Объектно - Ориентированное Программирование

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

[Объектно-ориентированное программирование](#_p22c3ghmnqfu)

[Класс - объект](#_c713xnuw3xem)

[Инкапсуляция](#_k2m9izuaozd0)

[Конструкторы](#_bt9jl8bywodg)

[Методы](#_rw0gbgarr3tz)

[Свойства](#_rdcoel8gu1g2)

[Автоматические свойства](#_wo95tmoloxe9)

[Перегрузка операторов](#_5s81oi63sade)

[Класс Vector.](#_xyrciluiwgn0)

[Структуры](#_rse08c91xm0z)

[Наследование](#_921y4z1zqlxc)

[Наследование включением (агрегация)](#_8h67y3yhzt9i)

[Виртуальный метод](#_8xk8ye80ncfg)

[ToString()](#_ogv1bvxknqgd)

[Полиморфизм](#_twn4hiyci00t)

[IS и AS](#_b62jeuhpzu0)

[\*Раннее и позднее связывание](#_hq1c2phk2i3b)

[Практика](#_16rjjaad2a5u)

[ToString()](#_ehm5znpunas9)

[Перегрузка Equals](#_xly8imgqf8z2)

[Компьютерная игра](#_syknqmhb9vqq)

[Класс Program](#_38j1ap2n5967)

[Класс Game](#_k209ol6mkp8p)

[Объект Star](#_so8bkg1ivdi0)

[Советы](#_ygi3s9odzato)

[Правила для названий классов и методов](#_i79534j6p1pc)

[Домашнее задание](#_3tfrjxxltv85)

[Используемая литература](#_9qfw16igergc)

# 

# 

# 

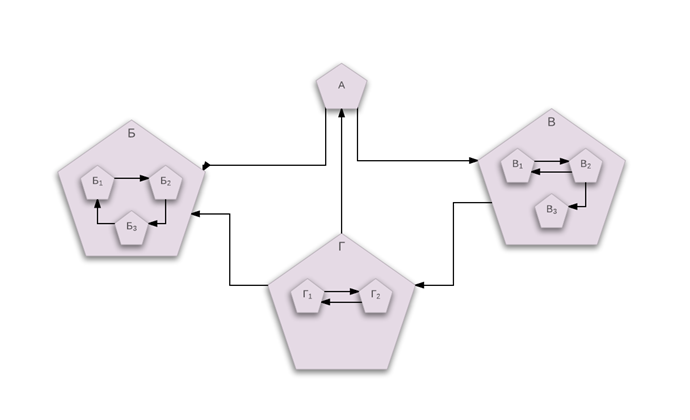
“Не паникуйте раньше времени! Все со временем станет понятным”

Берн Страуструп  
“Язык программирования C++”

# Объектно-ориентированное программирование

Как отмечал известный нидерландский программист Эдсгер Дейкстра, человечество ещё 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 переформулировывать, представляя все данные в виде переменных, массивов, списков и других структур данных. При моделировании больших систем объём этих данных увеличивается, они становятся плохо управляемыми, и это приводит к большому числу ошибок. Так как любой алгоритмы может обратиться к любым глобальным (общедоступным) данным, повышается риск случайного недопустимого изменения каких-то значений.

Поскольку формулировка задач, решаемых на компьютерах, всё более приближается к формулировкам реальных жизненных задач, возникла такая идея: представить программу в виде множества объектов (моделей), каждый из которых 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 идёт 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, который должен написать соответствующую часть программы, т.е. определить, как именно объект будет выполнять свои функции. При этом конкретному 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**

# 

Пример ООП на примере приложения логической игры “Удвоитель” в Windows Forms. Приложение состоит их объектов Windows, Label, Button (разработанные другими программистами) и некоторых других. У каждого объекта есть предназначение: Window - содержит на себе другие элементы. Button - для отображения надписей на кнопке и реализации нажатия на кнопке. Label - для реализации отображения надписи. Программист, разрабатывая приложение, связывает эти объекты между собой.

# Класс - объект

Инкапсуляция — это механизм программирования, объединяющий вместе код и данные, которыми он манипулирует, исключая как вмешательство извне, так и неправильное использование данных. Например, в данном примере мы создали класс Vector, который содержит в себе 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 мы можем описать объект v1 класса Vector или другие объекты класса, каждый из которых будет содержать в себе свои 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 упрощаем себе на первых порах жизнь, но нарушаем одно из правил объектно-ориентированного программирования. Это правило требует, чтобы доступ к внутренней структуре объекта извне было невозможно получить. Это может удивить, а как же тогда изменять данные объекта? И почему вообще нужно закрывать данные?

Данные нужно закрывать, потому что программист, который описывает некоторый класс (или лучше говорить поведение будущего объекта), должен сделать, чтобы объект вёл себя правильно. Если же дать возможность изменять данные напрямую, то это может дать возможность заполнить данные объекта не правильными данными, которые в этом объекте быть не могут. Например, попробуйте создать:

|  |
| --- |
| DateTime date = new DateTime(2016, 10, 20); |

здесь мы создаем объект, заполнив правильными данными.

|  |
| --- |
| DateTime date = new DateTime(2016, 15, 40); |

а здесь, пытаемся записать 15 месяц и 40 число.

Для того, чтобы управлять данными внутри объекта, существуют различные технологии ООП. Рассмотрим некоторые из них.

## Конструкторы

Конструктор - это специальный метод, который вызывается при создании экземпляра объекта. В .Net Framework, если в созданном вами классе нет конструктора, описанного вами, то создаётся конструктор без параметров, который заполняет поля объекта данными по умолчанию (0,false,null). Если мы хотим, чтобы при создании объекта объект был заполнен какими-то данными, то мы создаём конструктор с параметрами.

|  |
| --- |
| using System;  namespace Class\_Vector0010  {  class Vector  {  // Теперь поля приватные  double x;  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  double x;  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;  }  public double GetX()  {  return x;  }  public void SetX(double value)  {  x = value;  }  public double GetY()  {  return y;  }  public void SetY(double value)  {  y = value;  }  // Метод для получения данных в строковой форме  public string ToString()  {  return String.Format("X={0} Y={1}", x, 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:{0}", v1.ToString());  Console.WriteLine("v2:{0}", 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  double x;  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 { return x; }  set { x = value; }  }  // Свойство Y для доступа к полю y  public double Y  {  get { return 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 логичными, как при записи  v1.X = 10;  v2.X = -10;  Console.WriteLine("v1: X={0} Y={1}", v1.X, v1.Y);//так и при чтении  }  }  } |

При описании свойств используются акцессоры доступа get и set. С их помощью программист управляет возможностями читать или записывать данные.

### Автоматические свойства

Начиная с версии C# 3.0, появилась возможность для реализации очень простых свойств, не прибегая к явному определению переменной, которой управляет свойство. Вместо этого базовую переменную для свойства автоматически предоставляет компилятор. Такое свойство называется автоматически реализуемым и принимает следующую общую форму:

|  |
| --- |
| тип имя { get; set; } |

где 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 создать автоматически переменную, иногда ещё называемую поддерживающим полем, для хранения значения. Такая переменная недоступна непосредственно и не имеет имени. Но в то же время она может быть доступна через свойство.

Пример описания автоматического свойства

|  |
| --- |
| public string Name { get; set; } public int CustomerID { get; set; } |

# Перегрузка операторов

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

## Класс Vector.

|  |
| --- |
| using System;  namespace Vectors  {  // введём структуру Vector и перегрузим для неё операции +,-,=  class Vector  {  double x;  double y;  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 - значит мы указываем, что есть потеря данных  public static explicit operator Vector(double x)  {  return new Vector(x, x);  }  public static explicit operator double(Vector x)  {  return x.x;  }  // переопределение метода ToString  public string ToString()  {  return String.Format("X=" + x + " Y=" + y);  }  // перегрузка бинарного оператор +  public static Vector operator +(Vector v1, Vector v2)  {  Vector res = new Vector();  res.x = v1.x + v2.x;  res.y = v1.y + v2.y;  return res;  }  // перегрузка бинарного оператора -  public static Vector operator -(Vector v1, Vector v2)  {  Vector res = new Vector();  res.x = v1.x - v2.x;  res.y = v1.y - v1.y;  return res;  }  // перегрузка унарного оператора -  public static Vector operator -(Vector v1)  {  Vector res = new Vector();  res.x = -1\*v1.x;  res.y = -1\*v1.y;  return res;  }  // свойство, которое возвращает закрытое поле X  public double X  {  get { return x; }  }  // свойство, которое возвращает закрытое поле Y  public double Y  {  get { return y; }  }  }  class Program  {  static void Main()  {  // создаём вектор  Vector v1 = new Vector(-5, 5);  // другой вектор задаём, используя перегрузку = explict  Vector v2 = (Vector)10;  Vector v3 = v1 + v2;// проверяем работу +  // демонстрация доступа к закрытым полям через свойства  Console.WriteLine("v1.x={0} v1.y={1}", v1.X,v1.Y);  Console.WriteLine("v2:{0}", v2);  Console.WriteLine("-(v1+v2):{0}",-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 некоторые отличия. Главные - это то, что структуры относятся к типам значений и не поддерживают наследования.

Пример структуры:

|  |
| --- |
| using System;  namespace Struct\_Vector0010  {  struct Vector  {  // Теперь поля приватные  public double x,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)  {  Vector v1;// Для структур не нужно использовать ключевое слово new  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 также, как и для классов, возможно переопределять операторы +,-,= и др.

Выбор между структурой и классом приходится делать программисту. В случаях, когда модель содержит небольшое количество данных внутри себя (как в примере с Vector), то модель бывает выгодней описать в виде структуры, так как на работу со структурой не тратится время на выделение памяти в куче.

# Наследование

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

В языке С# класс, который наследуется, называется базовым, а класс, который наследует, — 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 свои 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#, или правильнее 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, программисты стараются избегать его, так как приходится заботиться о том, какие данные хранятся в переменных типа object.

Пример наследования:

|  |
| --- |
| using System;  namespace inheritance\_0020  {  class Vector  {  // Теперь поля приватные  double x;  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 { return x; }  set { x = value; }  }  // Свойство Y для доступа к полю y  public double Y  {  get { return y; }  set { y = value; }  }  // Метод для получения данных в строковой форме  public string ToString()  {  return String.Format("X={0} Y={1}", x, y);  }  }  class MyObject: Vector  {  }  class Program  {  static void Main(string[] args)  {  MyObject obj1 = new MyObject();  obj1.X = 10;  obj1.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 не приватными полями, методами и свойствами базового класса.

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

## Наследование включением (агрегация)

Альтернативным наследованию механизмом использования одним классом другого является вложения, когда один класс является полем другого.

|  |
| --- |
| using System;  namespace Inheritance\_Incloser\_0010  {  // Пример наследования  class Vector  {  // Теперь поля приватные  double x;  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 { return x; }  set { x = value; }  }  // Свойство Y для доступа к полю y  public double Y  {  get { return y; }  set { y = value; }  }  // Метод для получения данных в строковой форме  public string ToString()  {  return String.Format("X={0} Y={1}", x, y);  }  }  class MyObject  {  public Vector pos;  double width, 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()  {  return String.Format("width:{0} height:{1} ", width, 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, то лучше использовать наследование. Если же внутри класса вам необходимо использовать несколько объектов другого класса, то необходимо включение. Понимание, какое наследование использовать, приходит с опытом.

# Виртуальный метод

Виртуальным называется такой метод, который объявляется как virtual в базовом классе.

Виртуальный метод отличается тем, что он может быть переопределён в одном или нескольких производных классах.

Для переопределения используется override.

### ToString()

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

Это сделано специально для того, чтобы наследники (то есть, все классы) могли их переопределить и сделать их поведение более естественным для их объекта. Если их не переопределить, то вызывается поведение, заданное в System.Object, но оно не может знать о реализации вашего объекта и ведёт себя довольно примитивно. Например, не переопределенный (то есть, принадлежащий System.Object) ToString выводит информацию к какому классу и к какому пространству имён принадлежит созданный вами класс.

Для того, что бы правильно переопределить поведение ToString, мы должны добавить слово override перед описанием метода ToString в нашем классе. Без override мы скрываем базовое поведение и не даём будущим потомкам нашего класса использовать поведение заложенное в базовом классе.

|  |
| --- |
| using System;  namespace Override\_010  {  class MyObject  {  int a;  public MyObject(int \_a)  {  a = \_a;  }  // Запустите программу, закомментировав метод ToString  public override string ToString()  {  return "a=" + Convert.ToString(a);  }  }  class Program  {  static void Main(string[] args)  {  MyObject obj = new MyObject(2);  Console.WriteLine(obj.ToString());  }  }  } |

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

# 

# 

# Полиморфизм

Полиморфизм по-гречески означает "множество форм" — 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

Проверяет совместимость объекта с заданным типом. Например, 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 всё 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("obj1 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 выполнения определенных преобразований типов между совместимыми ссылочными типами или тип, допускающий значение NULL. Оператор as подобен оператору приведения: (int)x. Однако если преобразование невозможно, то 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 c obj как MyObject");  obj = obj2 as MyObject2;  if (obj != null) Console.WriteLine("Мы теперь можем обращаться к c obj как MyObject2");  }  } |

# \*Раннее и позднее связывание

Имеются два термина, часто используемых, когда речь заходит об объектно-ориентированных языках программирования: раннее и позднее связывание.

В терминах объектно-ориентированного программирования раннее связывание означает, что объект и вызов функции связываются между собой на этапе компиляции. Это означает, что вся необходимая информация для того, чтобы определить, какая именно функция будет вызвана, известна на этапе компиляции программы. В качестве примеров раннего связывания можно ука­зать 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. Его недостатком служит невысокая 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 использо­вания виртуальных методов и производных классов. Его достоинством является высокая гиб­кость. Оно может использоваться для поддержки общего интерфейса, позволяя при этом различным объектам иметь свою собственную реализацию этого интерфейса. Более того, оно помогает со­здавать библиотеки классов, допускающие повторное использование и расширение.

Рассмотрим как это реализовано в .NET Framework.

Предположим мы описали какой-то метод виртуальным:

|  |
| --- |
| virtual public void Draw() |

Объявление метода виртуальным означает, что все ссылки на этот метод будут разрешаться по факту его вызова, то есть не на стадии компиляции, а во время выполнения программы. Этот механизм и есть “позднее связывание”.

Для его реализации необходимо, чтобы адреса виртуальных методов хранились там, где ими можно будет в любой момент воспользоваться, поэтому компилятор формирует для этих методов таблицу виртуальных методов (Virtual Method Table, VMT). В неё 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. Обеспечение этой связи устанавливается во время выполнения программы при создании объекта.

# 

# 

# Практика

### ToString()

Простой практический пример использования полиморфизма:

|  |
| --- |
| using System;  namespace Override\_ToString  {  class MyClass: Object  {  int a;  public MyClass(int a)  {  this.a = a;  }  // Попробуйте раскомментировать этот метод и запустить программу  //public override string ToString()  //{  // return a.ToString();  //}  }  class Program  {  static void Main(string[] args)  {  MyClass obj = new MyClass(10);  Console.WriteLine(obj);  Console.ReadKey();  }  }  } |

## 

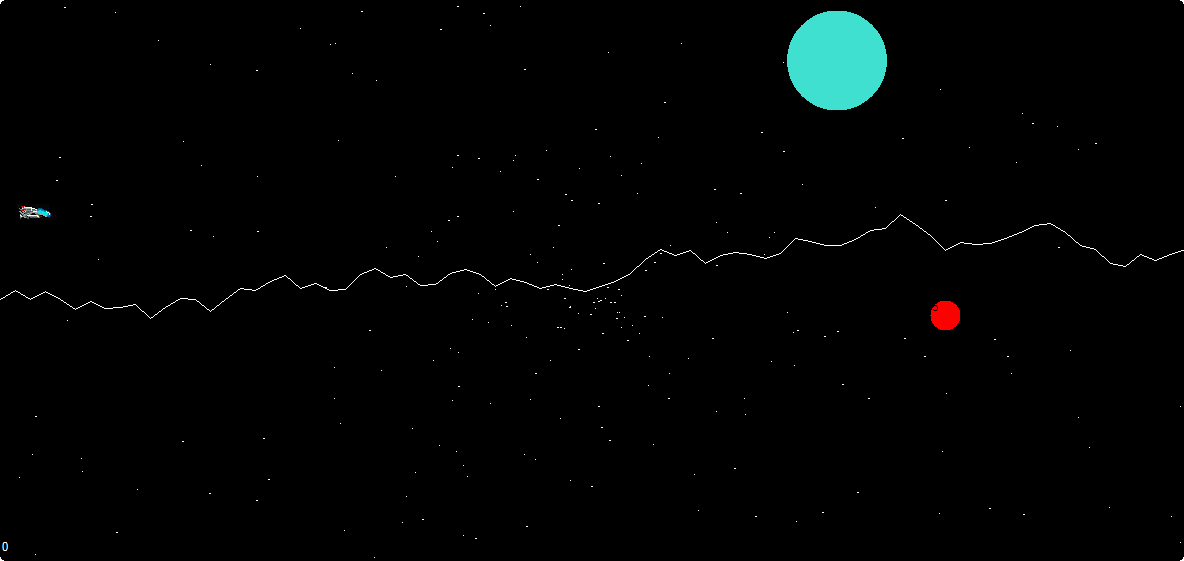
## 

## Перегрузка Equals

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

|  |
| --- |
| using System;  namespace OverrideEquals  {  class Vector  {  public double x, y;  public Vector(double x, double y)  {  this.x = x;  this.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”.

Для начала создадим заготовку из двух классов Program и Game.

Для вывода графики воспользуемся формой из пространства имён System.Windows.Form (требуется подключить соответствующую библиотеку)

### Класс Program

|  |
| --- |
| using System;  using System.Windows.Forms;  // Создаём 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 имён. Для вывода графики на форму используется класс Graphic. Этот класс содержит методы для рисования на форме. Но для того, чтобы убрать мерцание в игре, мы будет выводить графику в промежуточный буфер. После того, как графический кадр сформирован, мы будет выводить его на экран методом Render. Для получения графического буфера используется класс BufferedGraphicsManager и его свойство Current. Для связи буфера и графики метод Allocate.

|  |
| --- |
| using System;  using System.Windows.Forms;  using System.Drawing;  namespace MyGame  {  static class Game  {  static BufferedGraphicsContext context;  static public BufferedGraphics buffer;  // Свойства  // Ширина и высота игрового поля  static public int Width { get; set; }  static public int Height { get; set; }  static Game()  {  }  static public 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();// Создаём объект - поверхность рисования и связываем его с формой  // Запоминаем размеры формы  Width = form.Width;  Height = form.Height;  // Связываем буфер в памяти с графическим объектом.  // для того, чтобы рисовать в буфере  buffer = context.Allocate(g, new Rectangle(0, 0, Width, Height));  }  static public 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. Создадим для отработки навыка программирования с использованием ООП иерархию объектов.

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

|  |
| --- |
| using System;  using System.Drawing;  namespace Game\_000200  {  class BaseObject  {  protected Point pos;  protected Point dir;  protected Size size;  public BaseObject(Point pos, Point dir, Size size)  {  this.pos = pos;  this.dir = dir;  this.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;  }  }  } |

Внесём изменения в класс с игрой. Здесь мы с вами создадим массив объектов BaseObject.Чтобы не загромождать метод Init, создадим дополнительно метод Load, в котором реализуем инициализацию наших объектов.

|  |
| --- |
| static public 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.

|  |
| --- |
| static public void Draw()  {  buffer.Graphics.Clear(Color.Black);  foreach (BaseObject obj in objs)  obj.Draw();  buffer.Render();  }  static public void Update()  {  foreach (BaseObject obj in objs)  obj.Update();  } |

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

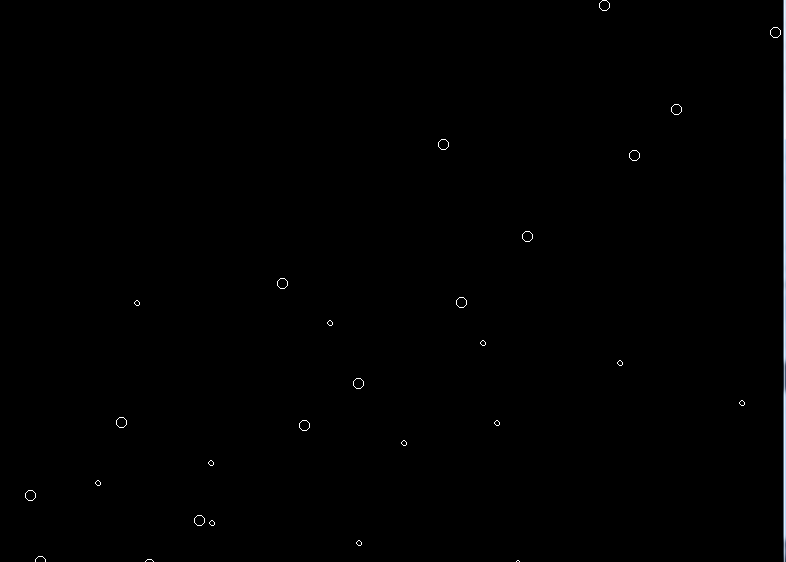
|  |
| --- |
| Timer timer = new Timer(); timer.Interval = 100; timer.Start(); timer.Tick += Timer\_Tick; |

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 всё сделали правильно, то можете насладиться наблюдением движущихся объектов по экрану.  
Поэкспериментируйте с программой, меняя направление движения, размеры и положения объектов на экране.



Теперь давайте познакомимся с полиморфизмом.

Создадим на базе этого объекта другие. Начнём со звезд.

### Объект Star

Создадим класс Star, который будет наследовать BaseObject.

|  |
| --- |
| class Star: BaseObject  {  } |

Компилятор подчеркнет Star красным цветом, так как 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 полностью идентичен предыдущему, но здесь важна одна особенность. Поэтому давайте 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, разберём далее.

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

|  |
| --- |
| virtual public void Draw() virtual public void Update() |

Это пока никак не влияет на поведение, так как в Star вызываются наследуемые методы. Теперь давайте переопределим эти методы.

Для начала переопределите метод Draw и запустите. 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); } |

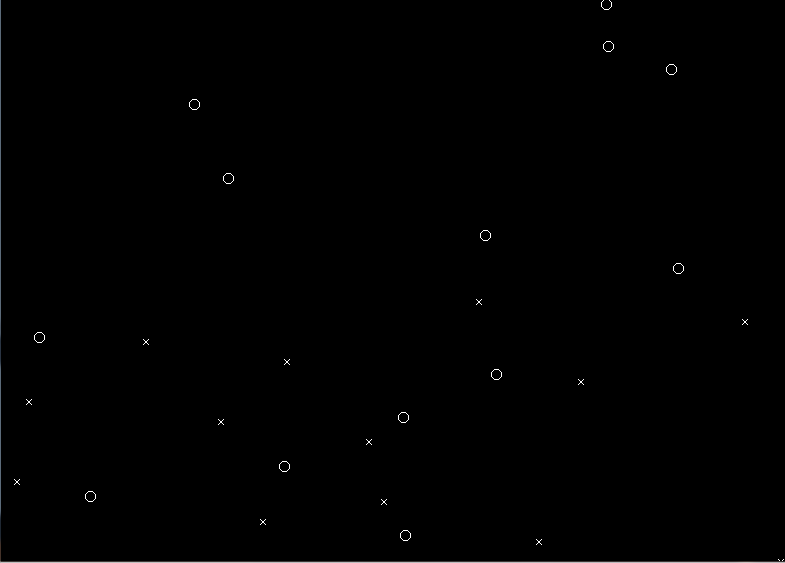
Теперь переопределите метод Update.

|  |
| --- |
| public override void Update() {  pos.X = pos.X + dir.X;  if (pos.X < 0) pos.X = Game.Width + size.Width; } |

Подразумевается, что звёзды будут двигаться справа налево, поэтому мы меняем только координату по X, а если звезда уехала за пределы экрана, то мы её возвращаем с правой стороны.

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

|  |
| --- |
| static public 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 при наименовании классов или методов придерживайтесь следующего правила. Для наименования классов используйте имена существительные — Cat, Machine, Girl. При создании методов используйте глаголы в следующей нотации — MoveLeft, ShowMessage и т. д. В этом случае вам и другим программистам будет проще ориентироваться в коде.

# 

# 

# Домашнее задание

1. Добавить свои объекты в иерархию объектов, чтобы получился красивый задний фон, похожий на полёт в звёздном пространстве.
2. \*Заменить кружочки картинками, используя метод DrawImage.
3. \*\*Разработать собственный класс Screensaver, аналогичный классу Game.

# Используемая литература

При создании данного методического пособия были использованы следующие ресурсы:

1. Т.А. Павловская. “Программирование на языке высокого уровня”, 2009 г.
2. “Язык программирования C# 5.0 и платформа .NET 4.5”. Эндрю Троелсен, Питер 2013 г.
3. Г.Шилдт. “C# 4.0. Полное руководство”.
4. [MSDN](https://msdn.microsoft.com/ru-ru/default.aspx).