Урок 1

Объектно-ориентированное программирование. Часть 1

Структура. Класс. Объект. Инкапсуляция. Конструкторы. Свойства. Индексаторы. Наследование и полиморфизм.

«Не паникуйте раньше времени! Все постепенно станет понятным»

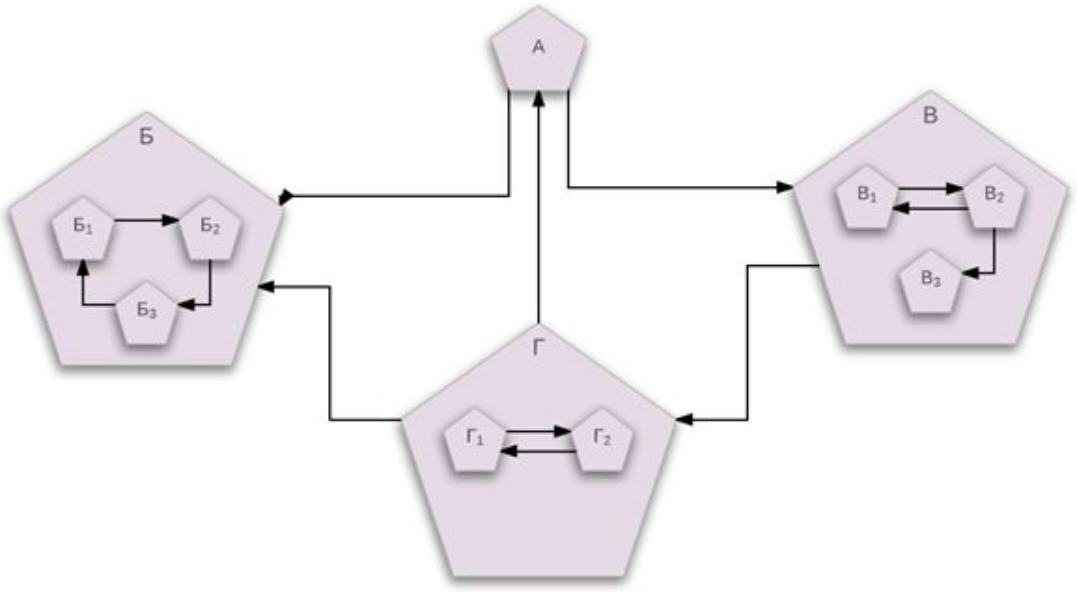
Бьерн Страуструп
«Язык программирования C++»

Объектно-ориентированное программирование

Как отмечал известный нидерландский программист Эдсгер Дейкстра, человечество еще в древности придумало способ управления сложными системами: «Разделяй и властвуй». Это означает, что исходную систему нужно разбить на подсистемы так, чтобы работу каждой из них можно было рассматривать и совершенствовать независимо от других.

Для этого в классическом (процедурном) программировании используют метод проектирования «сверху вниз»: сложная задача разбивается на части (подзадачи и соответствующие им алгоритмы), которые затем снова разбиваются на более мелкие подзадачи и т.д. Однако при этом задачу «реального мира» приходится переформулировать, представляя все данные в виде переменных, массивов, списков и других структур данных. При моделировании больших систем объем этих данных увеличивается, они становятся плохо управляемыми, и это приводит к большому числу ошибок. Так как любой алгоритм может обратиться к любым глобальным (общедоступным) данным, повышается риск случайного недопустимого изменения значений.

Поскольку формулировка задач, решаемых на компьютерах, все более приближается к формулировкам реальных жизненных задач, возникла идея представить программу в виде множества объектов (моделей). Каждый из них обладает своими свойствами и поведением, но его внутреннее устройство скрыто от других объектов. Тогда решение задачи сводится к моделированию взаимодействия этих объектов. Построенная таким образом модель задачи называется объектной. Здесь тоже идет проектирование «сверху вниз», только не по алгоритмам (как в процедурном программировании), а по объектам.



Если построена объектная модель задачи, можно поручить разработку каждого из объектов отдельному программисту. Он должен написать соответствующую часть программы, т.е. определить, как именно объект будет выполнять свои функции. При этом разработчику не обязательно держать в голове полную информацию обо всех объектах – нужно лишь строго соблюдать соглашения о способе обмена данными своего объекта с другими.

Пример ООП в приложении логической игры «Удвоитель» в Windows Forms:

Приложение состоит из объектов **Form**, **Label**, **Button** (разработанные разными программистами) и некоторых других. У каждого объекта есть предназначение. **Form** содержит другие элементы. **Button** отображает надписи на кнопке и реализует нажатия. Label реализует отображения надписи. Разрабатывая приложение, программист связывает эти объекты между собой.

Класс – объект

Инкапсуляция – это механизм программирования, объединяющий код и данные, которыми он манипулирует. Она исключает как вмешательство извне, так и неправильное использование данных. В примере ниже мы создали класс **Vector**, который содержит два поля: **x** и **y**. С помощью такого класса можно описать координаты объекта на плоскости. Для создания экземпляра класса мы должны использовать ключевое слово **new**. Для доступа к полям **x** и **y** мы сделали их публичными.

Теперь можем описать объект **v1** класса **Vector** (или другие объекты класса, каждый из которых будет содержать собственные координаты):

```
using System;
namespace Class_Vector0010
{
    class Vector
    {
        public double X;
        public double Y;
    }
    class Program
    {
        static void Main(string[] args)
        {
            Vector v1 = new Vector();
            v1.X = 10;
            v1.Y = 5;
            Vector v2;
            v2 = new Vector();
            v2.X = -5;
            v2.Y = -10;
        }
    }
}
```

Как же правильно: класс или объект? Говоря про класс **Vector**, мы подразумеваем все объекты, которые могут быть созданы. Говоря про объект **v1**, мы имеем в виду конкретный экземпляр объекта, который хранится в памяти.

Инкапсуляция

Делая поля публичными, мы на первых порах упрощаем себе жизнь, но нарушаем одно из правил объектно-ориентированного программирования. Оно требует, чтобы доступ к внутренней структуре объекта извне было невозможно получить. Разберемся, как же тогда изменять данные объекта и почему вообще нужно закрывать данные.

Данные нужно закрывать, потому что программист, который описывает класс (точнее, поведение будущего объекта), должен обеспечить правильное поведение объекта. Если предоставить возможность изменять данные напрямую, то в объекте их могут заполнить в некорректной форме. Попробуйте прописать такой пример, заполнив его правильными данными:

```
DateTime date = new DateTime(2016, 10, 20);
```

А здесь попытаемся записать 15 месяц и 40 число:

```
DateTime date = new DateTime(2016, 15, 40);
```

Выскочит исключительная ситуация – правильно сконструированный класс не даст создать неправильный объект.

Чтобы управлять данными внутри объекта, существуют различные технологии ООП. Рассмотрим некоторые из них.

Модификаторы доступа

В C# .Net существует 6 модификаторов доступа:

- **public** – публичный, общедоступный класс или член класса. Он доступен из любого места в коде, а также из других программ и сборок;
- **private** – закрытый класс или член класса, полная противоположность модификатору **public**. Доступен только из кода в том же классе или контексте;
- **protected** – 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. Но он недоступен для других программ и сборок (как модификатор **public**);
- **protected internal** – совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.
- **private protected** – комбинация ключевых слов private protected является модификатором доступа к члену. К члену private protected имеют доступ типы, производные от содержащего класса, но только в пределах содержащей сборки. Сравнение модификатора private protected с другими модификаторами доступа (https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/private-protected).

Если мы явно не указываем модификатор доступа, он выставляется автоматически. Для класса это **internal**, поля класса и функции – **private**.

Статический конструктор

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

```
internal class Helper
{
    static Helper()
    {
    }
}
```

Конструкторы

Конструктор – это специальный метод, который вызывается при создании экземпляра объекта. В **.Net Framework**, если в созданном вами классе нет описанного вами конструктора, создается конструктор без параметров, который заполняет поля объекта данными по умолчанию (**0**, **false**, **null**). Чтобы при создании объект был заполнен данными, добавляем конструктор с параметрами.

```
using System;
namespace Class_Vector0010
{
    class Vector
    {
        // Теперь поля приватные
        private double _x;
        private double _y;
        // Переопределим конструктор по умолчанию
        public Vector()
        {
            _x = _y = 0;
        }
        // Конструктор, который будет заполнять поля объекта
        public Vector(double x,double y)
        {
            _x = x;
            _y = y;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Vector v1 = new Vector(10,5);
            Vector v2;
            v2 = new Vector(-5,-10);
        }
    }
}
```

Методы

Для доступа к закрытым данным можно использовать открытые методы:

```
using System;
namespace Class_Vector0010
{
       class Vector
      {
// Теперь поля приватные
            private double _x;
            private double _y;
// Переопределим конструктор по умолчанию
            public Vector()
            {
                  _x = _y = 0;
            }
// Конструктор, который будет заполнять поля объекта
            public Vector(double x, double y)
            {
                  _x = x;
                  _y = y;
            }
// С версии C# 6.0 появилась новая функциональность - встроенные методы, или
Expression-Bodied Methods. Они позволяют применять лямбда-выражения для
сокращенного написания методов в одну строку.
            public double GetX() => _x;

            public void SetX(double value) => _x = value;

            public double GetY() => _y;

            public void SetY(double value) => _y = value;

            // Метод для получения данных в строковой форме
            public string ToString() => $"X={_x} Y={_y}";
      }
      class Program
      {
            static void Main(string[] args)
            {
                  Vector v1 = new Vector(10, 5);
                  Vector v2;
                  v2 = new Vector(-5, -10);
                  v1.SetY(10);
                  v2.SetX(-10);
                  Console.WriteLine($"v1:{v1.ToString()}");
                  Console.WriteLine($"v2:{v2.ToString()}");
            }
      }
}
```

Хотя методы вполне подходят для доступа к закрытым полям, существует альтернативный способ доступа через свойства.

Свойства

С помощью свойств можно задавать значения закрытым полям объекта. Технология свойств объединяет в себе поле и метод. Вместо того, чтобы создавать несколько методов для записи или чтения данных из полей объекта, можно объединить эти действия в одном свойстве:

```
using System;
namespace Class_Vector0010
{
    class Vector
    {
        // Теперь поля приватные
        private double _x;
        private double _y;
        // Переопределим конструктор по умолчанию
        public Vector()
        {
            _x = _y = 0;
        }
        // Конструктор, который будет заполнять поля объекта
        public Vector(double x, double y)
        {
            _x = x;
            _y = y;
        }
        // Свойство X для доступа к полю x
        public double X
        {
            get => _x;
            set => _x = value;
        }
        // Свойство Y для доступа к полю y
        public double Y
        {
            get => _y;
            set => _y = value;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Vector v1 = new Vector(10, 5);
            Vector v2;
            v2 = new Vector(-5, -10);
            // Доступ к полям стал более логичным при записи
            v1.X = 10;
            v2.X = -10;
            Console.WriteLine($"v1: X={v1.X} Y={v1.Y}"); // и при чтении
        }
    }
}
```

При описании свойств используются аксессоры доступа **get** и **set**. С их помощью программист управляет возможностями читать или записывать данные. Можно применять модификаторы доступа не только ко всему свойству, но и к отдельным блокам – **get** или **set**. Модификатор можно применить только к одному из блоков:

```
private double _x;

// До C# 7.0
public double X
{
    get { return _x; }
    protected set { _x = value; }
}

// В последующих версиях

public double X_2
{
    get => _x;
    protected set => _x = value;
}
```

Автоматические свойства

С версии C# 3.0 появилась возможность реализовать очень простые свойства, не прибегая к явному определению переменной, которой они управляют. Вместо этого базовую переменную для свойства автоматически предоставляет компилятор. Такое свойство называется автоматически реализуемым и принимает следующую общую форму:

```
тип имя { get; set; }
```

Здесь тип обозначает конкретный тип свойства, а имя – присваиваемое свойству имя. Обратите внимание на то, что после обозначений аксессоров **get** и **set** сразу же следует точка с запятой, а тело у них отсутствует. Такой синтаксис предписывает компилятору автоматически создать для хранения значения переменную, иногда еще называемую поддерживающим полем. Такая переменная недоступна непосредственно и не имеет имени, но может быть доступна через свойство.

Пример описания автоматического свойства:

```
public string Name { get; set; }
public int CustomerID { get; set; }
```

С версии C# 6.0 можно проинициализировать автосвойство:

```
public string Name { get; set; } = "Ivan";
public int CustomerID { get; private set; } = 15;
```

В C# 5.0, чтобы автосвойство было доступным для установки только из класса, необходимо указать **private set**. С версии C# 6.0 не обязательно писать **private set** – можно оставить только выражение **get**.

Для хранения значения этого свойства для него неявно будет создаваться поле с модификатором **readonly**. Следует учитывать, что подобные get-свойства можно установить либо из конструктора класса, либо при инициализации свойства.

Перегрузка операторов

Покажем перегрузку операторов на примере класса **Vector**. Перегрузка операторов позволяет использовать пользовательские классы с привычными операторами **+, -, *, =** и другими.

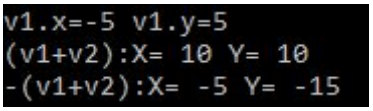
Класс Vector

```
using System;
namespace Vectors
{
// Введем структуру Vector и перегрузим для нее операции +, -, =
      class Vector
      {
            public double X { get; private set; }

            public double Y { get; private set; }

            public Vector()
            {
                  X = Y = 0;
            }

            public Vector(double x, double y)
            {
                  X = x;
                  Y = y;
            }

// Ключевое слово implicit служит для объявления неявного оператора
преобразования пользовательского типа. Этот оператор обеспечивает неявное
преобразование между пользовательским типом и другим типом, если при
преобразовании исключается утрата данных
// explicit - Ключевое слово для объявления явного оператора преобразования.
Значит, мы указываем на потерю данных.

            public static explicit operator Vector(double x) => new Vector(x,
x);

            public static implicit operator double(Vector x) => x.X;
// Переопределение метода ToString
            public override string ToString() => $"X= {X} Y= {Y}";

// Перегрузка бинарного оператора +
            public static Vector operator +(Vector v1, Vector v2)
```

```
            {
                  Vector res = new Vector
                  {
                        X = v1.X + v2.X,
                        Y = v1.Y + v2.Y
                  };
                  return res;
            }

// Перегрузка бинарного оператора -
            public static Vector operator -(Vector v1, Vector v2)
            {
                  Vector res = new Vector
                  {
                        X = v1.X - v2.X,
                        Y = v1.Y - v2.Y  Ошибка
                  };
                  return res;
            }

// Перегрузка унарного оператора -
            public static Vector operator -(Vector v1)
            {
                  Vector res = new Vector
                  {
                        X = -1 * v1.X,
                        Y = -1 * v1.Y
                  };
                  return res;
            }
      }

      class Program
      {
            static void Main()
            {
// Создаем вектор
                  Vector v1 = new Vector(-5, 5);
// Другой вектор задаем, используя перегрузку = explict
                  Vector v2 = (Vector) 10;
                  Vector v3 = v1 + v2; // Проверяем работу +
// Демонстрация доступа к закрытым полям через свойства
                  Console.WriteLine($"v1.x={v1.X} v1.y={v1.Y}");
                  Console.WriteLine($"(v1+v2):{v3}");Ошибка
                  Console.WriteLine($"-(v1+v2):{-v3}"); // и -
                  Console.ReadKey();
            }
      }
}
```

В примере продемонстрировано, как можно перегрузить операции для новых типов данных. Эта возможность позволяет упростить дальнейшее программирование, так как теперь операции сложения, вычитания и присваивания происходят как над единым объектом.

Результат примера:



Структуры

Структуры похожи на классы, но имеют и некоторые отличия. Структуры относятся к типам значений. В отличие от классов, структуры не поддерживают наследование. Структура не может наследовать от другой структуры или класса и не может быть базовой для класса. По этой причине члены структуры не могут объявляться как **protected**. Структуры могут реализовывать интерфейсы именно так, как это делают классы.

Пример структуры:

```
using System;
namespace Struct_Vector0010
{
    struct Vector
    {
        public double X;
        public double Y;
        // В структурах не может быть описан свой конструктор без параметров
        // public Vector()
        // {
        //    x = y = 0;
        // }
        // Конструктор, который будет заполнять поля объекта
        public Vector(double x, double y)
        {
            X = x;
            Y = y;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
        // Для структур не нужно использовать ключевое слово new
            Vector v1;
            v1.X = 10;
            v1.Y = 5;
            Vector v2;
       // Хотя можно вызывать конструкторы, но у структур они нужны только для
заполнения полей структуры
            v2 = new Vector(-5, -10);
        }
    }
```

```
}
```

Для структур, как и для классов, возможно переопределять операторы +, -, = и другие.

Выбор между структурой и классом приходится делать программисту. Когда модель содержит небольшое количество данных внутри себя (как в примере с **Vector**), ее бывает выгодней описать в виде структуры, так как на работу со структурой не тратится время на выделение памяти. В C# нельзя объявить или переопределить конструктор по умолчанию для структуры. Можно только объявить параметризованный конструктор. И здесь два варианта:

1. Либо в параметризованном конструкторе инициализировать все поля структуры.

2. Либо в параметризованном конструкторе инициализировать не все поля, но вызывать конструктор по умолчанию.

У структуры нельзя проинициализировать поля по умолчанию.

```
private double _x;
private double _y;
public Vector(double x) : this()
{
     _y = 0;
     _x = x;
}
```

Наследование

Наследование представляет собой процесс, в ходе которого один объект приобретает свойства другого.

В языке C# наследуемый класс называется базовым, а наследующий – производным. Следовательно, производный класс представляет собой специализированный вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы, определяемые в базовом классе, добавляя к ним собственные элементы.

Наследование является одним из трех основополагающих принципов объектно-ориентированного программирования, поскольку оно допускает создание иерархических классификаций. Благодаря наследованию можно создать общий класс, в котором определяются характерные особенности, присущие множеству связанных элементов. От этого класса могут затем наследовать другие, более конкретные классы, добавляя в него свои индивидуальные особенности.

В C# (точнее, в .Net Framework) все объекты наследуются от базового класса **object**. В практическом смысле мы можем присваивать переменной класса **object** любое значение.

```
Object a=new Object();
a=10;
a=3.14;
a="Строка";
a=new Random();
```

Хотя такой способ существует и может быть использован при программировании, разработчики стараются избегать его, так как приходится заботиться о том, какие данные хранятся в переменных типа **object**. При данном подходе вы столкнетесь с такими явлениями, как упаковка (boxing) и распаковка (unboxing). Первая предполагает преобразование объекта значимого типа (например, типа **int**) к типу **object**. При упаковке общеязыковая среда **CLR** обертывает значение в объект типа **System.Object** и сохраняет его в управляемой куче (хипе). Распаковка (unboxing), наоборот, предполагает преобразование объекта типа **object** к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования (выделение памяти и копирование).

Все классы по умолчанию могут наследоваться, но есть ряд ограничений:

- Не поддерживается множественное наследование – класс может наследоваться только от одного класса. Хотя проблема множественного наследования решается с помощью концепции интерфейсов;
- При создании производного класса надо учитывать тип доступа к базовому классу – тип доступа к производному классу должен быть таким же, как у базового класса, или более строгим. Если базовый класс имеет тип доступа **internal**, то производный класс может иметь тип доступа **internal** или **private**, но не **public**;
- Если класс объявлен с модификатором **sealed**, от него нельзя наследовать и создавать производные классы.

Пример наследования:

```
using System;
namespace inheritance_0020
{
       class Vector
      {
            public double X { get; set; }
            public double Y { get; set; }
            // Переопределим конструктор по умолчанию
            public Vector()
            {
                  X = Y = 0;
            }
            // Конструктор, который будет заполнять поля объекта
            public Vector(double x, double y)
            {
                  X = x;
                  Y = y;
            }
            // Метод для получения данных в строковой форме
            public string ToString() => $"X={X} Y={Y}";
      }
      class MyObject : Vector
      {
      }
      class Program
      {
            static void Main(string[] args)
            {
                  MyObject obj1 = new MyObject
```

```
                    {
                        X = 10,
                        Y = 20
                    };
                    Console.WriteLine(obj1.ToString());
                }
            }
}
```

Мы создали новый класс **MyObject**, который наследует публичные (**public**) и защищенные (**protected**) члены класса **Vector**.

Теперь мы можем добавлять в новый класс собственные поля, методы и свойства, а также пользоваться неприватными полями, методами и свойствами базового класса.

Запустите пример в пошаговом режиме (F11) и обратите внимание, как происходят вызовы конструкторов объектов. Важно понять, что для построения нашего объекта должен быть сконструирован объект базового класса и все объекты в иерархии наследования.

Конструкторы не передаются производному классу при наследовании. И если в базовом классе не определен конструктор по умолчанию без параметров (а только конструкторы с параметрами), то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово **base**.

```
public class Animals
{
    private string _name;
    private int _numberPaws;
    public Animals(string name, int numberPaws)
    {
        _name = name;
        _numberPaws = numberPaws;
    }
}
public sealed class Cat : Animals
{
    private ushort? _lengthTail;
    public Cat(string name, int numberPaws, ushort? lengthTail) : base(name,
        numberPaws)
    {
        _lengthTail = lengthTail;
    }
}
```

Значение null и Nullable-типы

Бывают случаи, когда программистам удобно, чтобы объекты значимых типов данных имели значение **null**, то есть были бы не определены. Для этого надо использовать знак вопроса (**?**) после типа значений.

```
int? test = null;
bool? isEnabled = null;
```

Запись **?** является упрощенной формой использования структуры **System.Nullable<T>**. Это обобщенная структура (рассмотрим обобщения в последующих уроках).

```
int? test = 5;
bool? isEnabled = null;
double? pi = 3.14;

Nullable<int> test1 = 5;
Nullable<bool> isEnabled1 = null;
Nullable<double> pi1 = 3.14;
```

Чтобы получать значение объекта, необходимо обратиться к свойству **Value**:

```
int? b = 1;
if (b.HasValue) // Прежде чем получить значение объекта, необходимо проверить, хранит ли объект какое-либо значение
{
    Console.WriteLine($"Значение b = {b.Value}");
}
```

Оператор **??** называется оператором null-объединения. Он применяется для установки значений по умолчанию для типов значений и ссылочных типов, которые допускают значение **null**. Оператор **??** возвращает левый операнд, если он не равен **null** – иначе возвращается правый операнд (в этом случае левый операнд должен принимать null):

```
int? x = null;
int y = x ?? 2;  // Равно 2, так как x равен null

int? a = 5;
int b = a ?? 10; // Равно 5, так как a не равен null
```

С версии C# 6.0 в языке появился оператор условного **null** (**Null-Conditional Operator),** или элвис-оператор. Он позволяет упростить проверку на значение **null** в условных конструкциях:

```
class Person
{
	public Adress Adress;
}

class Adress
{
	public string City;
}

class Program
{
	private string GetCityInPast(Person person)
	{
		// До C# 6.0
		string city = String.Empty;
		if (person != null)
		{
			if (person.Adress != null)
			{
				city = person.Adress.City;
			}
		}
		return city;
	}

	private string GetCityInPresent(Person person)
	{
		return person?.Adress?.City ?? ""; // Начиная с C# 6.0
	}
}
```

Наследование включением (агрегация)

Есть альтернативный наследованию механизм использования одним классом другого. Это вложения, когда один класс является полем другого.

```
using System;
namespace Inheritance_Incloser_0010
{
// Пример наследования
	class Vector
	{
		public double X { get; set; }
		public double Y { get; set; }
		// Переопределим конструктор по умолчанию
		public Vector()
		{
			X = Y = 0;
		}
		// Конструктор, который будет заполнять поля объекта
		public Vector(double x, double y)
		{
			X = x;
			Y = y;
		}
		// Метод для получения данных в строковой форме
		public string ToString() => String.Format($"X={X} Y={Y}");
	}
	class MyObject
	{
		public Vector Pos { get; set; }
		private double _width;
		private double _height;
		public MyObject(double width, double height, Vector v)
		{
			_width = width;
			_height = height;
			Pos = v;
		}
// Переопределим метод, который выводит информацию о нашем поле в виде строки
		public string ToString() => $"width:{_width} height:{_height}
{Pos.ToString()}";
	}
	class Program
	{
		static void Main(string[] args)
		{
			MyObject obj1 = new MyObject(10, 20, new Vector(1, 1));
// Теперь доступ к полям можно осуществить через поле pos
			obj1.Pos.X = 10;
			obj1.Pos.Y = 20;
			Console.WriteLine(obj1.ToString());
		}
	}
}
```

Если ваш класс является разновидностью класса, лучше использовать наследование. Если же внутри класса вам необходимо использовать несколько объектов другого класса, необходимо включение. Понимание, какое наследование использовать, приходит с опытом.

Виртуальный метод

Виртуальным называется такой метод, который объявляется как **virtual** в базовом классе. Он отличается тем, что может быть переопределен в одном или нескольких производных классах. Для переопределения используется **override**.

При определении класса-наследника и наследовании методов базового класса мы можем выбрать одну из следующих стратегий:

1. Обычное наследование всех членов базового класса в классе-наследнике;
2. Переопределение членов базового класса в классе-наследнике;
3. Скрытие членов базового класса в классе-наследнике.

```
using System;

public class Animals
{
    public virtual string DisplayFirstWay()
    {
        return $"I am a {nameof(Animals)} class method";
    }

    public virtual string DisplaySecondWay()
    {
        return $"I am a {nameof(Animals)} class method";
    }

    public virtual string DisplayThirdWay()
    {
        return $"I am a {nameof(Animals)} class method";
    }
}

public class Cat : Animals
{
    public override string DisplaySecondWay()
    {
        return $"I am a {nameof(Cat)} class method";
    }

    public new string DisplayThirdWay()
    {
        return $"I am a {nameof(Cat)} class method";
    }
}

public class Program
{
    public static  void Main()
    {
        Animals animals = new Animals();
        Console.WriteLine(animals.DisplayFirstWay());
        Console.WriteLine(animals.DisplaySecondWay());
        Console.WriteLine(animals.DisplayThirdWay());

        Animals animals1 = new Cat();
        Console.WriteLine(animals1.DisplayFirstWay());
        Console.WriteLine(animals1.DisplaySecondWay());
        Console.WriteLine(animals1.DisplayThirdWay());

        Cat animals2 = new Cat();
        Console.WriteLine(animals2.DisplayFirstWay());
        Console.WriteLine(animals2.DisplaySecondWay());
        Console.WriteLine(animals2.DisplayThirdWay());
    }
}
```

ToString()

На самом деле, мы с вами уже несколько раз переопределяли (точнее, скрывали) виртуальный метод **ToString**. Все классы в **.Net Framework** наследуются от базового класса **System.Object**, в котором определены несколько виртуальных методов: **ToString()**, **Equals()**, **GetHashCode()**.

Это сделано специально, чтобы наследники (все классы) могли их переопределить и сделать их поведение более естественным для их объекта. Иначе вызывается поведение, заданное в **System.Object** – но оно не может знать о реализации вашего объекта и ведет себя довольно примитивно. Например, не переопределенный (принадлежащий **System.Object**) метод **ToString** выводит информацию о том, к какому классу и пространству имен принадлежит созданный вами класс.

Чтобы правильно переопределить поведение **ToString**, мы должны добавить слово **override** перед описанием данного метода в нашем классе. Без **override** мы скрываем базовое поведение и не даем будущим потомкам нашего класса использовать поведение, заложенное в базовом классе:

```
using System;
namespace Override_010
{
    class MyObject
    {
        private int _a;
        public MyObject(int a)
        {
            _a = a;
        }
        // Запустите программу, закомментировав метод ToString
        public override string ToString() => $"a= {_a}";
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyObject obj = new MyObject(2);
            Console.WriteLine(obj.ToString());
        }
    }
}
```

Программист может использовать уже готовые классы, разрабатывать собственные или изменять другие классы, используя наследование и иные механизмы.

Полиморфизм

Полиморфизм – это свойство, позволяющее одному интерфейсу получать доступ к общему классу действий. Здесь «интерфейс» – это способ взаимодействия. Про интерфейсы, как технологию программирования, мы поговорим на следующем уроке.

Пример полиморфизма:

```
using System;
namespace Virtual_0010
{
    class MyObject
    {
        public virtual void Show()
        {
            Console.WriteLine("Я - виртуальный метод Show");
        }
    }
    class MyObject2 : MyObject
    {
        public override void Show()
        {
            Console.WriteLine("Я - переопределенный метод Show");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyObject obj1 = new MyObject();
            MyObject2 obj2 = new MyObject2();
            // Вызываем метод Show объекта класса MyObject
            obj1.Show();
            // Вызываем метод Show объекта класса MyObject2
            obj2.Show();
            // Демонстрируем полиморфизм
            // Объекты базовых классов могут ссылаться на объекты производных
классов
            MyObject obj3 = new MyObject2();
            // Но при вызове метода будет вызываться переопределенный метод
            obj3.Show();
        }
    }
}
```

IS и AS

IS проверяет совместимость объекта с заданным типом, AS — проверяет и, при возможности, преобразует объект. В следующем коде определяется, является ли объект экземпляром типа **MyObject** или типа, производного от **MyObject**:

```
using System;
namespace IsAs
{
    class MyObject
    {
    }
    class MyObject2 : MyObject
    {
    }
    class Program
    {
        static void Main(string[] args)
        {
            // Здесь все понятно
            MyObject obj1 = new MyObject();
            MyObject2 obj2 = new MyObject2();
            if (obj1 is MyObject) Console.WriteLine("obj1 является объектом
класса MyObject");
            else Console.WriteLine("obj1 не является объектом класса MyObject");
            if (obj2 is MyObject2) Console.WriteLine("obj2 является объектом
класса MyObject2");
            else Console.WriteLine("obj2 не является объектом класса
MyObject2");
            // Здесь мы демонстрируем полиморфизм
            // Объекты базовых классов могут ссылаться на объекты производных
классов
            MyObject obj3 = new MyObject2();
            if (obj3 is MyObject) Console.WriteLine("obj3 является объектом
класса MyObject");
            else Console.WriteLine("obj3 не является объектом класса MyObject");
        }
    }
}
```

Оператор **as** используется для преобразования типов между совместимыми ссылочными типами или для типа, допускающего значение **NULL**. Он подобен оператору приведения **(int)x**. Если преобразование невозможно, **as** возвращает **null** вместо вызова исключения.

```
using System;

namespace IsAs020
{
    class MyObject
    {
    }

    class MyObject2 : MyObject
    {
    }

    class Program
    {
        static void Main(string[] args)
        {
            MyObject obj1 = new MyObject();
            MyObject2 obj2 = new MyObject2();
            MyObject obj = obj1 as MyObject;
            if (obj != null) Console.WriteLine("Мы теперь можем обращаться к с
obj как MyObject");
            obj = obj2 as MyObject2;
            if (obj != null) Console.WriteLine("Мы теперь можем обращаться к с
obj как MyObject2");
        }
    }
}
```

Раннее и позднее связывание

В терминах объектно-ориентированного программирования раннее связывание означает, что объект и вызов функции связываются между собой на этапе компиляции. Вся информация, чтобы определить, какая именно функция будет вызвана, известна на этапе компиляции программы. Примеры раннего связывания: стандартные вызовы функций, вызовы перегруженных функций и перегруженных операторов. Принципиальным достоинством раннего связывания является его эффективность – оно более быстрое и обычно требует меньше памяти, чем позднее связывание. Недостаток – слабая гибкость.

Позднее связывание означает, что объект связывается с вызовом функции только во время исполнения программы, а не раньше. Позднее связывание достигается в C# с помощью виртуальных методов и производных классов. Его достоинство – высокая гибкость. Оно может использоваться для поддержки общего интерфейса, позволяя объектам иметь собственную реализацию этого интерфейса. Оно помогает создавать библиотеки классов, допускающие повторное использование и расширение.

Рассмотрим, как это реализовано в **.NET Framework**.

Предположим, мы описали метод виртуальным:

```
virtual public void Draw()
```

Объявление метода виртуальным означает, что все ссылки на него будут разрешаться по факту его вызова (не на стадии компиляции, а во время выполнения программы). Это и есть механизм позднего связывания.

Для его реализации необходимо, чтобы адреса виртуальных методов хранились там, где ими можно будет в любой момент воспользоваться. Поэтому компилятор формирует для них таблицу виртуальных методов (**Virtual Method Table**, **VMT**). В нее записываются адреса виртуальных методов (в том числе, унаследованных) в порядке описания в классе. Для каждого класса создается одна таблица.

Каждый виртуальный объект во время выполнения имеет доступ к **VMT**. Эта связь устанавливается во время выполнения программы при создании объекта.

Практика

ToString()

Простой практический пример использования полиморфизма:

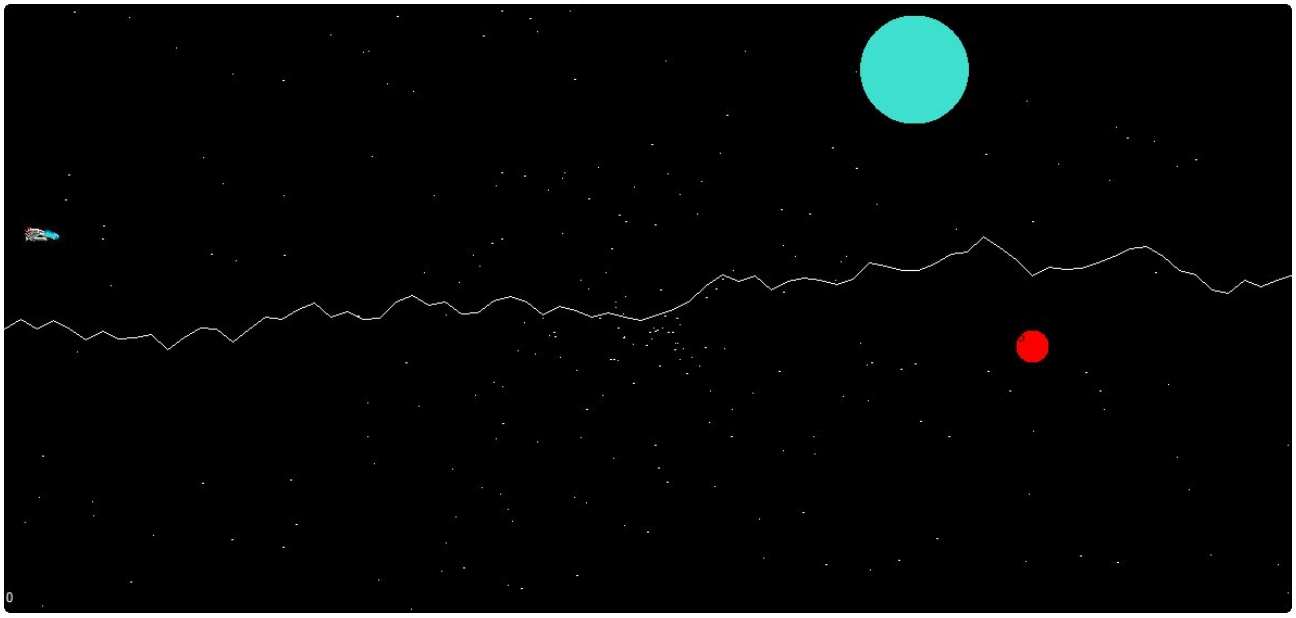
```
using System;
namespace Override_ToString
{
    class MyClass: Object
    {
        private int _a;
        public MyClass(int a)
        {
            _a = a;
        }
        // Попробуйте раскомментировать этот метод и запустить программу
        // public override string ToString() =>  _a.ToString();
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyClass obj = new MyClass(10);
            Console.WriteLine(obj);
            Console.ReadKey();
        }
    }
}
```

Перегрузка Equals

По умолчанию **Equals** сравнивает, ссылаются ли два объекта на одну и ту же область памяти. Можно переопределить его так, чтобы **Equals** сравнивал два объекта по их содержимому:

```
using System;
namespace OverrideEquals
{
    class Vector
    {
        public double X, Y;
        public Vector(double x, double y)
        {
            X = x;
            Y = y;
        }
        public override bool Equals(object obj)
        {
            Vector v = obj as Vector;
            if (v == null) return false; // Или можно создать исключение throw
new InvalidCastException()
            return (obj as Vector).X == X && (obj as Vector).Y == Y;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Vector v1 = new Vector(1, 1);
            Vector v2 = new Vector(1, 1);
            Console.WriteLine(v1.Equals(v2));
        }
    }
}
```

Компьютерная игра



Для закрепления полученных знаний разработаем приложение – прототип игры «Астероиды».

Сначала создадим заготовку из двух классов: **Program** и **Game**. Для вывода графики воспользуемся формой из пространства имен **System.Windows.Form** (требуется подключить соответствующую библиотеку).

Класс Program

```
using System;
using System.Windows.Forms;
// Создаем шаблон приложения, где подключаем модули
namespace MyGame
{
    class Program
    {
        static void Main(string[] args)
        {
            Form form = new Form();
            form.Width = 800;
            form.Height = 600;
            Game.Init(form);
            form.Show();
            Game.Draw();
            Application.Run(form);
        }
    }
}
```

Класс Game

Это основной класс, где будут происходить все действия игры. Для вывода графики требуется подключить сборку **System.Drawing** и включить соответствующее пространство имен. Для вывода

графики на форму используется класс **Graphic**, который содержит методы для рисования на форме. Чтобы убрать мерцание в игре, будем выводить графику в промежуточный буфер. Когда графический кадр сформирован, выводим его на экран методом **Render**. Для получения графического буфера используется класс **BufferedGraphicsManager** и его свойство **Current**. Для связи буфера и графики применяем метод **Allocate**.

```
using System;
using System.Windows.Forms;
using System.Drawing;
namespace MyGame
{
    static class Game
    {
        private static BufferedGraphicsContext _context;
        public static BufferedGraphics Buffer;
        // Свойства
        // Ширина и высота игрового поля
        public static int Width { get; set; }
        public static int Height { get; set; }
        static Game()
        {
        }
        public static void Init(Form form)
        {
            // Графическое устройство для вывода графики
            Graphics g;
            // Предоставляет доступ к главному буферу графического контекста для
текущего приложения
            _context = BufferedGraphicsManager.Current;
            g = form.CreateGraphics();
             // Создаем объект (поверхность рисования) и связываем его с формой
            // Запоминаем размеры формы
            Width = form.ClientSize.Width;
            Height = form.ClientSize.Height;
            // Связываем буфер в памяти с графическим объектом, чтобы рисовать в
буфере
            Buffer = _context.Allocate(g, new Rectangle(0, 0, Width, Height));
        }
        public static void Draw()
        {
            // Проверяем вывод графики
            Buffer.Graphics.Clear(Color.Black);
            Buffer.Graphics.DrawRectangle(Pens.White, new Rectangle(100, 100, 200,
200));
            Buffer.Graphics.FillEllipse(Brushes.Wheat, new Rectangle(100, 100, 200,
200));
            Buffer.Render();
        }

    }
}
```

В качестве демонстрации реализуем задний фон игры. Чтобы отработать навык программирования с использованием ООП, создадим иерархию объектов.

Создадим класс **BaseObject**, в котором зададим начальное поведение некоторых объектов. Пусть это будут круги, которые при достижении края формы меняют направление движения.

```
using System;
using System.Drawing;

namespace MyGame
{
    class BaseObject
    {
        protected Point Pos;
        protected Point Dir;
        protected Size Size;
        public BaseObject(Point pos, Point dir, Size size)
        {
            Pos = pos;
            Dir = dir;
            Size = size;
        }
        public void Draw()
        {
            Game.Buffer.Graphics.DrawEllipse(Pens.White, Pos.X, Pos.Y,
Size.Width, Size.Height);
        }
        public void Update()
        {
            Pos.X = Pos.X + Dir.X;
            Pos.Y = Pos.Y + Dir.Y;
            if (Pos.X < 0) Dir.X = -Dir.X;
            if (Pos.X > Game.Width) Dir.X = -Dir.X;
            if (Pos.Y < 0) Dir.Y = -Dir.Y;
            if (Pos.Y > Game.Height) Dir.Y = -Dir.Y;
        }
    }
}
```

Внесем изменения в класс с игрой. Здесь создадим массив объектов **BaseObject**. Чтобы не загромождать метод **Init**, добавим дополнительно метод **Load**, в котором реализуем инициализацию наших объектов:

```
public static BaseObject[] _objs;
public static void Load()
{
    _objs = new BaseObject[30];
    for (int i = 0; i < _objs.Length; i++)
        _objs[i] = new BaseObject(new Point(600, i * 20), new Point(15 - i, 15 -
i), new Size(20, 20));
}
```

Нужно добавить вызов метода **Load** в **Init**.

Добавим в метод **Draw** вывод всех этих объектов на экран, а также добавим метод **Update** для изменения состояния объектов.

```
public static void Draw()
{
    // Проверяем вывод графики
    Buffer.Graphics.Clear(Color.Black);
    Buffer.Graphics.DrawRectangle(Pens.White, new Rectangle(100, 100, 200,
200));
    Buffer.Graphics.FillEllipse(Brushes.Wheat, new Rectangle(100, 100, 200,
200));
    Buffer.Render();

    Buffer.Graphics.Clear(Color.Black);
    foreach (BaseObject obj in _objs)
        obj.Draw();
    Buffer.Render();
}

public static void Update()
{
    foreach (BaseObject obj in _objs)
        obj.Update();
}
```

Добавим в **Init** таймер и обработчик таймера, в котором заставим вызываться **Draw** и **Update**.

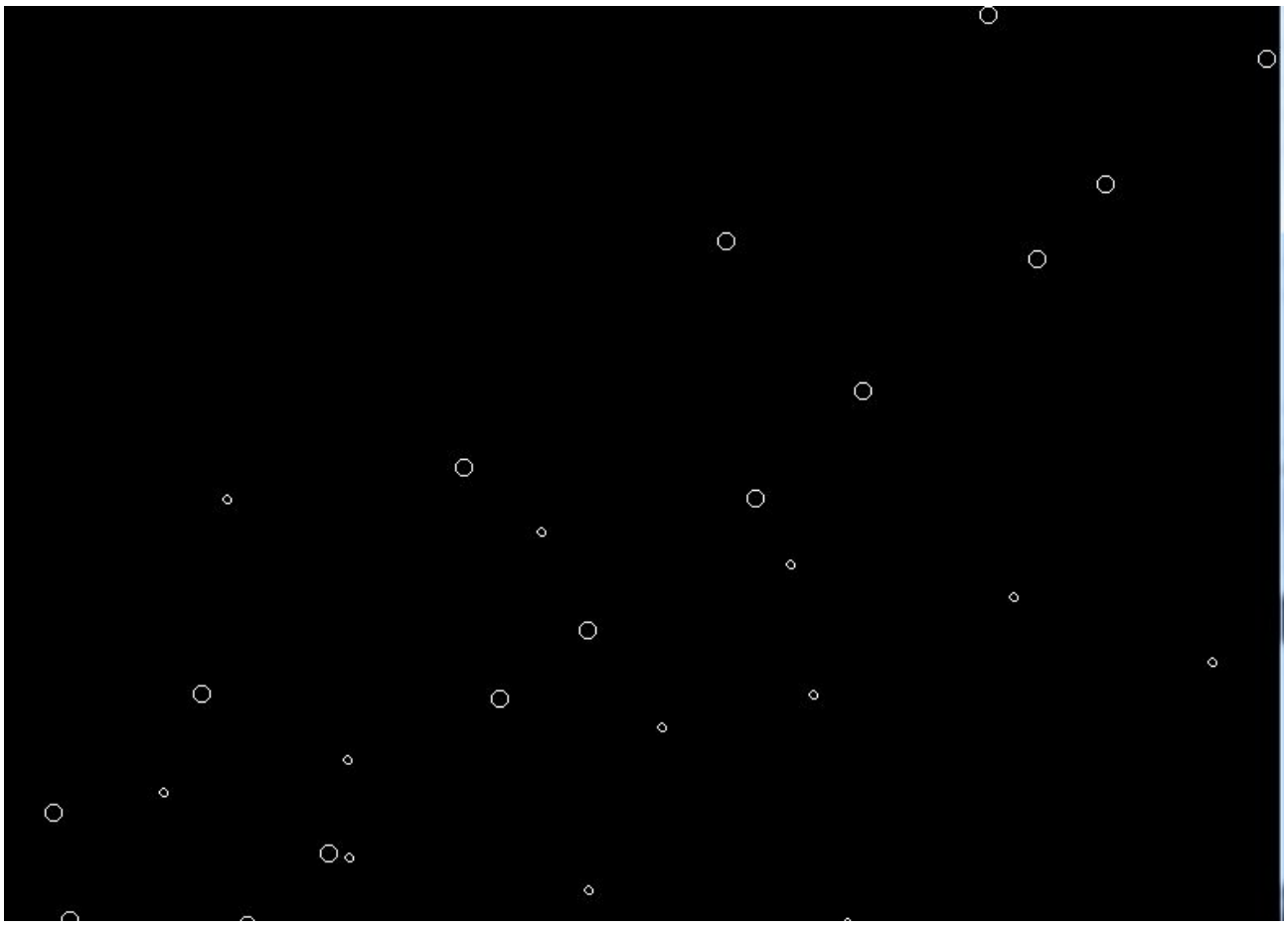
```
Timer timer = new Timer {Interval = 100};
timer.Start();
timer.Tick += Timer_Tick;
```

Обработчик таймера:

```
private static void Timer_Tick(object sender, EventArgs e)
{
    Draw();
    Update();
}
```

Более подробно про события и обработчики событий мы поговорим на следующих уроках.

Если вы все сделали правильно, можете насладиться движением объектов на экране. Поэкспериментируйте с программой, меняя их направление, размеры и положение.



Познакомимся на практике с полиморфизмом: создадим на базе этого объекта другие. Начнем со звезд.

Объект Star

Создадим класс **Star**, который будет наследовать **BaseObject**.

```
class Star: BaseObject
    {
    }
```

Компилятор подчеркнет **Star** красным цветом – мы должны создать конструктор, который будет передавать параметры базовому объекту, чтобы создать его.

Чтобы не писать конструктор полностью заново, воспользуемся ключевым словом **base**:

```
class Star: BaseObject
{
   public Star(Point pos, Point dir, Size size):base(pos,dir,size)
   {
   }
}
```

Этот объект идентичен предыдущему, но важна одна особенность. Поэтому в методе **Load** будем создавать экземпляры не **BaseObject**, а **Star**:

```
_objs = new BaseObject[30];
for (int i = 0; i < _objs.Length; i++)
    _objs[i] = new Star(new Point(600, i * 20), new Point(-i, 0), new Size(20,
20));
```

В начале программы массив должен остаться **static BaseObject[] objs**. Здесь нужно понять, что мы можем присваивать базовым объектам объекты-потомки. Почему это важно, разберем далее.

Теперь нужно добиться, чтобы при вызове **Draw** и **Update** каждый объект вел себя по-разному. Для этого в базовом объекте оба метода обозначим виртуальными, добавив в начале слово **virtual**:

```
public virtual void Draw()
public virtual void Update()
```

Это пока никак не влияет на поведение, так как в **Star** вызываются наследуемые методы. Переопределим их.

Переопределяем метод Draw и запускаем его. Теперь объект выводится на экран по-другому:

```
public override void Draw()
{
    Game.Buffer.Graphics.DrawLine(Pens.White, Pos.X, Pos.Y, Pos.X + Size.Width,
Pos.Y + Size.Height);
    Game.Buffer.Graphics.DrawLine(Pens.White, Pos.X + Size.Width, Pos.Y, Pos.X,
Pos.Y + Size.Height);
}
```

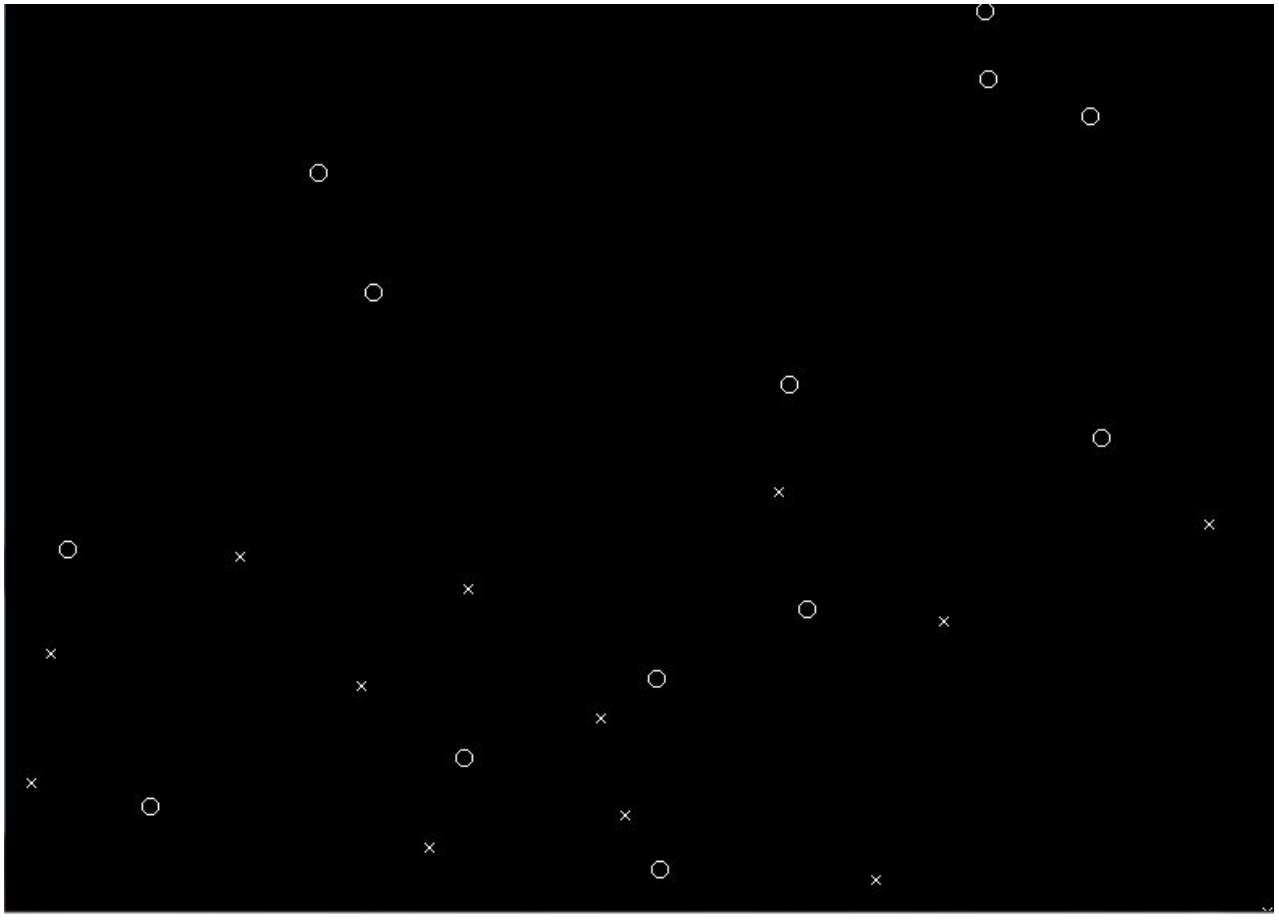
Теперь переопределяем метод **Update**:

```
public override void Update()
{
    Pos.X = Pos.X - Dir.X;
    if (Pos.X < 0) Pos.X = Game.Width + Size.Width;
}
```

Подразумевается, что звезды будут двигаться справа налево, поэтому мы меняем только координату по **X**. Если звезда уехала за пределы экрана, возвращаем ее с правой стороны.

Для более наглядной демонстрации полиморфизма заполним массив объектов различными фигурами и убедимся, что они отрисовываются и ведут себя соответствующим образом.

```
public static void Load()
{
    _objs = new BaseObject[30];
    for (int i = 0; i < _objs.Length / 2; i++)
        _objs[i] = new BaseObject(new Point(600, i * 20), new Point(-i, -i), new
Size(10, 10));
    for (int i = _objs.Length / 2; i < _objs.Length; i++)
        _objs[i] = new Star(new Point(600, i * 20), new Point(-i, 0), new Size(5,
5));
}
```



Советы

Правила для названий классов и методов

Обычно классы представляют объекты, а методы – действия. Поэтому для наименования классов используйте существительные – Cat, Machine, Girl. При создании методов используйте глаголы в следующей нотации – MoveLeft, ShowMessage. Так вам и другим программистам будет проще ориентироваться в коде.

Практическое задание

1. Добавить свои объекты в иерархию объектов, чтобы получился красивый задний фон, похожий на полет в звездном пространстве.
2. * Заменить кружочки картинками, используя метод **DrawImage**.

Дополнительные материалы

1. yield (справочник по C#);
2. Полный список перегружаемых операторов: документация msdn;
3. https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/structs;
4. https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/nullable-types/using-nullable-types.

Используемая литература

При создании данного методического пособия были использованы следующие ресурсы:

1. Татьяна Павловская. Программирование на языке высокого уровня. – 2009 г.
2. Эндрю Троелсен. Язык программирования C# 5.0 и платформа .NET 4.5. – 2013 г.
3. Герберт Шилдт. C# 4.0. Полное руководство.
4. MSDN.