# Классы и объекты

Учебные вопросы

1. Определение класса. Поля. Методы. Объекты.
2. Конструкторы. Деструктор.
3. Ключевое слово this. Ключевое слово static.
4. Инкапсуляция. Спецификаторы доступа. Свойства.
5. Перегрузка операций.

### Определение класса. Поля. Методы. Объекты.

Класс является обобщенным понятием, определяющим характеристики и поведение некоторого множества конкретных объектов этого класса, называемых *экземплярами класса.* В общем случае класс содержит *данные,* задающие свойства объектов класса, и *функции,* определяющие их поведение. В класс также могут добавляться *события,* на которые может реагировать объект класса (приложения, построенные на основе событийно-управляемой модели, например, при программировании для Windows.).

Примеры: Студенты, Преподаватели, Аудитории, Дисциплины…

**Пространство имен.**

Все классы библиотеки .NET, а также все классы, которые создает программист в среде .NET, имеют одного общего предка — класс object и организованы в единую иерархическую структуру. Внутри нее классы логически сгруппированы в так называемые пространства имен, которые служат для упорядочивания имен классов и предотвращения конфликтов имен: в разных пространствах имена могут совпадать. Пространства имен могут быть вложенными, их идея аналогична иерархической структуре каталогов на компьютере.

Любая программа, создаваемая в .NET, использует пространство имен System. В нем определены классы, которые обеспечивают базовую функциональность, например, поддерживают выполнение математических операций, управление памятью и ввод-вывод.

Обычно в одно пространство имен объединяют взаимосвязанные классы.

**Описание класса.**

[ атрибуты ] [ спецификаторы ] class имя\_класса [ : предки ] тело-класса

Рисунок 1 Состав класса

* **Константы** класса хранят неизменяемые значения, связанные с классом.
* **Поля** содержат данные класса.
* **Методы** реализуют вычисления или другие действия, выполняемые классом или экземпляром.
* **Свойства** определяют характеристики класса в совокупности со способами их задания и получения, то есть методами записи и чтения.
* **Конструкторы** реализуют действия по инициализации экземпляров или класса в целом.
* **Деструкторы** определяют действия, которые необходимо выполнить до того, как объект будет уничтожен.
* **Индексаторы** обеспечивают возможность доступа к элементам класса по их порядковому номеру.
* **Операции** задают действия с объектами с помощью знаков операций.
* **События** определяют уведомления другим классам, которые может генерировать класс.
* **Локальные типы** – это типы данных, внутренние по отношению к классу.

**Данные**, содержащиеся в классе, могут быть переменными (поля) или константами*.*

[ атрибуты ] [ спецификаторы ] [ const ] тип имя [ = начальное\_значение ]

Спецификаторы полей и констант класса.

* new - новое описание поля, скрывающее унаследованный элемент класса.
* **public** - доступ к элементу не ограничен.
* **protected** - доступ только из данного и производных классов и из данной программы (сборки).
* **private** - доступ только из данного класса.
* internal - доступ только из данной программы (сборки).
* **static** - одно поле для всех экземпляров класса
* readonly - поле доступно только для чтения

По умолчанию элементы класса считаются закрытыми (private). Все методы класса имеют непосредственный доступ к его закрытым полям.

**Метод** *—* это функциональный элемент класса, который реализует вычисления или другие действия, выполняемые классом и л и экземпляром. Методы определяют поведение класса.

Метод представляет собой законченный фрагмент кода, к которому можно обратиться по имени. Он описывается один раз, а вызываться может столько раз, сколько необходимо. Один и тот же метод может обрабатывать различные данные, переданные ему в качестве аргументов.

**Синтаксис метода:**

[ атрибуты ] [ спецификаторы ] тип имя\_метода ( [ параметры ] ) тело метода

Чаше всего для методов задается спецификатор доступа public, ведь методы составляют интерфейс класса - то, с чем работает пользователь, поэтому они должны быть доступны. Статические (static) методы, или методы класса, можно вызывать, не создавая экземпляр объекта. Именно таким образом используется метод Main.

**Пример 1.**

//простейшее описание класса

class Person

{

public string name;

public int age;

public void Show(){

Console.WriteLine(name + ", " + age);

}

}

Для каждого объекта при его создании в памяти выделяется отдельная область, в которой хранятся его данные. Объекты создаются явным или неявным образом, то есть либо программистом, либо системой. Программист создает экземпляр класса с помощью операции new.

//создание объекта класса

Person p=new Person();

p.name=“Кузнецов”;

**Пример 2.**

class Car

{

public string name;

public int speed;

public void Show()

{

Console.WriteLine("{0}, скорость={1}", name, speed);

}

public void SpeedUp(int delta)

{

speed += delta;

}

}

static void Main(string[] args)

{

// Разместить в памяти и сконфигурировать объект Саг.

Car myCar = new Car(); //ссылочный тип

myCar.name = "Ford Focus";

myCar.speed = 10;

// Повысить скорость автомобиля в несколько раз и вывести новое состояние.

for (int i = 0; i <= 10; i++)

{

myCar.SpeedUp(5);

myCar.Show();

}

}

### Конструкторы. Деструктор.

**Конструктор** предназначен для инициализации объекта. Он вызывается автоматически при создании объекта класса с помощью операции new. Имя конструктора совпадает с именем класса.

**Свойства конструкторов**:

1. Конструктор не возвращает значение,даже типа void.
2. Класс может иметь несколько конструкторовс разными параметрами для разных видов инициализации.
3. Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, полям значимых типов присваивается нуль, полям ссылочных типов – значение null .

Конструктор, вызываемый без параметров, называется конструктором по умолчанию.

class Car

{

public string name;

public int speed;

public Car()

{

name = "Opel Astra";

speed = 0;

}

. . . . . . .

}

Часто бывает удобно задать в классе несколько конструкторов,чтобы обеспечить возможность инициализации объектов разными способами.

//конструктор с параметрами

class Car

{

public string name;

public int speed;

public Car()

{

name = "Opel Astra";

speed = 0;

}

**public Car(string carName)**

**{**

**name = carName;**

**}**

**public Car(string carName,**

**int carSpeed)**

**{**

**name = carName;**

**speed = carSpeed;**

**}**

. . . . . .

}

//использование конструкторов

Car first = new Car();

Car second = new Car("Opel Corsa");

Car third = new Car("Toyota Corolla", 60);

first.Show();

second.Show();

third.Show();

### 3. Деструкторы. Сбор мусора.

При использовании оператора new объектам динамически выделяется память из пула свободной памяти. Безусловно, объем буфера динамически выделяемой памяти не бесконечен, и рано или поздно свободная память может исчерпаться. Поэтому одним из ключевых компонентов схемы динамического выделения памяти является восстановление свободной памяти от неиспользуемых объектов, что позволяет делать ее доступной для создания последующих объектов. Например, в C++ для этого служит оператор delete. Однако в С# эта проблема решается по-другому, а именно с использованием системы сбора мусора.

Система сбора мусора С# автоматически возвращает память для повторного использования, действуя незаметно и без вмешательства программиста. Ее работа заключается в следующем. Если не существует ни одной ссылки на объект, то предполагается, что этот объект больше не нужен, и занимаемая им память освобождается.

Эту (восстановленную) память снова можно использовать для размещения других объектов. Система сбора мусора удаляет объекты, на которые нет ссылок. Он работает в соответствии со своей внутренней стратегией в неизвестные для программиста моменты времени.

В С# существует специальный вид метода, называемый **деструктором***.* Он вызывается системой **сбора мусора**непосредственно перед удалением объекта из памяти. В деструкторе описываются действия, гарантирующие корректность последующего удаления объекта, например, проверяется, все ли ресурсы, используемые объектом, освобождены (файлы закрыты, удаленное соединение разорвано и т. п.).

Синтаксис деструктора:

[ атрибуты ] [ extern ] ~имякласса()

тело

Как видно из определения, деструктор не имеет параметров, не возвращает значения и не требует указания спецификаторов доступа. Его имя совпадает с именем класса и предваряется тильдой ( ~ ) , символизирующей обратные по отношению к конструктору действия.

Поскольку деструктор вызывается сборщиком мусора, невозможно гарантировать, что деструктор будет обязательно вызван в процессе работы программы. Важно понимать, что деструктор вызывается только перед началом работы системы сбора мусора и не вызывается, например, когда объект выходит за пределы области видимости. (Этим С#-деструкторы отличаются от С++-деструкторов, которые как раз вызываются, когда объект выходит за пределы области видимости.) Все деструкторы будут вызваны перед завершением программы. Применение деструкторов замедляет процесс сборки мусора.

**Ключевое слово this.**

Ключевое слово this обеспечивает доступ к текущему объекту класса.

public void setCarName(string name)

{

this.name = name;

}

**Ключевое слово static.**

Класс С# может определять любое количество статических элементов с использованием ключевого слова static.

Статические элементы класса используются независимо от объекта этого класса. Обычно к элементу класса доступ предоставляется через объект этого класса. Однако можно создать элемент, который заведомо разрешено использовать сам по себе, т.е. без ссылки на конкретный экземпляр.

Если элемент объявлен как static , к нему можно получить доступ до создания объектов этого класса и без ссылки на объект. С использованием ключевого слова static можно объявлять как методы, так и переменные.

Примером static-элемента является метод Main(), который должен быть вызван операционной системой в начале работы программы.

К static – элементу получают доступ не через экземпляр класса, а с помощью имени класса. Аналогично можно вызвать и static-метод, т.е. с помощью оператора "точка" после имени класса.

Переменные, объявленные как static-члены, являются по сути глобальными переменными. При объявлении объектов класса копии static-переменной не создаются, причем все экземпляры класса совместно используют одну и ту же static - переменную. Инициализация static-переменной происходит при загрузке класса. Если инициализатор явно не указан, static-переменная, предназначенная для хранения числовых значений, инициализируется нулем; объектные ссылки — null-значениями, а переменные типа bool – значением false. Таким образом, static -переменная всегда имеет значение.

На static-методы накладывается ряд ограничений.

* static-метод не имеет ссылки this;
* static-метод может напрямую вызывать только другие static-методы. Он не может напрямую вызывать метод экземпляра своего класса, т.к. методы экземпляров работают с конкретными объектами класса;
* static-метод должен получать прямой доступ только к static-данным. Он не может напрямую использовать переменные экземпляров, поскольку не работает с экземплярами класса.

Статические элементы — это «общие» элементы класса, поэтому нет необходимости создавать объект класса при их вызове.

Пример: Console.WriteLine();

Cтатические методы могут оперировать только статическими данными и вызывать статические методы определяющего их класса.

Пример.

class Car

{

**static Random rnd = new Random();**

**public static int GetRandomSpeed()**

**{**

**return rnd.Next(20, 120);**

**}**

. . . . .

}

Car second = new Car("Opel Corsa",Car.GetRandomSpeed());

Car third = new Car("Toyota Corolla", Car.GetRandomSpeed());

**Статические поля**

Когда класс определяет нестатические данные (данные объекта), то каждый объект этого типа поддерживает независимую копию поля.

class Car

{

static Random rnd = new Random();

public string name;

public int speed;

public double cost;

**public static double nds=0.18;**

. . . .

}

**Статический конструктор**

Статический конструктор — это специальный конструктор, для инициализации значений статических данных, когда их значение не известно на момент компиляции (например, когда его нужно прочитать из внешнего файла или сгенерировать случайное число).

* В отдельном классе может быть определен только один статический конструктор.
* Не имеет модификатора доступа и не может принимать параметров.
* Выполняется только один раз, независимо от того, сколько объектов отдельного класса создается.
* Вызывается, когда создает экземпляр класса или перед первым обращением к статическому члену этого класса.
* Выполняется перед любым конструктором объекта класса.

**Статические классы.**

Если класс определен как статический, его нельзя создать с использованием ключевого слова new, и он может включать в себя только статические члены или поля. Если это правило нарушить, возникнет ошибка компиляции. Статический класс не нуждается в выделении памяти.

## 4. Инкапсуляция. Спецификаторы доступа. Свойства.

Объединение данных с функциями их обработки в сочетании со скрытием ненужной для использования этих данных информации называется инкапсуляцией:

* связывает данные с методами их обработки;
* предоставляет средства управления доступом к элементам класса.

**Спецификаторы доступа**

|  |  |  |
| --- | --- | --- |
| **Модификатор доступа** | **К чему применяется** | **Описание** |
| public | классы / элементы классов | не имеют ограничений доступа. |
| private | элементы классов / вложенные классы | доступны только в классе, в котором они определены |
| protected | элементы классов / вложенные классы | могут использоваться классом, который определил их, и любым дочерним классом. |
| internal | классы / элементы классов | доступны только в пределах текущей сборки. |
| protected internal | элементы классов / вложенные классы | доступен внутри определяющей сборки, определяющего класса и всех его наследников |

По умолчанию члены типов являются неявно приватными (private) - элементы и неявно внутренними (internal) - классы.

class Car

{

string name;

int speed;

double cost;

. . . . .

}

**Основной принцип инкапсуляции:** поля класса не должны быть напрямую доступны через объект этого класса.

Если вызывающий код желает изменить состояние объекта, то должен делать это через методы доступа (селектор, или метод get) и изменения (модификатор, или метод set).

Вместо определения общедоступных полей (которые легко приводят к повреждению данных), управление данными осуществляется с помощью:

* селектора и модификатора;
* свойства .NET.

class Car

{

string name;

int speed;

double cost;

public string GetName()//селектор

{

return name;

}

public void SetName(string name)//модификатор

{

this.name = name;

}

. . . . . . .

}

**Свойства**служат для организации доступа к полям класса.

Синтаксис свойства:

[ атрибуты ] [ спецификаторы ] тип имя\_свойства

{

[ get код\_доступа ]

[ set код\_доступа ]

}

Свойства, а в особенности их часть set — это общепринятое место для размещения бизнес-правил класса:

class Car

{

string name;

int speed;

double cost;

public int Speed

{

get { return speed; }

//бизнес-правила

set { if (value < 200) speed = value;

else

Console.WriteLine("Слишком большая скорость!");

}

}

. . . . . .

}

//свойство доступное для чтения

public int Speed

{

get { return speed; }

}

**Автоматические свойства**

Хотя большинство свойств С# содержат в своем контексте бизнес-правила, не так уж редко бывает, что некоторые свойства выполняют только простое присваивание и возврат значений.

Создавать автоматические свойства, предназначенные только для чтения или только для записи, нельзя.

class Car

{

public string Name

{get; set; }

public string Speed

{get; set;}

. . . . .

}

**Решение задачи с использованием классов.**

1. Написать класс для работы с простыми дробями (числитель и знаменатель). В классе реализовать:

* конструктор без параметров,
* конструктор с параметром,
* свойства, метод для вывода дроби на экран (при выводе учесть, что, если числитель больше, чем знаменатель, то у дроби должна выводиться целая часть, если числитель равен знаменателю, то выводится единица).

namespace Fraction\_1

{

class Fraction

{

int numerator;

int denominator;

//свойства

public int Numerator

{

get { return numerator; }

set { numerator = value; }

}

public int Denominator

{

get { return denominator; }

set { if (value != 0)denominator = value; }

}

public Fraction()//конструктор без параметров

{

numerator = 0;

denominator = 1;

}

public Fraction(int numerator, int denominator)

{

this.numerator = numerator;

if (denominator != 0) this.denominator = denominator;

else this.denominator = 1;

}

public void Show() //метод для вывода дроби

{

int int\_part;

if (numerator < denominator)

Console.Write(numerator + "/" + denominator + "\n");

else

if (numerator == denominator) Console.WriteLine(1);

else

{

int\_part = numerator % denominator;

if (int\_part == 0)

Console.WriteLine(numerator / denominator);

else

Console.WriteLine(numerator / denominator + " " + numerator % denominator + "/" + denominator);

}

}

}

class Program

{

static void Main(string[] args)

{

Fraction x1 = new Fraction(2, 1);

x1.Show();

Fraction x2 = new Fraction(1, 2);

x2.Show();

Fraction x3 = new Fraction(2, 2);

x3.Show();

}

}

}

1. Написать статическую функцию для сложения двух дробей

class Program

{

//сложение дробей

public static Fraction plus(Fraction a, Fraction b)

Fraction res = new Fraction();

if (a.Denominator == b.Denominator)

{

res.Numerator = a.Numerator + b.Numerator;

res.Denominator = a.Denominator;

}

else

{

res.Numerator = a.Numerator \* b.Denominator + b.Numerator \* a.Denominator;

res.Denominator = a.Denominator \* b.Denominator;

}

return res;

}

//точка входа

static void Main(string[] args)

{

Fraction x1 = new Fraction(2, 1);

x1.Show();

Fraction x2 = new Fraction(1, 2);

x2.Show();

Fraction x3 = new Fraction(2, 2);

x3.Show();

Fraction x4 = new Fraction();

x4 = plus(x1, x2);

x4.Show();

Fraction x5 = new Fraction();

x5 = plus(x2, x3);

x5.Show();

}

}

}

1. Заменить статическую функцию plus класса Program на метод plus класса Fraction.

//закрытая функция для вычисления наибольшего общего делителя, используется при //приведении дроби к общему знаменателю

private int NOD (int a,int b)

{

int r;

while(b!=0)

{

r = a%b;

a = b; b = r;

}

return a;

}

//сложение дробей

public Fraction plus(Fraction b)

{

Fraction res = new Fraction();

if (Denominator == b.Denominator)

{

res.Numerator = Numerator + b.Numerator;

res.Denominator = Denominator;

}

else

{

res.Numerator = Numerator \* b.Denominator + b.Numerator \* Denominator;

res.Denominator =Denominator \* b.Denominator;

}

int nod = NOD(res.numerator, res.denominator);

res.numerator /= nod;

res.denominator /= nod;

return res;

}

class Program

{

static void Main(string[] args)

{

Fraction x1 = new Fraction(2, 1);

x1.Show();

Fraction x2 = new Fraction(1, 2);

x2.Show();

Fraction x3 = new Fraction(2, 2);

x3.Show();

Fraction x4 = new Fraction();

x4 = x1.plus(x2);

x4.Show();

Fraction x5 = new Fraction();

x5 = x2.plus(x3);

x5.Show();

}

}

}

1. Добавим статическую переменную для подсчета количества объектов класса Fraction в класс Fraction и статическую функцию для просмотра найденного количества.

class Fraction

{

int numerator;

int denominator;

static int count=0;

public Fraction()//конструктор без параметров

{

numerator = 0;

denominator = 1;

**count++;**

}

public Fraction(int numerator, int denominator)

{

this.numerator = numerator;

if (denominator != 0) this.denominator = denominator;

else this.denominator = 1;

**count++;**

}

. . . .

static public int GetCount()

{

return count;

}

}

static void Main(string[] args)

{

Fraction x1 = new Fraction(2, 1);

x1.Show();

Fraction x2 = new Fraction(1, 2);

x2.Show();

Fraction x3 = new Fraction(2, 2);

x3.Show();

Fraction x4 = new Fraction();

x4 = x1.plus(x2);

x4.Show();

Fraction x5 = new Fraction();

x5 = x2.plus(x3);

x5.Show();

Console.WriteLine("В программе создано {0} объектов", Fraction.GetCount());

}

