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

# ===11. Класс. Элементы класса. Свойства и индексаторы.

**Класс** – это некоторое абстрактное понятие-шаблон, по которому определяется форма объекта.

Описанием объекта является класс, а объект представляет экземпляр этого класса. **Объект** – это физическая реализация класса(шаблона).

**Класс:**

**Данные члены:**

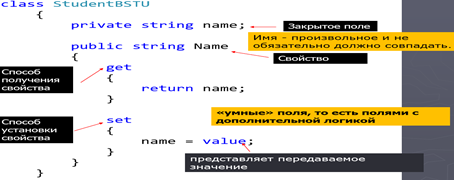
* Поля (любые переменные, ассоциированные с классом
* Константы (могут быть ассоциированы с классом тем же способом, что и переменные. Константа объявляется с помощью ключевого слова **const.** Если она объявлена как public, то в этом случае становится доступной извне класса.)
* События (определяют уведомления, которые может генерировать класс)

**Функции-члены:**

* Методы (реализуют вычисления или другие действия, выполняемые классом или экземпляром)
* Свойства (определяют методы записи и чтения)
* Конструкторы (вызываются при создании нового объекта данного класса, выполняют инициализацию объекта. Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор без параметров)
* Финализаторы
* Операции (задают действия с объектами с помощью знаков операций)
* Индексаторы (обеспечивают воз-сть доступа к элементам класса по их порядковому номеру)

**Свойства класса**

**Свойства** – специальные методы класса, служат для организации доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки (предоставляет инкапсуляцию).



**Назначение** - свойства позволяют вложить дополнительную логику

**Ограничения свойств**:

1) не может быть передано методу в качестве параметра ref или out.

2) не подлежит перегрузке

3) не должно изменять состояние базовой переменной при вызове аксессора get

4) могут быть статическими, экземплярными, абстрактными и виртуальными

5) могут иметь модификатор доступа 6) могут определяться в интерфейсах

**Автоматические свойства:** тип имя { get; set; }. Компилятор автоматически реализует методы для правильного возвращения значения из поля и назначения значения полю

**Проблемы:**

• неявная инициализация

• проблемы при сериализации и десериализации

• во время отладки нельзя установить точку останова

**Индексаторы** (свойства с параметрами)

►Позволяют индексировать объекты таким же способом, как массив или коллекцию

►«умный» индекс для объектов

►средство, позволяющее разработчику перегружать оператор []

Атрибуты спецификаторы тип this [ список\_параметров ] get код доступа set код доступа

**Ограничения на индексаторы:**

1) значение, выдаваемое индексатором, нельзя передавать методу в качестве параметра ref или out

2) индексатор не может быть объявлен как static.

Можно перегружать

# ===12. Класс. Константы. Поля только для чтения. Инициализаторы класса.

**Класс** – это некоторое абстрактное понятие-шаблон, по которому определяется форма объекта.

**Объект** – это физическая реализация класса(шаблона).

**Константы.**

const int CC =100; // значение не может изменено

1) компилятор сохраняет значение константы в метаданных модуля константы можно определять только для таких типов, которые компилятор считает примитивными (или не примитивный но тогда = null)

2) константы считаются не явно статическими, всегда связаны с типом, а не с экземпляром тип

3) нельзя получать адрес константы и передавать ее по ссылке

4) определять можем один раз

5) к моменту компиляции они должны быть определены.

**Поля для чтения readonly**

1) Запись в поле разрешается при объявлении или в коде конструктора

2) Инициализировать или изменять их значение в других местах нельзя, можно только считывать их значение.

public readonly double K = 23;

**Инициализаторы.**

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

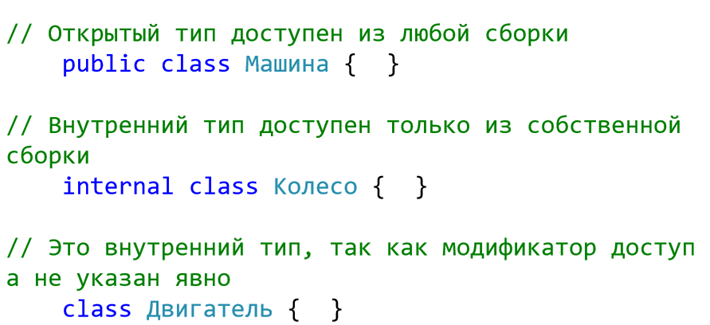
Person tom = new Person { name = "Tom", age=31 };

tom.GetInfo(); // Имя: Tom Возраст: 31

# ===13. Спецификаторы доступа C#. Видимость типов. Доступ к членам типов.

Модификаторы доступа позволяют задать допустимую область видимости для членов класса. То есть контекст, в котором можно употреблять данную переменную или метод.

**Видимость типа** может быть открытой (public) или внутренней (internal).



**Доступ к членам типов**

►**public** - доступ не ограничен – все члены во всех сборках

►**private** - по умолчанию для членов класса (используется для вложенных классов). Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.

►**protected** - (используется для вложенных классов) такой член класса доступен из любого места в текущем классе или в производных классах.

► **internal** - класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок.

► **protected internal -** совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.

Объявление полей класса без модификатора доступа равнозначно их объявлению с модификатором private. Классы, объявленные без модификатора, по умолчанию имеют доступ internal.

Существуют различные **контексты:**

* **Контекст класса.** Переменные, определенные на уровне класса, доступны в любом методе этого класса
* **Контекст метода.** Переменные, определенные на уровне метода, являются локальными и доступны только в рамках данного метода. В других методах они недоступны
* **Контекст блока кода.** Переменные, определенные на уровне блока кода, также являются локальными и доступны только в рамках данного блока. Вне своего блока кода они не доступны.

# ===14. Класс. Конструкторы и их свойства. Деструкторы

**Класс** – это некоторое абстрактное понятие - шаблон, по которому определяется форма объекта.

**Объект** – это физическая реализация класса (шаблона).

**Конструкторы** — это специальные методы, позволяющие корректно инициализировать новый экземпляр типа.

Есть конструкторы по умолчанию и с аргументами. Если нет конструктора, то компилятор создает конструктор по умолчанию, который инициализирует все поля класса значениями по умолчанию (0, false, null).

Свойства конструкторов:

* не может возвращать значение
* имя равно имени класса
* не наследуются
* нельзя применять ключевые слова virtual, new, override, sealed, abstract
* для класса без явно заданных конструкторов компилятор создает конструктор по умолчанию (без параметров)
* для статических классов (запечатанных и абстрактных) компилятор не создает конструктор по умолчанию
* может быть несколько конструкторов, сигнатуры и уровни доступа к конструкторам обязательно должны отличаться
* можно явно заставлять один конструктор вызывать другой конструктор посредством зарезервированного слова this: обеспечивает доступ к текущему экземпляру класса

**Деструкторы** вызываются непосредственно перед окончательным уничтожением объекта системой "сборки мусора", чтобы гарантировать четкое окончание срока действия объекта. Если программа завершиться до того, как произойдет "сборка мусора", деструктор может быть вообще не вызван.

**СИНТАКСИС: ~имя\_класса() { // код деструктора }**

Свойства деструкторов:

* Класс может иметь только один деструктор
* Деструкторы не могут быть унаследованы или перегружены
* Деструкторы невозможно вызвать, они запускаются автоматически
* Деструктор не принимает модификаторы и не имеет параметров

# ===15. Класс и методы System.Object.

**Класс** – это некоторое абстрактное понятие - шаблон, по которому определяется форма объекта.

**Объект** – это физическая реализация класса (шаблона).

**System.Object** - класс .Net, от которого наследуются все классы, объекты, типы значений. Переменная ссылочного типа object может ссылаться на объект любого другого типа.

Методы System.Object:

* Открытые:
  + **ToString()** – получение строкового представления данного объекта.
    - для базовых типов выводится их строковое значение
    - для классов - выводит полное название класса с указанием пространства имен, в котором определен этот класс: Console.WriteLine(Olga.ToString());
  + **GetHashCode()** – возвращает хэш-код для значения данного объекта. Позволяет возвратить некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код.
    - Максимально быстрый
    - Объекты с одинаковым значением должны возвращать одинаковые коды
  + **Equals()** – сравнивает 2 объекта на равенство.
  + **GetType()** – возвращает тип данного объекта
* Закрытые:
  + **Finalize()** – деструктор - вызывается при сборке мусора для очистки ресурсов, занятых ссылочным объектом.
  + **Clone()** – создает копию объекта и возвращает ссылку на эту копию (в случае типа значение – ссылку на упаковку).
    - Копируются все типы значений в классе.
    - Если класс включает в себя члены ссылочных типов, то копируются только ссылки, а не объекты, на которые они указывают.
    - Не виртуальный, переопределять его реализацию нельзя

# ===16. Статические методы и статические конструкторы класса.

Переменные и свойства, которые хранят состояние, общее для всех объектов класса, следует определять как статические.

Методы, которые определяют общее для всех объектов поведение, также следует объявлять как статические.

public class StudentBSTU {

private static string uo;

public static string UO { get; set; } = "BSTU";

public static void getUo (){ Console.WriteLine(UO); } }

Статические члены класса являются общими для всех объектов этого класса, поэтому к ним надо обращаться по имени класса. При использовании статических членов необязательно создавать экземпляр класса:

StudentBSTU.getUo();

Console.WriteLine(StudentBSTU.UO);

Для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса.

**Свойства статических методов:**

► отсутствует ссылка this, поскольку такой метод не выполняется относительно какого-либо объекта

► в методе static допускается непосредственный вызов только других методов типа static

► для метода static непосредственно доступными оказываются только другие данные типа static, определенные в его классе

**Статические конструкторы или конструкторы типа.**

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы выполняются при самом первом создании объекта данного класса или первом обращении к его статическим членам (если таковые имеются).

Конструктор экземпляра инициализирует данные экземпляра.

Конструктор класса (типа)— данные класса.

**Свойства:**

► закрытые автоматически

► не имеет параметров

► как и в статических методах, в статических конструкторах нельзя использовать ключевое слово this для ссылки на текущий объект класса и можно обращаться только к статическим членам класса

► нельзя вызвать явным образом (вызываются до создания первого экземпляра объекта или до вызова любого статического метода).

class D

{

private D() { } // закрытый конструктор

static D(){ \_a = 200; }// статический конструктор

static int \_a;

}

# ===17. Статические классы. Методы расширения и правила их определения.

Статические классы объявляются с модификатором static и могут содержать только статические поля, свойства и методы.

static class имя класса { // ...

► прямой потомок System.Object

► экземпляры такого класса создавать запрещено

►не должен реализовывать никаких интерфейсов (не вызвать)

►нельзя использовать в качестве поля, параметра метода или локальной переменной

►от него запрещено наследовать

►все элементы такого класса должны явным образом объявляться с модификатором static

►может иметь статический конструктор

►Компилятор не создает автоматически конструктор по умолчанию

Статические классы применяются главным образом в двух случаях. Во-первых, статический класс требуется при создании метода расширения. Методы расширения связаны в основном с языком LINQ. И во-вторых, статический класс служит для хранения совокупности связанных друг с другом статических методов:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

**Методы расширения** (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса. Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем.

public static class NewFromAlex {

public static bool isLetter(this String st, char a) {

for (Int32 index = 0; index < st.Length; index++)

if (st[index] == a) return true;

return false; } }

Для того, чтобы создать метод расширения, вначале надо создать статический класс, который и будет содержать этот метод. Затем объявляем статический метод. Собственно метод расширения - это обычный статический метод, который в качестве первого параметра всегда принимает такую конструкцию: this имя\_типа название\_параметра. Затем у всех строк мы можем вызвать данный метод.

1) Проверяется класс и его базовые

2) Ищется любой статический класс с методом ####, у которого первый параметр соответствует типу выражения ( this)

**Правила для методов расширений**

►1) Методы расширения должны быть объявлены в статическом необобщенном классе (первого уровня)

►2) this перед первым аргументом и только один

►3) использовать аккуратно

# ===18. Анонимные типы.

Позволяют создать объект с некоторым набором свойств без определения класса (тип в одном контексте или один раз). Анонимный тип определяется с помощью ключевого слова **var и инициализатора объектов**:

1. var user = new { Name = "Tom", Age = 34 };
2. Console.WriteLine(user.Name);

В данном случае user - это объект анонимного типа, у которого определены два свойства Name и Age. И мы также можем использовать его свойства, как и у обычных объектов классов. Однако тут есть ограничение - **свойства анонимных типов доступны только для чтения.**

При этом во время компиляции компилятор сам будет создавать для него имя типа и использовать это имя при обращении к объекту.

Для исполняющей среды CLR анонимные типы будут также, как и классы, представлять ссылочный тип.

Если в программе используются несколько объектов анонимных типов с одинаковым набором свойств, то для них компилятор создаст одно определение анонимного типа.

Зачем нужны анонимные типы? Иногда возникает задача использовать один тип в одном узком контексте или даже один раз. Создание класса для подобного типа может быть избыточным. Если нам захочется добавить свойство, то мы сразу же на месте анонимного объекта это можем сделать. В случае с классом придется изменять еще и класс, который может больше нигде не использоваться. Типичная ситуация - получение результата выборки из базы данных: объекты используются только для получения выборки, часто больше нигде не используются, и классы для них создавать было бы излишне. А вот анонимный объект прекрасно подходит для временного хранения выборки.

# ===19. Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы.

**Параметры** представляют собой переменные, которые определяются в сигнатуре метода и создаются при его вызове.

Существует **два способа передачи параметров** в метод в языке C#: по значению и по ссылке.

Для обмена данными между вызывающей и вызываемой функциями предусмотрено четыре типа параметров:

►По умолчанию- параметры-значения;

► параметры-ссылки — ref;

► выходные параметры-ссылки — out:

► переменное количество — params (один и последний).

**Назначение:**

► позволить методу менять содержимое его аргументов

► возвращать более одного значения

**ref** заставляет С# организовать вместо вызова по значению вызов по ссылке.

При передаче по значению метод получает не саму переменную, а ее копию. А при передаче параметра по ссылке метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место.

Аргументу, передаваемому методу "в сопровождении" модификатора ref, должно быть присвоено значение до вызова метода.

Модификатор **out** подобен модификатору ref за одним исключением: его можно использовать для передачи значения из метода.

out-параметр "поступает" в метод без начального значения, но метод (до своего завершения) обязательно должен присвоить этому параметру значение.

По сути, как и в случае с ключевым словом ref, ключевое слово out применяется для для передачи аргументов по ссылке. Однако в отличие от ref для переменных, которые передаются с ключевым словам out, не требуется инициализация. И кроме того, вызываемый метод должен обязательно присвоить им значение.

**params** позволяет передавать методу переменное количество аргументов одного типа.

static void Addition(params int[] integers)

{

int result = 0;

for (int i = 0; i < integers.Length; i++)

{

result += integers[i];

}

Console.WriteLine(result);

}

**Необязательные аргументы**

►позволяет определить используемое по умолчанию значение для параметра метода

►можно применять в конструкторах, индексаторах

static int OptionalParam(int x, int y, int z=5, int s=4)

**Именованные аргументы**

Позволяет указать имя того параметра, которому присваивается его значение ( в конструкторах, индексаторах или делегатах.) И в этом случае порядок следования аргументов уже не имеет никакого значения.

// Использование именованных аргументов

// при вызове метода

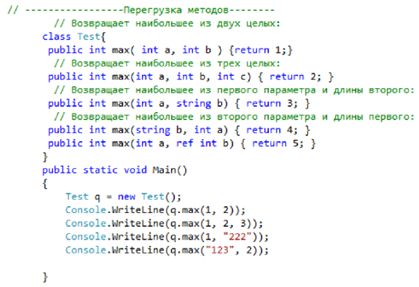
int sum1 = mySum(a: 3, b: 10);

# ===20. Перегрузка методов и операторов. Правила перегрузки операторов.

**Перегрузка методов**

* один и тот же метод, но с разным набором параметров
* позволяет обращаться к связанным методам посредством одного, общего для всех имени.
* никакие два метода внутри одного и того же класса не должны иметь одинаковую сигнатуру сигнатура (signature) = имя метода + список его параметров (не включает тип значения, возвращаемого методом, не включает params параметр)

Разные версии метода могут иметь разные возвращаемые значения.



**Перегрузка операций**

Перегрузка операторов заключается в определении в классе, для объектов которого мы хотим определить оператор, специального метода:

public static возвращаемый\_тип operator оператор(параметры)

{ }

►способ объявления новых операций для типа

Спецификация CLR требует, чтобы перегруженные операторные методы были:

1) открытыми и статическими

2) тип одного из параметров или возвращаемого значения совпадал с типом, в котором определен операторный метод

**Операции подлежащие перегрузке**

* +, -, !, ++, --
* true, false (попарно)
* +, -, \*, /, %, &, |, ^, <<, >>
* ==, !=, <, >, <=, >= (перегрузка парами)

**Операции не подлежащие перегрузке:**

* [] (но есть индексатор)
* () (можно определить новые операторы преобразования)
* +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>= (но получаем автоматически в случае перегрузки бинарной операции)
* &&, ||
* =, ., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof

**Правила:**

* префиксные операции ++ и – – перегружаются парами;
* операции сравнения перегружаются парами: == и != ; < и >;<= и >=.
* Перегруженные операции обязаны возвращать значения
* Должны объявляться как public и static
* префиксная и постфиксная формы операций ++ и --, в отличие от оригинальных операций, семантически НЕ различаются.

Если перегружаются операторы == и !=, то для этого требуется переопределить методы Object.Equals() и Object.GetHashCode().