не требуется объект, достаточно объявить тип.
typeof()

// 3. Не нужно объявлять тип, необходимо знать лишь имя типа.
System.Type.GetType()
```

Атрибуты

Атрибут -языковая конструкция, позволяющая добавлять метаданные к программной сборке. Примеры:

- `Serializable` указывает на возможность сериализовать объект, к которому применен данный атрибут. Не наследуется
- `NonSerializable` указывает на невозможность сериализовать объект. Наследуется

```
[Serializable]
class Student

[NonSerializable]
private string Surname
```

Атрибуты можно создавать вручную. Для этого нужно наследоваться от класса `Attribute`:

```
class DemoAttribute : Attribute { }
// При его применении можно убирать слово Attribute и писать
```

Рекомендации пользовательских атрибутов:

- Класс атрибута должен представлять некоторое состояние цели, ради которой он применяется
- Если для атрибута требуются поля, стоит добавить параметрический конструктор для сбора значений
- Не стоит реализовывать публичные методы и другие функциональные члены, за исключением свойств
- В целях безопасности стоит объявить класс атрибута опечатанным
- Стоит использовать атрибут `AttributeUsage` при объявлении собственного атрибута, чтобы явно указать множество его целей.

Цели атрибута указываются с помощью слов: `event`, `field`, `method`, `param`, `property`, `return`, `type`, `typevar`, `assembly`, `module`.

Ограничения использования атрибутов:

| Атрибут | Роль | Значение по умолчанию |
|----------------------------|---|-----------------------|
| <code>ValidOn</code> | Хранит список типов целей, к которым может применяться атрибут. Первый параметр конструктора должен быть значением перечислимого типа | |
| <code>Inherited</code> | Булево значение, указывающее на то, может ли атрибут наследоваться производными классами декорированного типа | <code>true</code> |
| <code>AllowMultiple</code> | Булево значение, указывающее можно ли к цели применять сразу несколько экземпляров атрибута текущего типа | <code>false</code> |

Перечисление `AttributeTargets` содержит в себе значения: `All`, `Delegate`, `GenericParameter`, `Parameter`, `Assembly`, `Enum`, `Interface`, `Property`, `Class`, `Event`, `Method`, `ReturnValue`, `Constructor`, `Field`, `Module`, `Struct` (Могут быть нужны в конструкторе `AttributeUsage`)

Сериализация

Сериализация - процесс преобразования структуры данных в последовательность байтов (или XML узлов, или JSON..).

Десериализация - восстановление структуры данных из последовательности байтов.

В .NET есть несколько механизмов сериализации:

- Контракты данных
- Двоичная
- SOAP-сериализация
- XML-сериализация
- `IXmlSerializable`
- JSON-сериализация (любимка)

Контракт данных - формальное соглашение между службой и клиентом, абстрактно описывающее данные, обмен которыми происходит

Для контрактов данных используются атрибуты `[DataContract]` (для типов), `[DataMember]`, `[EnumMember]`

Шаги сериализации:

1. Создать объект класса
2. Создать байтовый поток и связать его с потоком для записи
3. Создать объект сериализации (форматер)
4. Используя метод `Serialize()` или `WriteObject()` объекта-форматера, сохранить в потоке представление объекта
5. Закрыть поток



Чтобы объект можно было двоично сериализовать, необходимо пометить класс атрибутом `Serializable`. При двоичной сериализации сериализуются все открытые и закрытые нестатические поля, не помеченные атрибутом `NonSerializable`



При XML и SOAP сериализации сериализуются только открытые свойства и поля



При JSON сериализации сериализуются только открытые свойства



Чтобы десериализовать объект, он должен иметь конструктор без параметров!

Адаптеры

Адаптерами являются классы: `BinaryReader`, `BinaryWriter`, `TextWriter`, `TextReader`, `StreamReader`, `StreamWriter`.

Обобщения

Обобщенное программирование - стиль программирования, при котором алгоритм описывается в терминах отложенного определения типов данных.



Обобщения применимы для классов, структур, интерфейсов, методов, делегатов.

Обобщенные типы могут быть:

- Открыто-сконструированными

```
class MyGenericClass<T> { }
```

- Закрыто-сконструированными

```
MyGenericClass<int>
```



У различных закрытых классов будет свой **независимый** набор статических полей

Ограничения

Для обобщенных типов можно указывать ограничения:

```
// where - контекстно-ключевое слово для указания ограничения  
public class ValueList<T> where T : ...
```

| Ограничения | Что означает? |
|------------------------------------|--|
| <code>class</code> | <code>T</code> может быть любым ссылочным типом |
| <code>struct</code> | <code>T</code> - любой не nullable тип значения. Автоматически включает в себя ограничение <code>new()</code> и не может сочетаться с ним явно. Несовместимо с ограничением <code>unmanaged</code> |
| <code><ClassName></code> | <code>T</code> является типом <code>ClassName</code> или его наследником |
| <code><InterfaceName></code> | <code>T</code> обязан реализовывать интерфейс <code>InterfaceName</code> |
| <code>new()</code> | <code>T</code> должен иметь открытый конструктор без параметров. Не совместимо с ограничениями <code>struct</code> и <code>unmanaged</code> |
| <code>notnull</code> | <code>T</code> не должен допускать значения <code>null</code> |
| <code>T : U</code> | <code>T</code> совпадает с <code>U</code> или является его наследником. При использовании nullable контекста <code>T</code> должен быть не nullable, если <code>U</code> такой |
| <code>unmanaged</code> | <code>T</code> должен быть "неуправляемым" (хз че это). Автоматически включает в себя ограничения <code>struct</code> и <code>new()</code> |
| <code>default</code> | Только для обобщенных методов! Используется для устранения неоднозначности, связанной с переопределением методов без ограничений |



Ограничения `class`, `struct`, `unmanaged`, `notnull`, `default` несовместимы друг с другом!

Порядки ограничений:

1. Первичные: `class`, `struct`, `unmanaged`, `notnull`, `ClassName`, `default`
2. Вторичные: `InterfaceName`
3. Ограничения на конструктор: `new()`



Неверный порядок ограничений приводит к ошибке компиляции

Наследования обобщенных типов

Обобщенные типы могут наследоваться как от необобщенных типов, так и других открытых и закрытых обобщенных.

```
public class GenericBase<T> { }
public class NonGenericBase { }

// Открытый -> открытый.
public class GenericDerived1<U> : GenericBase<U> { }

// Открытый -> закрытый.
public class GenericDerived2<U> : GenericBase<int> { }

// Открытый -> необобщенный.
public class GenericDerived3<T> : NonGenericBase { }
```

Необобщенные типы могут наследоваться только от закрытых обобщенных.

```
public class GenericBase<T> { }
public class NonGenericBase { }

// Необобщенный -> закрытый.
public class NonGenericDerived : GenericBase<int> { }
```

Ковариантность и контрвариантность

Вариантность - свойство преобразования типов операторами.

Оператор - любая сущность языка, преобразующая тип данных в производный от него.

Соотнесение типов:

- $T > U$. Экземпляр U можно заменить экземпляром T
- $T < U$. Экземпляр T можно заменить экземпляром U

- `T = U`. Замены `T` на `U` и `U` на `T` равнозначны
- `T U`. Типы не сравнимы

Преобразование **ковариантно**, если оно сохраняет отношение между парой типов при преобразовании в производные

Преобразование **контрвариантно**, если оно заменяет *больше* на *меньше*, но сохраняет *не сравнимы* и *равны*

Для указания поддержки ковариантности классов и интерфейсов используется слово `out : out T`

Для указания поддержки контрвариантности классов и интерфейсов используется слово `in : in T`

Для указания поддержки инвариантности классов и интерфейсов никакое слово не указывается.

Обобщенные методы

Обобщенные методы могут быть объявлены в:

- Классах
- Структурах
- Интерфейсах

```
// Обобщенный метод выглядит так.
public class Swapper
{
    public void Swap<T>(ref T first, ref T second) { }
}
```

Итераторы

Итератор - объект, который выполняет обход контейнера (например, списка)

Итераторы можно использовать для:

- Выполнения определенного действия с каждым элементом коллекции
- Перечисления настраиваемой коллекции

- LINQ или других библиотек
- Создания конвейера данных, обеспечивающего эффективных их поток через методы итератора.

Для итерации коллекции нужен оператор `foreach`. Он создает код, опираясь на интерфейсы `IEnumerable<T>` и `IEnumerator<T>`

Соответственно итерируемая коллекция - коллекция, реализующая интерфейсы `IEnumerator` и `IEnumerable` или их обобщенные версии.



В методе итератора не могут одновременно присутствовать `yield`, `return` и `return`

Компилятор преобразует `foreach` в подобную конструкцию:

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

Интерфейс `IEnumerator` содержит в себе члены:

- `object Current { get; }`
- `bool MoveNext()`
- `void Reset()`

Интерфейс `IEnumerable` содержит:

- `IEnumerator GetEnumerator()`

yield

Не является служебным словом. Может использоваться только в двух формах:

- `yield return` - чтобы указать следующее значение в итерации
- `yield break` - для явного сигнала о завершении итерации

`yield` нельзя использовать в конструкциях:

- Методы с `in`, `ref`, `out`
 - Лямбда-выражения и анонимные методы
 - Методы, содержащие небезопасные блоки
-

LINQ

Анонимные типы

Анонимный тип - создаваемый компилятором ссылочный тип, инкапсулирующий свойства только для чтения.

```
// Массив объектов анонимного типа
var rectangles = new[]
{
    new { a = 3, b = 6 },
    new { a = 4, b = 1 }
};
```

LINQ

LINQ представляет простой и удобный язык запросов к источнику данных. Этот источник должен реализовывать интерфейс `IEnumerable<T>` или быть набором данных `DataSet`, или документом XML.

Существует несколько разновидностей LINQ:

- LINQ to Objects - для работы с массивами и коллекциями
- LINQ to Entities - при обращении к базам данных через Entity Framework
- LINQ to XML - при работе с XML
- LINQ to DataSet - при работе с DataSet
- PLINQ - для выполнения параллельных запросов

Будем говорить только о LINQ to Objects.

Служебные слова LINQ: `from`, `where`, `select`, `group`, `into`, `orderby`, `join`, `let`.

Контекстно-зависимые ключевые слова:

- `join`

- `in`
- `on`
- `equals`
- `group`
 - `by`
- `orderby`
 - `ascending`
 - `descending`

Для работы с коллекциями можно использовать 2 способа: оператор запросов LINQ и методы расширения LINQ.

```
// Операторы запросов.
string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };

var selectedPeople = from p in people
                      where p.ToUpper().StartsWith("T")
                      orderby p
                      select p;
```

```
// Методы расширения.
string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };

var selectedPeople = people.Where(p => p.ToUpper().StartsWith("T"));
```



Для реализации возможностей LINQ в C# используются лямбда-выражения, перечислимые коллекции, анонимные типы, методы расширения и деревья выражений

| Методы расширения | Что делает? |
|---|---|
| <code>Select(Func<TSource, TResult> selector)</code> | Определяет проекцию выбранных значений |
| <code>SelectMany(Func<TSource, IEnumerable<TResult>> selector)</code> | Делает то же самое, что и <code>Select</code> , но и позволяет свести набор коллекций в |

| Методы расширения | Что делает? |
|---|---|
| | одну |
| <code>Where<TSource>(Func<TSource, bool> predicate)</code> | Фильтр выборки |
| <code>OrderBy (Func<TSource, TKey> keySelector)</code> <code>OrderBy (Func<TSource, TKey> keySelector, IComparer<TKey>? comparer)</code> | Упорядочивает элементы по возрастанию |
| <code>OrderByDescending</code> | Упорядочивает элементы по убыванию |
| <code>ThenBy</code> | Задаёт дополнительные критерии для упорядочивания по возрастанию |
| <code>ThenByDescending</code> | Задаёт дополнительные критерии для упорядочивания по убыванию |
| <code>Join</code> | Соединяет 2 коллекции по определённому признаку |
| <code>Aggregate</code> | Применяет к элементам последовательности агрегатную функцию, которая сводит их к одному объекту |
| <code>GroupBy</code> | Группирует элементы по ключу |
| <code>ToLookup</code> | Группирует элементы по ключу и добавляет их в словарь |
| <code>GroupJoin</code> | Выполняет соединение коллекций и группировку элементов по ключу |
| <code>Reverse</code> | Располагает элементы в обратном порядке |
| <code>All</code> | Определяет, все ли элементы коллекции удовлетворяют какому-то условию |
| <code>Any</code> | Определяет, есть ли какой-то элемент коллекции, удовлетворяющий определённому условию |
| <code>Contains</code> | Определяет, содержит ли коллекция элемент |
| <code>Distinct</code> | Удаляет повторы |
| <code>Except</code> | Возвращает разность 2 коллекций |

| Методы расширения | Что делает? |
|------------------------------|---|
| <code>Union</code> | Возвращает объединение 2 коллекций |
| <code>Intersect</code> | Возвращает пересечение 2 коллекций |
| <code>Count</code> | Возвращает количество элементов коллекции |
| <code>Sum</code> | Подсчитывает сумму числовых значений коллекции |
| <code>Average</code> | Подсчитывает среднее значение числовых элементов коллекции |
| <code>Min</code> | Находит минимальное значение |
| <code>Max</code> | Находит максимальное значение |
| <code>Take</code> | Выбирает определенное число элементов |
| <code>Skip</code> | Пропускает определенное число элементов |
| <code>TakeWhile</code> | Выбирает цепочку элементов, пока выполняется условие |
| <code>SkipWhile</code> | Пропускает цепочку элементов, пока выполняется условие |
| <code>Concat</code> | Объединяет 2 коллекции |
| <code>Zip</code> | Объединяет 2 коллекции в соответствии с определенным условием |
| <code>First</code> | Выбирает первый элемент коллекции |
| <code>FirstOrDefault</code> | Выбирает первый элемент коллекции или возвращает значение по умолчанию |
| <code>Single</code> | Выбирает единственный элемент в коллекции. Если элементов больше 1 или их нет, то выбрасывается исключение |
| <code>SingleOrDefault</code> | Выбирает единственный элемент в коллекции. Если коллекция пуста, то берется значение по умолчанию. Если в коллекции больше 1 элемента, выбрасывается исключение |

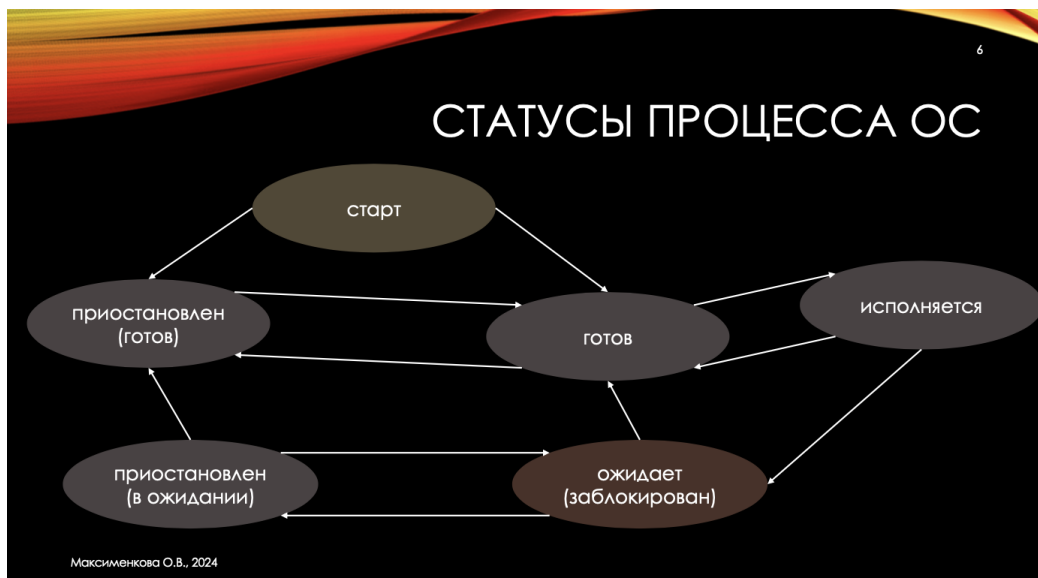
| Методы расширения | Что делает? |
|---------------------------------|--|
| <code>ElementAt</code> | Выбирает элемент последовательности по индексу |
| <code>ElementAtOrDefault</code> | Выбирает элемент последовательности по индексу или берет значение по умолчанию, если индекс некорректный |
| <code>Last</code> | Выбирает последний элемент коллекции |
| <code>LastOrDefault</code> | Выбирает последний элемент коллекции или возвращает значение по умолчанию |

Процессы и потоки

Параллельные вычисления - способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно)

Процесс - совокупность взаимосвязанных и взаимодействующих действий, преобразующих входные данные в выходные

Процесс - выполняющаяся программа и все ее ресурсы: адресное пространство, глобальные переменные, регистры, стек, открытые файлы и так далее



Параллельное взаимодействие осуществляется:

- Через разделяемую память
- С помощью передачи сообщений

Поток или **поток выполнения** - наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.



По умолчанию программа на C# запускается в одном потоке - **основном потоке**. В коде программы также могут быть созданы дополнительные потоки - **рабочие потоки**

Синхронность и асинхронность

Синхронное выполнение программы - выполнение в одном потоке

Асинхронное выполнение программы - выполнение в нескольких потоках

```
// Пример создания потока.
class Program
{
    static void Print()
    {
        for (int i = 0; i < 50; i++)
        {
            Console.Write(1);
        }
    }

    static void Main(string[] args)
    {
        Thread funcThread = new Thread(Print);
        funcThread.Start();

        for (int i = 0; i < 50; i++)
        {
            Console.Write(0);
        }
    }
}
```

```
}  
}
```

Методы взаимодействия потоков:

- Взаимоисключения (мьютексы)
- Семафоры
- Критические секции
- События



Для обеспечения эксклюзивного доступа к критической секции кода в одном процессе используются `lock` и `Monitor`

`Monitor` гарантирует снятие блокировки, если в критической секции появится исключение

Мьютекс - это особый вид двоичного семафора, который используется для обеспечения механизма блокировки.

Преимущества:

- Не возникает условий гонки, поскольку в критической секции в один момент времени находится только один процесс
- Обеспечивается целостность данных
- Это простой механизм блокировки, который активируется процессом перед входом в критическую секцию и освобождается при выходе из нее

Недостатки:

- Если после входа в критическую секцию поток заснет или будет вытеснен высокоприоритетным процессом, ни один другой поток не сможет войти в критическую секцию
- Когда предыдущий поток покидает критическую секцию, в нее могут войти только другие процессы
- Реализация мьютекса может привести к занятому ожиданию, что приводит к трате процессорного времени

Mutex - частный случай семафора, применяющийся для ограничения доступа к ресурсу, при этом освободить ресурс может только занявший его поток.

```
// Шаблон использования Mutex.  
private static Mutex mtx = new Mutex();  
  
public static void Method()  
{  
    mtx.WaitOne();  
    try  
    {  
        // Критическая секция.  
    }  
    finally  
    {  
        // Освобождаем блокировку.  
        mtx.ReleaseMutex();  
    }  
}
```

Семафор - обычная целочисленная неотрицательная переменная, с которой можно работать только с помощью двух специальных неделимых (атомарных) операций: *ожидания* (Wait, P) и *сигнала* (Signal, V)

Преимущества:

- Несколько потоков могут получить доступ к критической секции одновременно
- Одновременно к критической секции будет обращаться один процесс, однако будет допускаться работа нескольких потоков
- Семафоры являются машинно-независимыми, поэтому их не

Недостатки:

- Содержит инверсию приоритета
- Операции семафора должны быть реализованы корректно, чтобы избежать дедлоков, что приводит к потере модульности, поэтому семафоры нельзя использовать в крупномасштабных системах
- Семафор чувствителен к ошибкам при

следует запускать через
микроядро

- Гибкое управление ресурсами

программировании, что может
провоцировать дедлоки и
нарушение свойства взаимного
исключения

- ОС должна отслеживать все
вызовы операций ожидания
сигналов

`Semaphore` служит для ограничения количества потоков, получающих доступ к ресурсу. Подходит для контроля пулов ресурсов, позволяет получить именованный системный семафор.

```
// Шаблон использования класса Semaphore.  
private static Semaphore smph = new Semaphore(2, 2);  
  
public static void Method()  
{  
    smph.WaitOne();  
    try  
    {  
        // Критическая секция.  
    }  
    finally  
    {  
        // Освобождаем блокировку.  
        smph.ReleaseMutex();  
    }  
}
```

Для управления потоками с помощью событий используют `AutoResetEvent` и `ManualResetEvent`

Пул потоков - специальный набор рабочих потоков приложения, управляемых системой

С помощью `Thread.Sleep(int time)` можно создавать задержки в потоках.

Синхронизировать потоки можно с помощью нестатического метода `Join()`. Он ожидает пока рабочий поток выполнится и только потом передает работу дальше.

| | |
|-----------------------------------|--|
| <code>Thread.CurrentThread</code> | Статическое свойство, возвращающее Thread |
| <code>IsAlive</code> | Свойство, возвращающее значение <code>bool</code> в зависимости от того, запущен поток или нет |
| <code>Name</code> | Свойство, хранящее в себя имя потока (<code>string</code>) |
| <code>Start()</code> | Запуск потока |
| <code>Start(аргумент)</code> | Запуск потока с передачей в него аргумента |
| <code>Join()</code> | Синхронизация потоков |
| <code>Join(time)</code> | Блокирует другие потоки, пока данный не выполнится или не пройдет время |
| <code>Sleep(int time)</code> | Приостанавливает поток на <code>time</code> мс |
| <code>Thread.Sleep(0)</code> | Возможность переключиться на другой поток |

Асинхронность

Асинхронное программирование - параллельная модель программирования, в которой допускается выполнять больше количество задач одновременно в небольшом количестве потоков и при поддержке синтаксиса обычного синхронного кода

Асинхронное программирование основывается на **шаблоне на основе задач (TAP)**

Задача (Task) - конструкция, реализующая модель асинхронной обработки на основе обещаний (модель "обещает", что задача будет выполнена позже, позволяя взаимодействовать с помощью "обещаний" с чистым API)

Класс `Task` представляет одну задачу, которая не возвращает значение.

Класс

`Task<T>` представляет одну задачу, которая возвращает значение типа `T`.



По умолчанию задачи выполняются в текущем потоке и при необходимости делегируют работу операционной системе. Также для задач можно явно запрашивать запуск в отдельном потоке через метод `Task.Run()`