**ООП -** методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Принципы объектно-ориентированного программирования:

**Инкапсуляция (Encapsulation);**

**Наследование (Inheritance);**

**Полиморфизм (Polymorphism);**

**Абстракция данных (Abstraction);**

**Инкапсуляция -** механизм, связывающий вместе данные и код, обрабатывающий эти данные, и сохраняющий их от внешнего воздействия и ошибочного использования

**Инкапсуляция** — размещение одного объекта или класса внутри другого для разграничения доступа к ним.

**Свойства инкапсуляции:**

Совместное хранение данных

Сокрытие внутренней информации от пользователя

**Полиморфизм** — реализация задач одной и той же идеи разными способами;

**Полиморфизм** - способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта.

**Наследование** - процесс, благодаря которому один объект может наследовать (приобретать) свойства от другого объекта.

**Абстракция** подразумевает разделение и независимое рассмотрение интерфейса и реализации.

**Абстракция** - уровень описания/представления модели чего-либо.

**Класс** – пользовательский тип данных, ссылочный, абстрактный.

**Класс** – это собственный тип данных.

**Класс –** шаблон, по которому определяется форма объекта.

**Объект –** это физическая реализация класса.

**class** Student

{

    public int x; //переменные x, y – поля класса

public int y; //public – модификатор, который позволяет дать доступ полям вне класса Student.

}

**Экземпляр класса:**

**Student NameStudent = new Student ();**

**Модификаторы доступа:**

- **public** – общедоступный член класса.

- **private** – член класса доступен только внутри данного класса.

- **protected** – член класса доступен только внутри данного класса и внутри классов, производных от данного.

- **internal** – член класса доступен только внутри данной сборки (программы).

**const** – используется для объявления постоянного поля. Значение такого поля нельзя изменять; **readonly** – используется для объявления поля, которое предназначено только для чтения.

Первая разница между ними состоит в том, что readonly полям можно присваивать значения в двух местах: в месте их создания и в конструкторах. Для констант же значения можно присвоить только в месте их создания.

**Индексаторы** позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами

возвращаемый\_тип this [Тип параметр1, ...]

{

    get { ... }

    set { ... }

}

**Конструктор класса** – это специальный метод класса, который вызывается при создании объекта такого класса, (т.е там где мы используем ключевое слово new)

Конструктор по умолчанию, конструктор с параметрами, закрытый конструктор

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

1. НЕ ИМЕЕТ ВОЗВРАЩАЕМОГО ЗНАЧЕНИЯ.

2. ИМЯ ТАКОЕ ЖЕ КАК И ИМЯ ТИПА(КЛАССА).

3. НЕ НАСЛЕДУЕТСЯ

4. НЕЛЬЗЯ ПРИМЕНЯТЬ МОДИФИКАТОРЫ VIRTUAL,NEW,OVERRIDE,SEALED И ABSTRACT.

5. ДЛЯ КЛАССА БЕЗ ЯВНО ЗАДАННЫХ КОНСТРУКТОРОВ КОМПИЛЯТОР СОЗДАЕТ КОНСТРУКТОР ПО УМОЛЧАНИЮ(БЕЗ ПАРАМЕТРОВ).

6. ДЛЯ СТАТИЧЕСКИХ КЛАССОВ(ЗАПЕЧАТАННЫХ И АБСТРАКТНЫХ)КОМПИЛЯТОР НЕ СОЗДАЕТ КОНСТРУКТОР ПО УМОЛЧАНИЮ.

7. МОЖЕТ ОПРЕДЕЛЯТЬСЯ НЕСКОЛЬКО КОНСТРУКТОРОВ,СИГНАТУРЫ И УРОВНИ ДОСТУПА К КОНСТРУКТОРАМ ОБЯЗАТЕЛЬНО ДОЛЖНЫ ОТЛИЧАТЬСЯ.

8. МОЖНО ЯВНО ЗАСТАВЛЯТЬ ОДИН КОНСТРУКТОР ВЫЗЫВАТЬ ДРУГОЙ КОНСТРУКТОР ПОСРЕДСТВОМ ЗАРЕЗЕРВИРОВАННОГО СЛОВА this.

Деструктор – вызывается непосредственно перед окончательным уничтожением объекта системой “сборки мусора”, чтобы гарантировать четкое окончание срока действия объекта

**Статический конструктор** – это специальный метод, который называется также как и наш класс и у которого модификатор static. В любом классе может быть только один статический конструктор (т.е мы не можем использовать перегрузку), статический конструктор не может принимать параметров. Статический конструктор выполняется только одни раз.

Статический конструктор вызывается автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам (если таковые имеются).

**ToString**

Метод ToString служит для получения строкового представления данного объекта. Для базовых типов просто будет выводиться их строковое значение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int i = 5;  Console.WriteLine(i.ToString()); // выведет число 5    double d = 3.5;  Console.WriteLine(d.ToString()); // выведет число 3,5 |

**Метод GetHashCode**

Метод GetHashCode позволяет возвратить некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты. Можно определять самые разные алгоритмы генерации подобного числа или взять реализаци базового типа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Person  {      public string Name { get; set; }        public override int GetHashCode()      {          return Name.GetHashCode();      }  } |

В данном случае метод GetHashCode возвращает то хеш-код для значения свойства Name. То есть два объекта Person, которые имеют одно и то же имя, будут возвращать один и тот же хеш-код. Однако в реальности алгоритм может быть самым различным.

**Получение типа объекта и метод GetType**

Метод GetType позволяет получить тип данного объекта:

|  |  |
| --- | --- |
| 1  2 | Person person = new Person { Name = "Tom" };  Console.WriteLine(person.GetType());    // Person |

Этот метод возвращает объект Type, то есть тип объекта.

С помощью ключевого слова typeof мы получаем тип класса и сравниваем его с типом объекта. И если этот объект представляет тип Client, то выполняем определенные действия.

|  |  |
| --- | --- |
| 1  2  3 | object person = new Person { Name = "Tom" };  if (person.GetType() == typeof(Person))      Console.WriteLine("Это реально класс Person"); |

Причем поскольку класс Object является базовым типом для всех классов, то мы можем переменной типа object присвоить объект любого типа. Однако для этого переменной метод GetType все равно вернет тот тип, на объект которого ссылается переменная. То есть в данном случае объект типа Person.

В отличие от методов ToString, Equals, GetHashCode метод GetType не переопределяется.

**Метод Equals**

Метод Equals позволяет сравнить два объекта на равенство:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Person  {      public string Name { get; set; }      public override bool Equals(object obj)      {          if (obj.GetType() != this.GetType()) return false;            Person person = (Person)obj;          return (this.Name == person.Name);      }  } |

Метод Equals принимает в качестве параметр объект любого типа, который мы затем приводим к текущему, если они являются объектами одного класса. Затем сравниваем по именам. Если имена равны, возвращаем true, что будет говорить, что объекты равны. Однако при необходимости реализацию метода можно сделать более сложной, например, сравнивать по нескольким свойствам при их наличии.

Применение метода:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Person person1 = new Person { Name = "Tom" };  Person person2 = new Person { Name = "Bob" };  Person person3 = new Person { Name = "Tom" };  bool p1Ep2 = person1.Equals(person2);   // false  bool p1Ep3 = person1.Equals(person3);   // true |

И если следует сравнивать два сложных объекта, как в данном случае, то лучше использовать метод Equals, а не стандартную операцию ==.

Ключевые слова **Ref** и **out** в C# используются для передачи аргументов внутри метода или функции. Оба слова указывают на то, что аргумент/параметр передается по ссылке. По умолчанию параметры передаются в метод по значению. Используя эти ключевые слова (ref и out), мы можем передать параметр по ссылке.

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

**Статические поля**: Если какому-нибудь полю мы добавляем модификатор static, память которая будет выделяться под какое-то поле будет доступно для всех объектов класса (статическая переменная общая для всех экземпляров класса). Чтобы работать со static переменной не нужно создавать экземпляр класса. Чтобы получить доступ переменной нужно обратиться к самому классу, а не к экземпляру этого класса.

**Статические классы**

Класс можно объявлять как static. Статический класс обладает двумя основными свойствами. Во-первых, объекты статического класса создавать **нельзя**. И во-вторых, статический класс должен содержать только статические члены. Статический класс создается по приведенной ниже форме объявления класса, видоизмененной с помощью ключевого слова static.

*static class имя класса { // ...*

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

Методы расширения позволяют добавлять новые методы в уже существующие типы без создания нового производного класса.

**Перегрузка методов** – это наличие разных реализаций данного метода, которые могут работать по-разному. Когда мы говорим о перегрузке методов – это означает, что у нас есть несколько реализаций одного и того же метода, но с разной сигнатурой, т.е имя метода (writeline у нас одно и тоже, но количество принимаемых параметров (типы данных этих параметров, тип возвращаемого значения может отличатся).

Для того чтобы иметь возможность **переопределить метод**, который находится в базовом классе у этого метода должен быть модификатор virtual (виртуальные методы). Для того чтобы в классе наследнике сделать эту самую новую реализацию переопределить метод базового класса нам нужно использовать ключевое слово override.

Анонимные типы позволяют создать объект с некоторым набором свойств без определения класса. Анонимный тип определяется с помощью ключевого слова **var** и инициализатора объектов:

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