Наследование

Наследование в C#

Анализ механизма наследования в C#.
Спецификаторы доступа при наследовании.
Особенности использования конструкторов при наследовании.
Сокрытие имен при наследовании.
Ключевое слово base.
Наследование и исключения.
Наследование от стандартных классов исключений.

Наследование – механизм ООП, позволяющий описать новый класс на основе уже существующего (базового), при этом данные и функциональность базовго класса заимствуются новым классом .

Главная задача наследования - обеспечить повторное использование кода.

Существует два основных вида наследования:

- классическое наследование;
- включение-делегирование;

Наследуются:

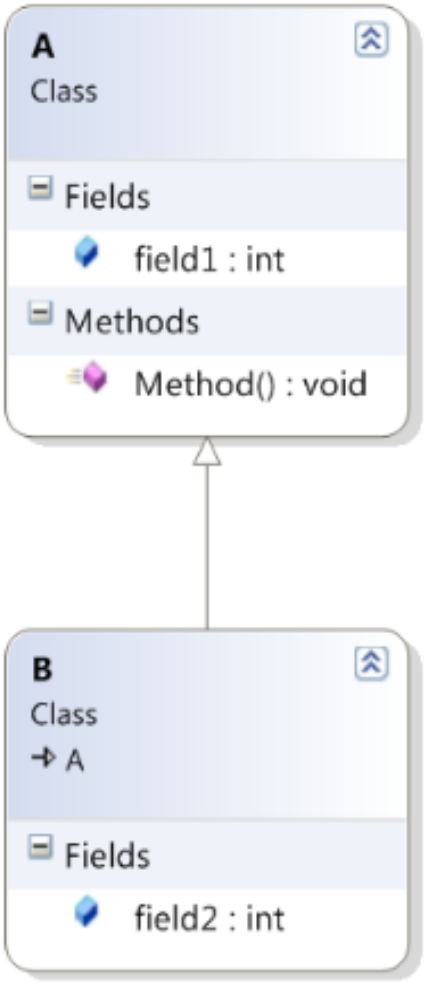
1. поля
2. методы
3. свойства
4. операторы
5. индексаторы
6. события

НЕКОТОРЫЕ ОСОБЕННОСТИ НАСЛЕДОВАНИЯ

- ✓ Объект производного класса содержит все члены базового класса !!!
- ✓ Производный класс имеет прямой доступ ко всем public полям, методам и свойствам базового класса
- ✓ Элементы базового класса с модификатором private не доступны в производном классе.
- ✓ Конструкторы базового класса не переносятся в производный класс.
- ✓ Наследование от двух и более классов в C# запрещено.
- ✓ Ни один класс не может быть базовым (ни прямо, ни косвенно) для самого себя.

Чтобы указать, что один класс является наследником другого, используется следующий синтаксис:

```
class имя_производного_класса: имя_базового_класса
   {
     //  тело производного класса
   }
```




```
class A
{
    public int field1;
    public void Method()
    {
        /* … */
    }
}


class B : A
{
    public int field2;
}
```

```
static void Main()
{
    B b = new B();
    b.field1=5;
    b.field2=8;
    b.Method();
}
```

```
 // класс транспортное средство
class Vehicle
{
 public string Type {get; set;} // тип транспортного средства
 public string Name {get; set;} // название транспортного средства
 public double Speed {get; set;}// скорость транспортного средства
 public string GetInfo()
 {
   return String.Format("{0} {1}, скорость: {2}",Type, Name, Speed );
 }
}
```

```
// класс грузовик
class Truck : Vehicle        // наследование
{
    public double Load {get; set;}     // грузоподъемность
}
```

```
static void Main(string[] args)
{
 Vehicle vehicle = new Vehicle();
 vehicle.Type = "Транспортное средство";
 vehicle.Name = "Обобщенное транспортное средство";
 vehicle.Speed = 150;
 // vehicle.Load = 2.8;                  // ОШИБКА
 Console.WriteLine(vehicle.GetInfo());

Truck truck = new Truck();
truck.Type = "Грузовик ";                // наследование открытых полей
truck.Name = «МАЗ";
truck.Speed = 90;
truck.Load = 3.8;
Console.WriteLine(truck.GetInfo());  // наследование открытых методов
}
```

Вызов конструкторов!

Вызов конструкторов базового класса

Производный класс может вызывать конструктор, определенный в его **базовом классе**, используя **расширенную форму объявления** конструктора производного класса и ключевое слово **base**.

Формат расширенного объявления таков:

```
конструктор_производного_класса (список_параметров)
                                    : base (список_аргументов)
{
    // тело конструктора
}
```

С помощью элемента **список_аргументов** задаются аргументы, необходимые конструктору в базовом классе.

Вызов конструкторов базового класса

```
public class  SomeType
{
}

// конструктор по умолчанию
public class SomeType
{
  public SomeType() : base()
   {
   }
}
```

Порядок вызова конструкторов

- в иерархии классов конструкторы вызываются по порядку выведения классов: от базового к производному;
- если в иерархии классов конструктору базового класса требуются параметры, то **все производные классы** должны предоставлять эти параметры вверх по иерархии, независимо от того, требуются они самому производному классу или нет

Особенности использования конструкторов при наследовании

```
class Vehicle
{ public string Type {get; set;}
  public string Name {get; set;}
  public double Speed {get; set;}
  public string GetInfo()
  { return String.Format("{0} {1},: {2}", Type, Name, Speed ); }

  // конструктор базового класса (конструкторы не наследуются)
  public Vehicle(string T, string N, double V)
  {        this.Type = T; this.Name = N; this. Speed = V;
  }
}
```

```
class Truck : Vehicle
{
  public double Load {get; set;}
  // Конструктор
  public Truck(string T, string N, double V, double Load)
         :base(T,N,V)
    {
        this.Load = Load;
    }
```

Доступ к членам базового класса

Наследование класса не отменяет ограничения, связанные с **закрытым доступом (private)**.
Производный класс включает все члены базового класса, но он не может получить **прямой доступ** к закрытым членам.

Использование защищенного доступа

Защищенный член создается с помощью модификатора доступа **protected**.

Когда **защищенный** член наследуется, то он становится **доступным для производного класса**.

Спецификаторы доступа при наследовании

```
 class Vehicle
{
 string Type;
 string Name;
 protected double Speed ;  // (модификатор доступа защищенный)
 public string GetInfo()
  { return String.Format("{0} {1}, : {2}", Type, Name, Speed ); }
 public Vehicle(string T, string N, double V)
  { this.TypeV = T; this.Name = N; this.Speed = V; }
}
```

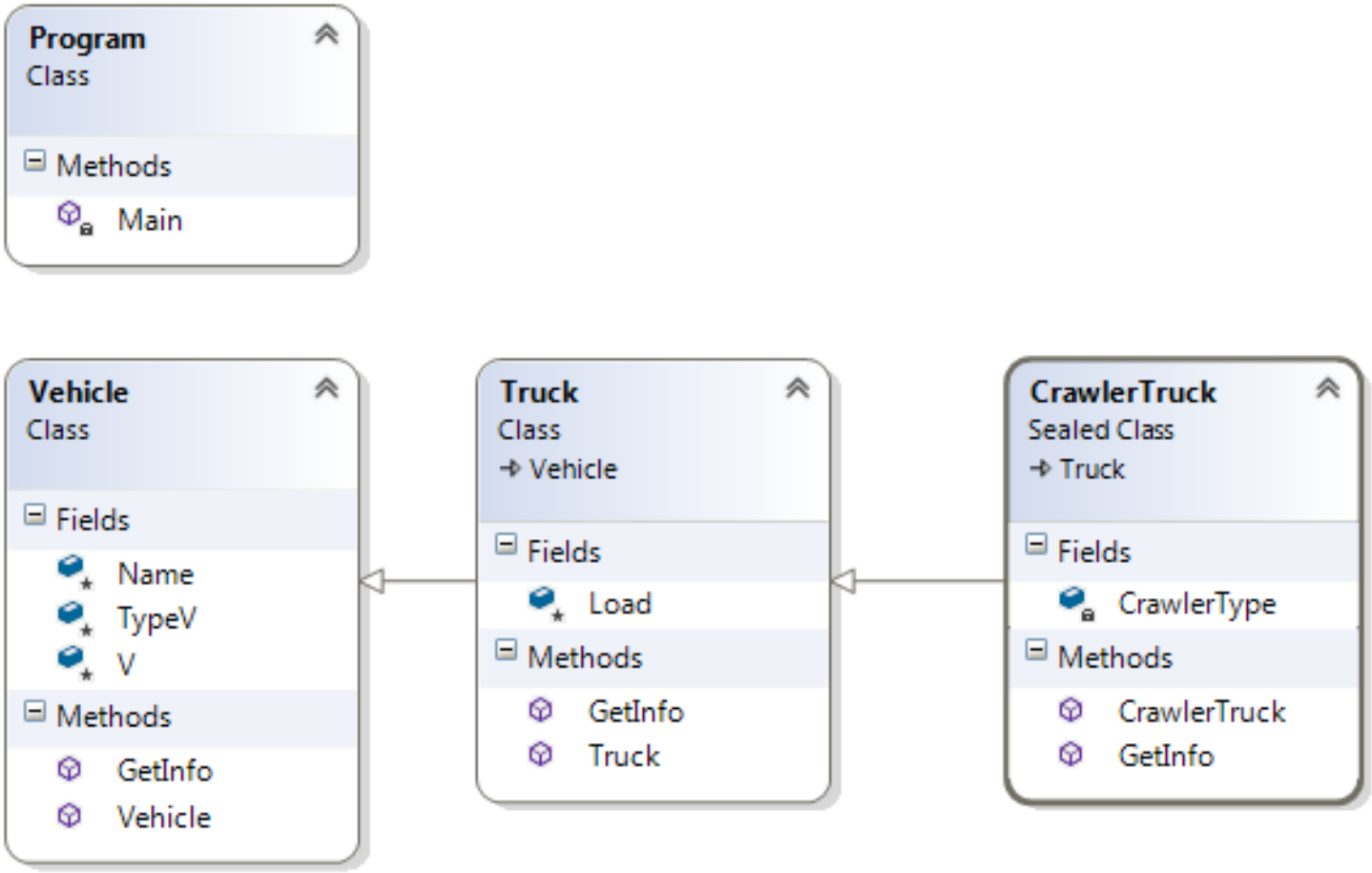
```
class Truck : Vehicle
{
  double Load;
  public Truck(string T, string N, double V, double Load)
        : base(T, N, V)
   {   this.Load = Load;       }
  public void  UpdateTruck()
   {  // truck.Type = "Большой грузовик";  // ОШИБКА т.к.  (private)
      Speed = 100;                         // ОШИБКИ НЕТ!
   }
}
```

Сокрытие имен при наследовании

```
class Vehicle
{ protected string TypeV;
  protected string Name;
  protected double Speed;
  public Vehicle(string T, string N, double V)
  {  this.TypeV = T; this.Name = N; this.Speed = V;   }
  public string GetInfo()
  {  return String.Format("{0} {1}, : {2}", TypeV, Name, Speed); }
}
```

```
class Truck : Vehicle
{ double Load;
  public Truck(string T, string N, double V, double Load): base(T,N,V)
  {  this.Load = Load;    }
// метод с одинаковым именем, как в базовом классе
// для исключения предупреждения о сокрытии имен используется new
new  public  string   GetInfo()
{
    // вызов варианта метода GetInfo() из базового класса
    // с использованием ключевого слова base
    return base.GetInfo() + " грузоподъемность: " + Load; }
} }
```

// Создать диаграмму классов: На проекте View->View Class Diagramm



Создание специальных исключений

Для получения специального исключения необходимо создать новый класс, унаследованный от класса **System.Exception** или **System.ApplicationException**.
(по соглашению, имена всех классов исключений оканчиваются суффиксом Exception).

- наследоваться от **ApplicationException**;
- сопровождаться атрибутом **[System.Serializable]**;
- иметь **конструктор по умолчанию**;
- иметь конструктор, который устанавливает значение унаследованного **свойства Message**;
- иметь конструктор для обработки “**вложенных исключений**";
- иметь конструктор для обработки **сериализации типа**.

```
// атрибут для сериализации (далее в курсе)
[Serializable]
public class TruckException : Exception
{
 // конструкторы
 public TruckException()
 {         }

 // конструкторы для инициализации св-ва Message
 public TruckException(string message) : base(message)
 {         }

 public TruckException(string message, Exception ex) : base(message)
 {         }

// Конструктор для обработки сериализации типа
protected TruckException(SerializationInfo info,
                                        StreamingContext contex)
  : base(info, contex)
  {          }
}
```

```
class Truck
{
     private double load;              // грузоподъемность
     private double loadMax=5000;      // макс. грузоподъемность
     // В свойстве генерируется исключение  TruckException
     public  double Load
      {  get { return Load; }
           set
         { load += value;
           if (load>=loadMax)
            throw new TruckException("Превышена грузоподъемность");
 }}}
```

```
try
{
   Truck truck = new Truck();
   truck.Load = 3000;
   truck.Load = 2500;
}
catch (TruckException ex)
{
    Console.WriteLine(ex.Message);
}
```

Использование ссылок на базовый класс.

C# - строго типизированный язык программирования

В C# имеется одно важное исключение: **переменной ссылке на объект базового класса может быть присвоена ссылка на объект любого производного от него класса**.

При этом доступ к конкретным членам класса определяется типом переменной ссылки на объект, а не типом объекта, на который она ссылается т.е. доступ разрешается только к тем частям этого объекта, которые определяются базовым классом.

Почему так? Поскольку базовому классу ничего не известно о тех членах, которые добавлены в производный от него класс.

```
class Vehicle
{
  protected string TypeV;
  protected string Name;
  protected double Speed;
  public Vehicle(string T, string N, double V)
  { this.TypeV = T; this.Name = N; this. Speed = V;  }

  public string GetInfo()
  { return String.Format("{0} {1}, : {2}", TypeV, Name, Speed);  }
}


class Truck : Vehicle
{  protected double Load;
   public Truck(string T, string N, double V, double Load)
    : base(T, N, V)
   {   this.Load = Load;   }

    public string GetInfo()
    {
        return base.GetInfo() + " грузоподъемность: " + Load;
    }
}
```

Использование ссылок на базовый класс.

```
class Program
 {   static void Main(string[] args)
  {
     Vehicle[] park = new Vehicle[3];
     park[0] = new Vehicle(“T/C", "Обобщенное ", 150);
     Console.WriteLine(park[0].GetInfo());

     // сохранение объекта производного класс в ссылке базового класса
     park[1] = new Truck(«T/C", "Грузовик", 90, 3.8);
     // вызов метода базового класса
     Console.WriteLine(park[1].GetInfo());

     // сохранение объекта производного класс в ссылке базового класса
     park[2] = new Bus(«T/C", "Автобус", 50, 7.4, 75);
     // вызов метода базового класса
     Console.WriteLine(park[2].GetInfo());
  }
}
```

Виртуальные методы

- **Что такое виртуальный метод?**
- **Необходимость использования виртуальных методов.**
- **Переопределение виртуальных методов.**

Абстрактный класс.

Виртуальным называется метод, объявляемый с помощью ключевого слова **virtual** в базовом классе и переопределяемый в одном или нескольких производных классах.

Чтобы объявить метод в базовом классе **виртуальным**, его объявление необходимо **обозначить ключевым словом virtual**. При **переопределении** виртуального метода в производном классе используется **модификатор override**.

Если базовый класс содержит виртуальный метод и из этого класса выведены производные классы, то будут выполняться различные версии этого виртуального метода.

При **вызове виртуального метода по ссылке на базовый класс** выполняется тот вариант виртуального метода, который переопределен в объекте, к которому происходит обращение по ссылке, причем это делается во время выполнения.

Вывод: вариант выполняемого виртуального метода выбирается по типу объекта, а не по типу ссылки на этот объект.

```
class Vehicle
{
  protected string TypeV;
  protected string Name;
  protected double Speed;
 public Vehicle(string T, string N, double V)
 { this.TypeV = T; this.Name = N; this. Speed = V;  }
 // virtual вирутальный метод
 public virtual string GetInfo()
 { return String.Format("{0} {1},: {2}", TypeV, Name, Speed);  }
}


class Truck : Vehicle
{  protected double Load;
   public Truck(string T, string N, double V, double Load)
    : base(T, N, V)
   {   this.Load = Load;   }

// переопределение виртуального метода из базового класса
public override string GetInfo()
{
    return base.GetInfo() + " грузоподъемность: " + Load;
}
```

```
Console.WriteLine("Виртуальный метод");

Vehicle[] park = new Vehicle[3];

park[0] = new Vehicle("Транспортное средство",
                      "Обобщенное транспортное средство", 150);
Console.WriteLine(park[0].GetInfo());

// сохранение объекта производного класс в ссылке базового класса
park[1] = new Truck("Транспортное средство", "Грузовик", 90, 3.8);

// вызов метода производного класса
// из ссылки на базовый т.к. виртуальный метод переопределен
Console.WriteLine(park[1].GetInfo());

 // сохранение объекта производного класс в ссылке базового класса
park[2] = new Bus("Транспортное средство", "Автобус", 50, 7.4, 75);
Console.WriteLine(park[2].GetInfo());
```

Особенности использования виртуальных методов

- виртуальный метод **не может** быть **static** или **abstract**
- переопределять виртуальный метод **не обязательно**.
- при наличии многоуровневой иерархии виртуальный метод не переопределяется в производном классе, то **выполняется ближайший** его вариант, обнаруживаемый **вверх по иерархии**,
- **свойства и индексаторы** также подлежат модификации ключевым словом **virtual** и переопределению ключевым словом **override**

1. Абстрактный метод создается с помощью модификатора типа abstract.
2. Абстрактный метод не содержит тела и не реализуется базовым классом.
3. Производный класс должен переопределить абстрактный метод.
4. Абстрактный метод автоматически является виртуальным.
5. Совместное использование модификаторов virtual и abstract считается ошибкой.
6. Модификатор abstract не может применяться в статических методах (static).
7. Абстрактными могут быть также индексаторы и свойства.

Для объявления абстрактного метода используйте следующий формат записи.

abstract ТИП ИМЯ (список_параметров);

Класс, содержащий один или несколько абстрактных методов, также должен быть объявлен как **абстрактный** с помощью спецификатора abstract , который ставится перед объявлением **class** .

- **Экземпляры абстрактного класса создавать нельзя**. Создание объекта приведет к ошибке во время компиляции.
- Когда производный класс **наследует абстрактный класс**, в нем **должны быть реализованы все абстрактные методы** базового класса.
- Если в производном классе **не реализован абстрактный метод** то данный класс должен быть определен как **abstract**. **Следовательно, abstract наследуется до тех пор, пока не будет достигнута полная реализация класса.**

```
// класс помечается abstract т.к. имеется абстрактный метод
abstract class Vehicle
{
    protected string TypeV;
    protected string Name;
    protected double Speed;
    public Vehicle(string T, string N, double V)
    {  this.TypeV = T; this.Name = N; this.Speed = V;    }
    // абстрактный  метод
    public abstract string GetInfo();
}


class Truck : Vehicle
{
    protected double Load;
    public Truck(string T, string N, double V, double Load)
        : base(T, N, V)
    {    this.Load = Load;   }
    // переопредение абстрактного метода из базового класса
    public override string GetInfo()
   { return String.Format("{0}{1}{2}{3}", TypeV, Name, Speed, Load);
}
}
```

Использование ключевого слова sealed.

- **sealed используется для запрета наследование класса,**
- класс не допускается объявлять одновременно как **abstract и sealed**
- sealed может быть также использовано в **виртуальных методах** для предотвращения их дальнейшего переопределения.

```
// Иерархия классов: Vehicle->Truck->CrawlerTruck
// sealed - запечатанный класс (наследоваться от него невозможно!)
sealed class CrawlerTruck : Truck
 {
    string CrawlerType;  // тип гусениц
    public CrawlerTruck(string T, string N, double crT)
       : base(T, N, crT)
     {
            this.CrawlerType = crT;
     }
     new public string GetInfo()
     {
            return base.GetInfo() + " тип гусениц: " + CrawlerType;
     }
}
```

Анализ базового класса Object.

Класс object считается базовым классом для всех остальных классов и типов, включая и типы значений.

Переменная ссылочного типа object может ссылаться на объект любого другого типа.

```
// массив объктов класса Object, который является
// базовым классов для всех типов!

Object[] park = new Object[2];
park[0] = new Truck("Т/С", "Грузовик", 90, 3.8);
park[1] = new Bus("Т/С", "Автобус", 50, 7.4, 75);
foreach(Object obj in park)
{
    Console.WriteLine(obj);
}
```

Анализ базового класса Object.

```
public class Object
    {
        public Object();
        public virtual bool Equals(object obj);
        public static bool Equals(object objA, object objB);
        public virtual int GetHashCode();
        public Type GetType();
        protected object MemberwiseClone();
        public static bool ReferenceEquals(object objA, object objB);
        public virtual string ToString();
    }
```

ToString()

Метод ToString() возвращает символьную строку, содержащую описание того объекта, для которого он вызывается. Этот метод переопределяется во многих классах, что позволяет приспосабливать описание к конкретным типам объектов, создаваемых в этих классах.

GetHashCode()

Этот метод используется, когда объект помещается в структуру данных, известную как хеш-таблица или словарем (dictionary). Применяется классами, которые манипулируют этими структурами, чтобы определить, куда именно в структуру должен быть помещен объект. Если вы намерены использовать свой класс как ключ словаря, то должны переопределить GetHashCode().

GetType()

Этот метод возвращает экземпляр класса, унаследованный от System.Type. Этот объект может предоставить большой объем информации о классе, членом которого является ваш объект, включая базовый тип, методы, свойства и т.п. System.Type также представляет собой стартовую точку технологии рефлексии .NET.

ReferenceEquals()

Метод ReferenceEquals сравнивает две ссылки. Если ссылки на объекты идентичны, то возвращает true.

```
public static bool ReferenceEquals(object objA, object objB)
{
    return objA == objB;
}
```

Equals()

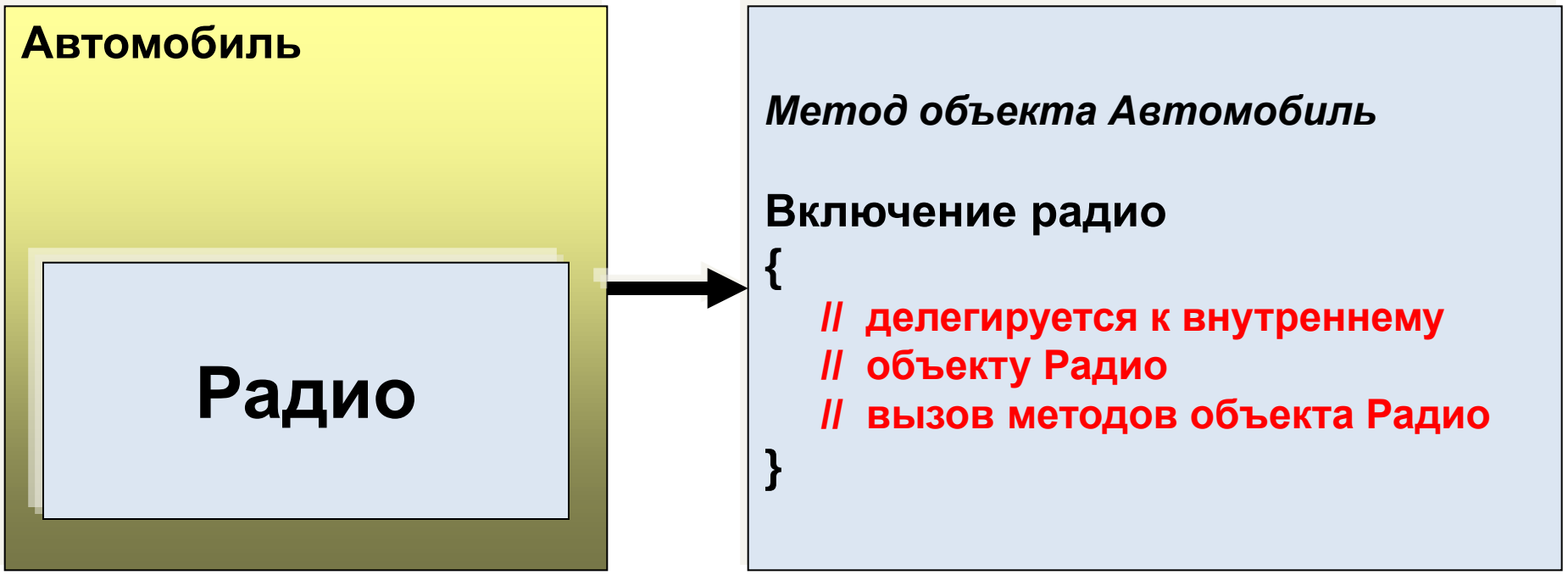
Метод проверяет экземпляры на тождество и если объекты не тождественны, то проверяет их на null и делегирует ответственность за сравнение переопределяемому экземплярному методу Equals.

```
public static bool Equals(object objA, object objB)
{
    return objA == objB ||
           (objA != null && objB != null && objA.Equals(objB));
}
```

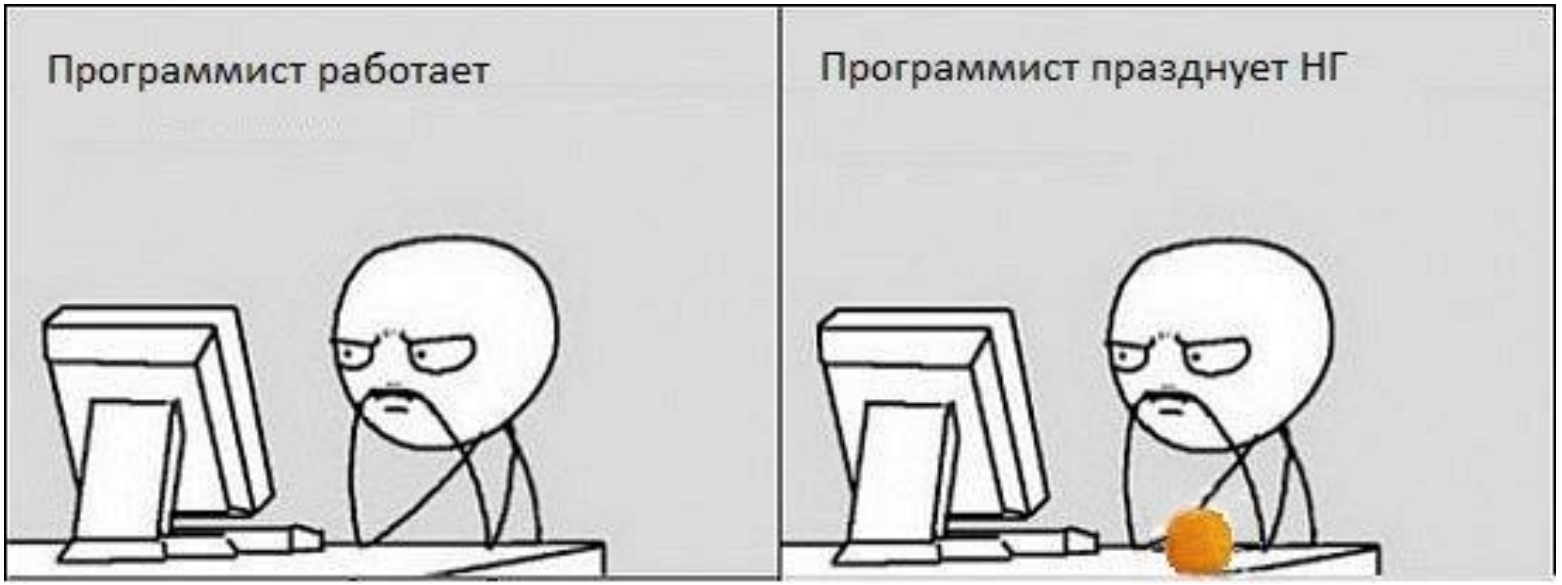
```
Vehicle vehicle = new Vehicle("Т/С", "общее", 50);
Console.WriteLine(vehicle.ToString());
Console.WriteLine(vehicle.GetType());
Console.WriteLine(vehicle.GetHashCode());

Vehicle vehicle2 = new Vehicle("Т/С", "общее", 50);
Console.WriteLine(vehicle.Equals(vehicle2));

Vehicle vehicle3 = vehicle;
Console.WriteLine(vehicle.Equals(vehicle3));

Console.WriteLine(Object.Equals(vehicle, vehicle2));
Console.WriteLine(Object.Equals(vehicle, vehicle3));

Console.WriteLine(Object.ReferenceEquals(vehicle, vehicle2));
Console.WriteLine(Object.ReferenceEquals(vehicle, vehicle3));
```

Наследование: включение-делегирование

Основная идея - создаются два независимых класса, работающих совместно, где внешний (контейнерный) класс создает внутренний класс и открывает внешнему миру его возможности. Делегирование заключается в простом добавлении во внешний контейнерный класс методов для обращения ко внутреннему классу.



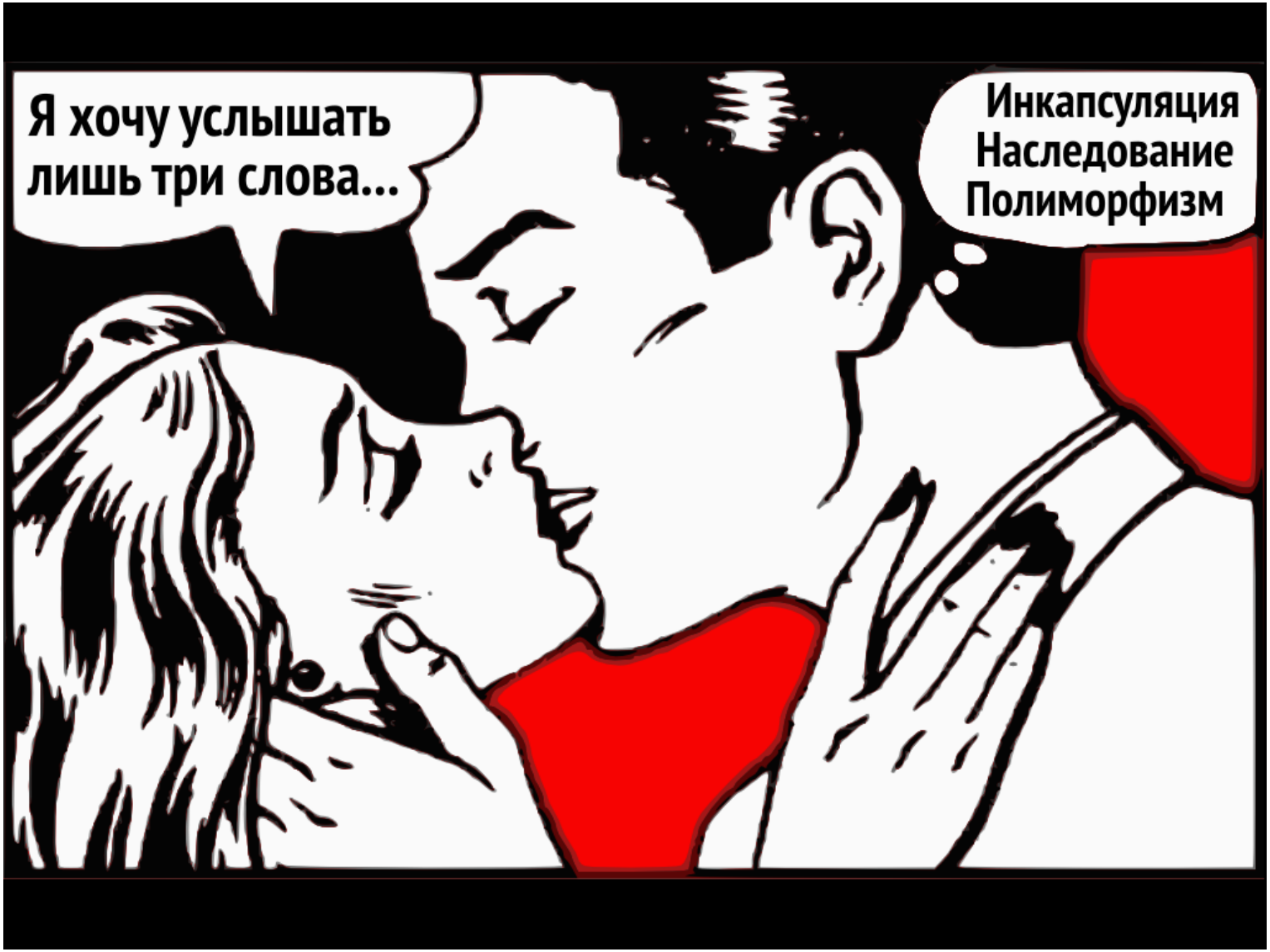
```
// класс транспортное средство
class Vehicle
{
    public string Type;
    public string Name;
    public double V;
    // делегирование класса Radio
    Radio radio = new Radio() { On=false};
    public string OnRadio(bool on)
    {  radio.On = on;
        return on ? "Радио вкл." : "Радио выкл.";
    }
}

// класс радио
class Radio
{
    bool on;  // включение радио
    public bool On
    {
        get { return on; }
        set { on = value; }
    }
}
```


Программист работает
Программист празднует НГ


Ура, я знаю где ошибка
Но если я исправлю это
полетит все...
Cyanide and Happiness © Explosm.net

Нормальная программа

Моя программа


Я хочу услышать лишь три слова...
Инкапсуляция
Наследование
Полиморфизм