Язык C#

Объектно-ориентированное программирование

Лекция #2

Определение класса

```
// Исходное определение класса
class Employee
{
     // Внутренние закрытые данные класса
     private string fullName;
     private int empID;
     private float currPay;

     // Конструкторы
     public Employee() {}  // Определение конструтора по умолчанию
     public Employee(string fullName, int empID, float currPay)
     {
          this.fullName = fullName;
          this.empID = empID;
          this.currPay = currPay;
     }

     // Метод для увеличения зарплаты сотрудника
     public void CiveBonus(float amount)
     { currPay += amount; }

     // Метод для вывода сведений о текущем состоянии объекта
     public virtual void DisplayStats()
     {
          Console.WriteLine("Name: {0}", fullName);
          Console.WriteLine("Pay: {0}", currPay);
          Console.WriteLine("ID: {0}", empID);
          Console.WriteLine("SSN: {0}", ssn);
     }
}
```

Определение класса

```
public static void Main()
{
 // Вызываем конструктор по умолчанию.
 // Заполняет все поля значениями по умолчанию
 Employee e = new Employee();

 // Вызываем пользовательский конструктор двумя способами
 Employee e1 = new Employee("Иван",80,30000);
 e1=GiveBonus(200);

 Employee e2;
 e2 = new Employee("Вася",81,50000);
 e2.GiveBonus(1000);
 e2.DisplayStats();

}
```

Использование ключевого слова this

```
class Employee
{
   public Employee(string fullName, int empID, float currPay)
   {
    this.fullName = fullname;
    this.empID = empID;
    this.currPay = currPay;
   }

   // Если пользователь вызовет этот конструктор, перенаправить
   // вызов варианту  с тремя параметрами
    public Employee(string fullName)
        :this(fullName, IDGenerator.GetNewEmpID(), 0.0F) {}
   ...
}
```

Определение открытого интерфейса по умолчанию

Открытый интерфейс по умолчанию – набор public-членов класса:

- **методы** – наборы действий
- **свойства** – функции для получения и изменения данных
- **поля** – не рекомендуется, но поддерживается C#

Указание области видимости на уровне типа

- class HelloClass { }
- // Класс доступен вне пределов сборки,
  // в которой он определен
  public class HelloClass { }
- // Класс доступен только внутри сборки,
  // в которой он определен
  internal HelloClass { }

Указание области видимости на уровне типа

```
namespace HelloClass
{
using System;

internal struct X   // Эта структура не сможет быть использована вне пределов данной сборки
{    private int myX;
     public int GetMyX() {return MyX; }
     public X(int x) { myX = x;}
}

internal enum Letters // Это перечисление не сможет быть использовано из-за пределов
            // данной сборки
{   a = 0, b = 1, c = 2
}

public class HelloClass  // Можно использовать откуда угодно
{
     public static int Main(string[] args)
     {    X theX = new X(26);
          Console.WriteLine(theX.GetMyX() + "\n" + Letters.b.ToString());
          return 0;
     }
}
}
```

Средства инкапсуляции в C#

Ко внутренним данным объекта нельзя обратиться через экземпляр этого объекта

- Создать традиционную пару методов (accessor, mutator)
- Определить свойство

Реализация инкапсуляции традиционными методами

```
// Определение традицонных методов доступа и изменения для закрытой
   переменной
public class Employee
{
  private string fullName;
  ...
  // Метод доступа
  public string GetFullName() {return  fullName; }

  // Метод изменения
  public void SetFullName(string n)
  {
   // Логика для удаления неположенных символов (!, @, #, $, % и прочих
   // Логика для проверки максимальной длины и прочего
   fullName = n;
  }
}
```

Реализация инкапсуляции традиционными методами

```
// Применение методов доступа и изменения
public static int Main(string[] args)
{
  Employee p = new Employee();
  p.SetFullName("Fred");
  Console.WriteLine("Employee is named: " +p.GetFullName());

  // Ошибка! К закрытым данным нельзя обращаться напрямую
  // через экземпляр объекта!
  // p.FullName;
  return 0;
}
```

Применение свойств класса

```
// Пользовательское свойство EmpID для доступа к переменной empID
public class Employee
{
  ...
  private int age;

  // Свойство для empID
  public int Age
  {
   get {return age;}
   set
   {
     // Здесь вы можете реализовать логику для проверки вводимых
     // значений и выполнения других действий
     age = value;
   }
  }
}
```

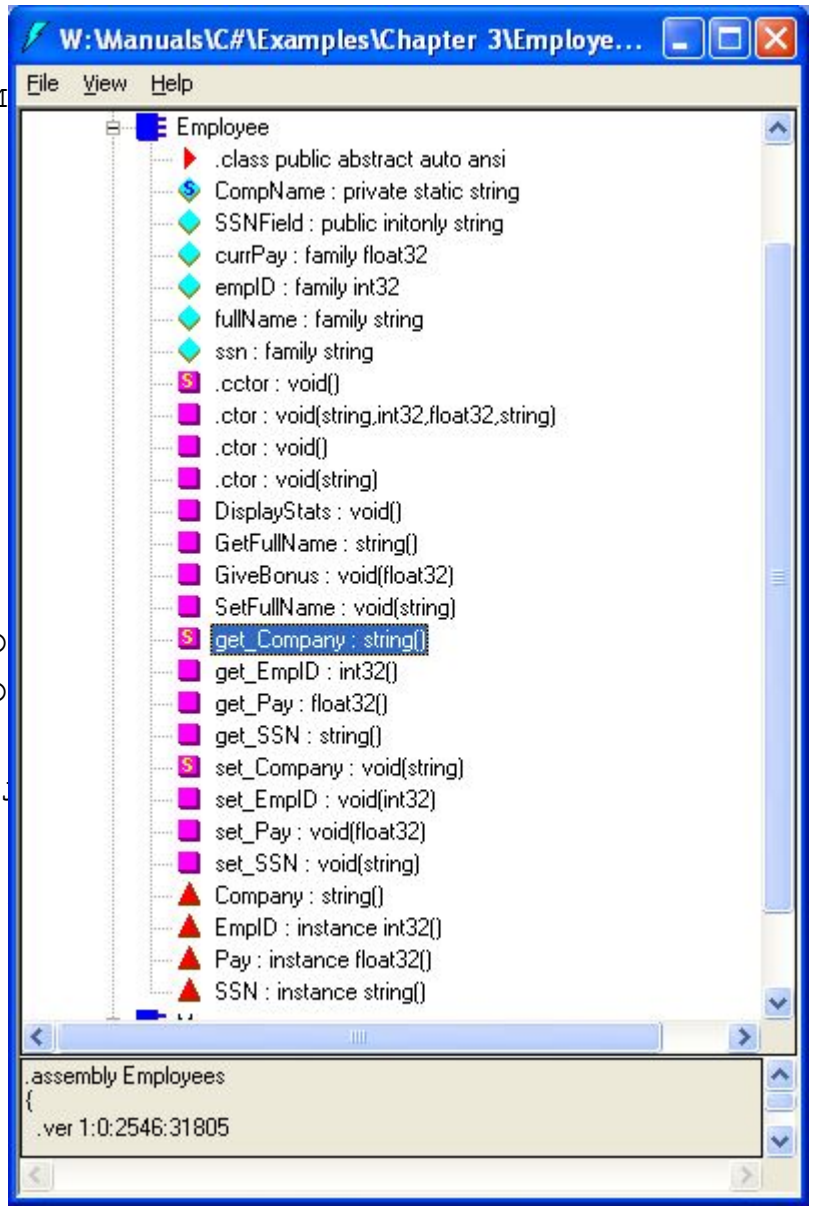
```
Employee joe = new Employee();
joe. Age++;
```

Внутреннее представление свойств

```
// Помните, что свойство C# автомати
// в пару методов get/set

public class Employee
{
...
        private string inn;

        // Определение свойства
    public string INN
    {
        get { return inn; }    // Ото
        set { inn = value; }   // Ото
    }
    // Ошибка! Эти методы уже опреде
    public string get_INN() { return
    public string set_INN(string val
}
```



Свойства только для чтения (или записи)

```
public class Employee
{
        // Будем считать, что исходное значение этого поля
    // присваивается конструктором класса
    private string inn;

        // Определение свойства только для чтения
    public string INN
    {   get { return inn; }
    }
}
```

Статические конструкторы

```
// Статические конструкторы используются
// для инициализации статических переменных
public class Employee
{
        // Статическая переменная
    private static string compname;

        // Статический конструктор
        // (не применяется модификатор видимости)
    static Employee()
    {
        compname = "Tsvetkov. Inc. Ltd.";
    }
}
```

Статические свойства

```
public class Employee
{
        // Статическая переменная
    private static string compname;

        // Статическое свойство
    public static string Company;
    {   get { return compname; }
        set { compname = value; }
    }
}
```

Создание полей «только для чтения»

```
public class Employee
{
 . . .
  // Поле только для чтения
  // (его значение устанавливается конструктором)

  public readonly string innField;
}
```

Статические поля «только для чтения»

```
// В классе Tire определен набор полей только для чтения
public class Tire
{
        public static readonly   Tire GoodStone = new Tire(90);
        public static readonly   Tire FireYear = new Tire(100);
        public static readonly   Tire ReadyLine = new Tire(43);
        public static readonly   Tire Blimpy = new Tire(83);

        private int manufactureID;

        public int MakeID
        {
                get { return manufactureID; }
        }

        public Tire (int ID)
        {
                manufactureID = ID;
        }
}
```

Статические поля «только для чтения»

```
// Так можно использовать динамически создаваемые поля только для чтения
public class Car
{
    // Какая у меня марка шин?
    public Tire tireType = Tire.Blimpy;      // Возвращает новый объект Tire
...
}

public class CarApp
{
    public static int Main(string[] args)
    {
      Car c = new Car();

      // Выводим на консоль идентификатор производителя шин
      // (в нашем случае — 83)
      Console.WriteLine("Manufacture ID of tires: {0}", c.tireType.MakeID);
      return 0;
    }
}
```

Поддержка наследования

Наследование в C#

- Классическое наследование (отношение «быть» – is-a)
- Включение-делегирование (отношение «иметь» – has-a)

Добавляем в пространство имен Employees два новых производных класса

```
namespace Employees {
public class Manager : Employee
{
    // Менеджерам необходимо знать количество имеющихся у них заявок на акции
    private ulong numberOfOptions;
    public ulong NumbOpts
    {
     get { return numberOfptions; }
     set { numberOfOptions = value; }
    }
}

public class SalesPerson : Employee
{
    // Продавцам нужно знать объем своих продаж
    private int numberOfSales;
    public int NumbSales
    {
     get { return numberOfSales; }
     set { numberOfSales = value; }
    }
}
```

Создаем объект производного класса и проверяем его возможности

```
public static int Main(string[] args)
{
    // Создаем объект «продавец»
    SalesPerson stan = new SalesPerson();

    // Эти члены унаследованы от базового класса Employee
    stan.EmpID = 100;
    stan.SetFullName("Stan the Man");

    // А это — уникальный член, определенный только в классе SalesPerson
    stan.NumbSales = 42;

    return 0;
}
```

Работа с конструктором базового класса

```
// При создании объекта производного класса конструктор производного класса
// автоматически вызывает конструктор базового класса по умолчанию
public Manager(string fullName, int empID, float currPay, string ssn,
               ulong numbOfOpts)
{
    // Присваиваем значения уникальным данным нашего класса
    numberOfOptions = numbOfOpts;

    // Присваиваем значения данным, унаследованным от базового класса
    EmpID = empID;
    SetFullName(fullName);
    SSN = ssn;
    Pay = currPay;
}
```

```
public Manager(string fullName, int empID, float currPay, string ssn,
        ulong numbOfOpts)  :base(fullName, empID, currPay, snn)
{
        numberOfOptions = numbOfOpts;
}
```

Множественное наследование

- В C# множественное наследование классов запрещено
- Множественное наследование интерфейсов разрешено

Защищаемые поля

- Использование ключевого слова protected

```
public class Employee
{
  protected string fullName;
  protected int empID;
}
```

Запрет наследования
«Запечатанные» классы

```
public sealed class PartTimePerson: SalesPerson
{
  public PartTimePerson(string fullName, int empID)
  {
   . . .
  }
  . . .
}
```

```
Public class ReallyPTSalesPerson : PartTimePerson
{   // ТАК НЕЛЬЗЯ – СИНТАКСИЧЕСКАЯ ОШИБКА
  …
}
```

Модель включения-делегирования
has-a

```
public class Radio
{

public Radio(){}

 public void TurnOn(bool on)
 {
  if (on)
     Console.WriteLine("Jamming…");
  else
     Console.WriteLine("Quiet time…");
 }

}
```

```
public class Car
{// Этот класс будет выступать в роли внешнего класса, класса-контейнера для Radio
     private int currSpeed;
     private int maxSpeed;
     private string petName;
     bool dead;                // Жива ли машина или уже нет

     public Car()   {   maxSpeed = 100; dead = false;}

     public Car(string name, int max, int curr)
     {  currSpeed = curr; maxSpeed = max; petName = name; dead = false; }

     public void SpeedUp (int delta)
     {  // Если машина уже «мертва» (при превышении максимальной скорости),
        // то следует сообщить об этом
          if(dead)
             Console.WriteLine(petName + " is out of order...");
          else      // Пока еще все нормально, увеличиваем скорость
          {  currSpeed += delta;
             if (currSpeed >= max.Speed)
             {
                Console.WriteLine(petName + " has overheated...");
                dead = true;
              }
             else
                Console.WriteLine("\tCurrSpeed = " + currSpeed);
          }
     }
}
```

Помещение радиоприемника внутрь автомобиля

```
// Автомобиль «имеет» (has-a) радио
public class Car
{
  . . .

  // Внутреннее радио
  private Radio theMusicBox;

}
```

За создание объектов внутренних классов ответственны контейнерные классы

```
public class Car
{
...
    // Встроенное радио
    private Radio theMusicBox;

    public Car()
    {maxSpeed = 100;
     dead = false;

     // Объект внешнего класса создаст необходимые объекты внутреннего класса
     // при собственном создании
     theMusicBox = new Radio();  // Если мы этого не сделаем, theMusicBox
                    // начнет свою жизнь с нулевой ссылки
     }

     public Car(string name, int max, int curr)
     {
          currSpeed = curr;
          maxSpeed = max;
          petName = name;
          dead = fales;
          theMusicBox = newRadio();
     }
...
}
```

Произвести инициализацию средствами C# можно и так

```
public class Car
{
    ...
    // Встроенное радио
    private Radio theMusicBox = new Radio;

    ...
}
```

Делегирование

```
public class Car
{
    ...
    // Встроенное радио
    private Radio theMusicBox = new Radio;

    ...


   public void CrankTunes(bool state)
   {
     // Передаем (делегируем) запрос внутреннему объекту
     theMusicBox.TurnOn(state)
   }

}
```

Использование делегирования

```
// Выводим автомобиль на пробную поездку
public class CarApp
{
    public static int Main(string[] args)
    {
     // Создаем автомобиль (который, в свою очередь, создаст радио)
     Car c1;
     c1 = new Car("Volga", 100, 10);

     // Включаем радио (запрос будет передан внутреннему объекту)
     c1.CrankTunes(true);

     // Ускоряемся
     for(int i = 0; i < 10; i++)
           c1.SpeedUp(20);

     // Выключаем радио (запрос будет вновь передан внутреннему объекту)
     c1.CrankTunes(false);
     return 0;
    }
}
```

Определение вложенных типов

```
// В C# можно вкладывать в друг друга классы, интерфейсы и структуры

public class MyClass
{
  // Члены внешнего класса
  …
  public class MyNestedClass
  {
    // Члены внутреннего класса
    …
  }
}
```

Внутренние классы – как правило, вспомогательные. Их область видимости ограничена внешним классом.

Класс Radio теперь вложен в Car

```
// Класс Radio вложен в класс Car. Все остальное — как в предыдущем
   приложении
public class Car : Object
{
...
  // К вложенному закрытому класу Radio нельзя обратиться из внешнего мира
   private class Radio
    {
   public Radio(){}
   public void TurnOn(bool on)
   {
    if (on)
         Console.WriteLine("Jamming...");
    else
         Console.WriteLine("Quiet time...");
   }
    }

   // Внешний класс может создавать экземпляры вложенных типов
   private Radio theMusicBox;
...
}
```

Поддержка полиморфизма в C#

```
// Пусть в классе Employee определен метод для поощрения сотрудников

public class Employee
{
 …
 public void GiveBonus (float amount)
 {
    currPay += amount
 }
    …
}
```

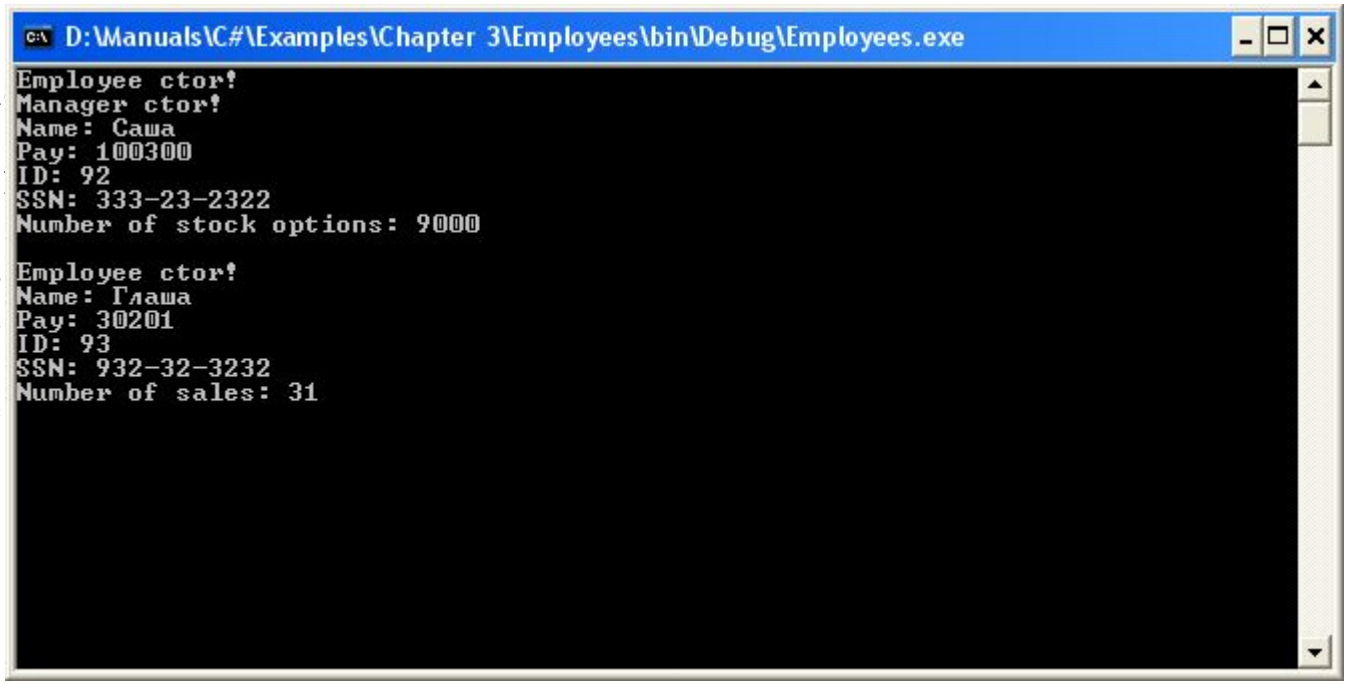
Использование метода GiveBonus

```
public static int Main(string[] args)
{
  Manager sasha = new Manager("Саша", 92, 100000, "333-23-2322", 9000);
  sasha.GiveBonus(300);
  sasha.DisplayStats();


  Console.WriteLin

  SalesPerson glas
31);
  glasha.GiveBonus
  glasha.DisplaySt

  Console.ReadLine
}
```



Проблема в том, что метод GiveBonus пока работает абсолютно одинаково в отношении всех производных классов.

Поддержка полиморфизма в C#

```
public class Employee
{
 …

// Для метода GiveBonus предусмотрена реализация по умолчанию.
// Однако он может быть замещен в производных классах

 public virtual void GiveBonus (float amount)
 {
    currPay += amount
 }
    …
}
```

Переопределение виртуальных методов

```
public class SalesPerson : Employee
{
  // На размер поощрения продавцу будет влиять объем его продаж
  public override void GiveBonus(float amount)
  {
   int salesBonus = 0;

   if(numberOfSales >= 0 && numberOfSales <=100)  salesBonus = 10;
   else if(numberOfSales >= 101 && numberOfSales <= 200)    salesBonus = 15;
   else salesBonus = 20; // Для объема продаж больше 200

   base.GiveBonus (amount * salesBonus);
  }
...
}

public class Manager : Employee
{
  private Random r = new Random();

  // Помимо денег менеджеры также получают некоторое количество дивидентов по акциям
  public override void GiveBonus(float amount)
  {
    base.GiveBonus(amount); // Деньги: увеличиваем зарплату
    numberOfOptions += (ulong) r.Next(500); // Дивиденты: увеличиваем их число
  }
...
}
```

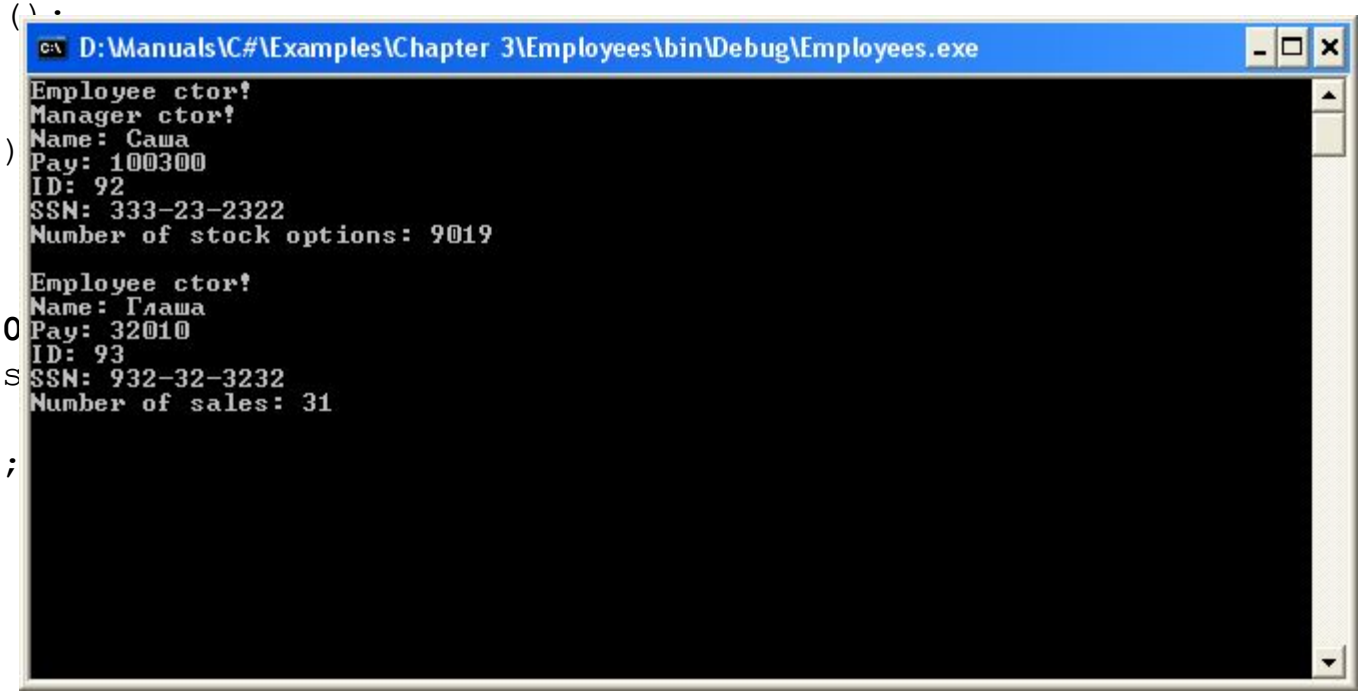
Улучшенная система поощрений!

```
public static int Main(string[] args)
{
  Manager sasha = new Manager("Саша", 92, 100000, "333-23-2322", 9000);
  sasha.GiveBonus(300);
  sasha.DisplayStats();


  Console.WriteLine()

  SalesPerson glasha
31);
  glasha.GiveBonus(20
  glasha.DisplayStats

  Console.ReadLine();
}
```



Код использования не изменился!

Абстрактные классы

```
Employee X = new Employee(); // А это кто такой?


// Создание объектов абстрактного класса запрещено!
abstract public class Employee()
{
}


Employee X = new Employee(); // Теперь это ошибка!
```

Абстрактные методы

```
namespace Shapes
{
  public abstract class Shape
   {  // Пусть каждый объект-геометрическая фигура получит у нас дружеское прозвище:
      protected string petName;

      // Конструкторы
      public Shape() {petName = "NoName";}
      public Shape(string s) {petName = s;}

      // Метод Draw() объявлен как виртуальный и может быть замещен
      public virtual void Draw()  {    Console.WriteLine("Shape.Draw()");  }
      public string PetName       {    get {return petName;}     set {petName = value;}
      }
   }

// В классе Circle метод Draw() НЕ ЗАМЕЩЕН
 public class Circle : Shape
{    public Circle() {}
     public Circle(string name): base(name) {}
}

// В классе Hexagon метод Draw() ЗАМЕЩЕН
public class Hexagon : Shape
{   public Hexagon() {}
    public Hexagon(string name) : base(name) {}
    public overrride void Draw()    {       Console.WriteLine("Drawing {0} the Hexagon", PetName);     }
}
}
```

Абстрактные методы

```
// В объекте Circle реализация базового класса для Draw() не замещена
public static int Main(string [ ] args)
{
        // Создаем и рисуем шестиугольник
        Hexagon hex = new Hexagon("Beth");
        hex.Draw();

        Circle cir = new Circle("Cindy");
        // М-мм! Придется использовать реализацию Draw() базового класса
        cir.Draw();
...
}
```

Абстрактные методы

```
// Каждая геометрическая фигура теперь ОБЯЗАНА самостоятельно определять  метод Draw()

public abstract class Shape
{
...
      // Метод Draw() теперь определен как абстрактный (обратите внимание  на точку с запятой)
      public abstract void Draw();

      public string PetName
      {
     get {return petName;}
     set {petName = value;}
     }

}
```

Абстрактные методы

```
// Если мы не заместим в классе Circle абстрактный метод Draw(), класс Circle будет
// также считаться абстрактным и мы не сможем создавать объекты этого класса!

public class Circle : Shape
{
    public Circle() {}
    public Circle(string name): base(name) {}

    // Теперь метод Draw() придется замещать в любом производном непосредственно
    // от Shape классе
    public override void Draw()
    {
      Console.WriteLine("Drawing {0} the Cricle", PetName);
    }
}
```

Полиморфизм в действии

```
namespace Shapes
{
using System;

public class ShapesApp
{
    public static int Main(string[] args)
    {
   // Массив фигур
   Shape[ ] s = {new Hexagon(), new Circle(), new Hexagon("Mick"),
                 new Circle("Beth"), new Hexagon("Linda")};

   // Отрисовываем в цикле каждый объект!
   for(int i = 0; i < s.Length; i++)
   {
         s[i].Draw();
   }

   return 0;

    }
```

Сокрытие методов

```
// Класс Oval наследует Circle, но скрывает унаследованный
// метод Draw
public class Oval : Circle
{
  public Oval() { base.PetName="Иван" }

  // Скрываем любые реализации Draw() базовых классов
  public new void Draw()
  {
    …
  }

}
```

Приведение типов

```
// Класс Manager – производный от System.Object
object o = new Manager(…)

// Класс Manager – производный от Emloyee
Employee e = new Manager(…)

// Класс Manager – сам по себе класс
Manager m = new Manager(…)
```

Если один класс является производным от другого, всегда безопасно ссылаться на объект производного класса через объект базового класса.

Использование приведения типов

```
public class TheMachine
{
  public static void FireThisPerson(Employee e)
  {
    // Удаляем сотрудника из базы данных
    // Отбираем у него ключ и точилку для карандашей
  }
}

TheMachine.FireThisPerson(e);
TheMachine.FireThisPerson(m);

TheMachine.FireThisPerson(o); // Ошибка компилятора
TheMachine.FireThisPerson((Employee) o); // Так можно
```

Обработка исключений

Генерация исключения

```
// В настоящее время SpeedUp()выводит сообщения об ошибках прямо на системную консоль
public void SpeedUp(int delta)
{
    if(dead)  // Если машины больше нет, сообщить об этом
     Console.WriteLine(petName + " is out of order...");
    else  // Еще жива, можно увеличивать скорость
    {currSpeed += delta;
     if (currSpeed >= maxSpeed)
     {    Console.WriteLine(petName + " has overheated...");
          dead = true;
     }
     else
          Console.WriteLine("\tCurrSpeed = " + currSpeed);
     }
}
```

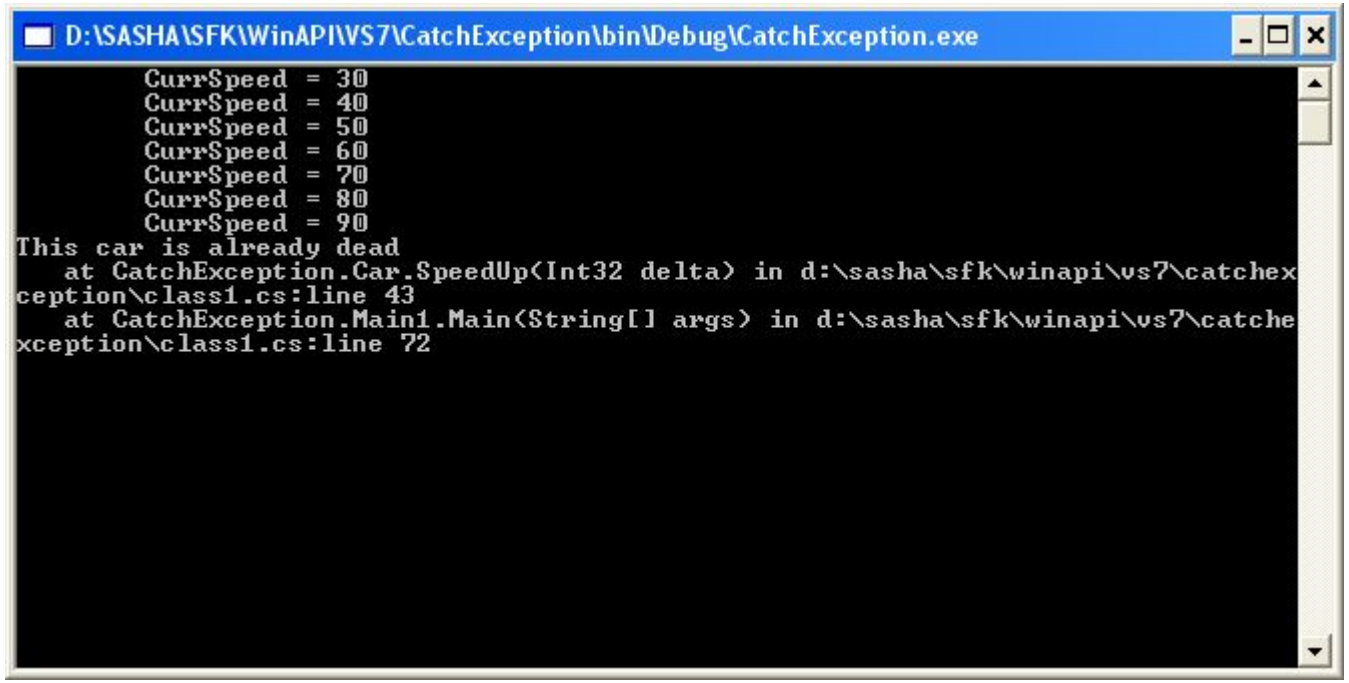
```
// При попытке ускорить вышедший из строя автомобиль
// будет сгенерировано исключение
public void SpeedUp(int delta)
{
     if (dead)
          throw new Exception("This car is already dead!")
     else
     { …
     }
}
```

Перехват исключений

```
public static int Main(string[ ] args) // Безопасно разгоняем автомобиль
{
    Car buddha = new Car("Buddha", 100, 20); // Создаем автомобиль

    try    // Пытаемся прибавить газ
    {
     for(int i = 0; i < 10; i++)
          buddha.SpeedUp(10);
    }
    catch(Exception e)    // Выводим сообщение и трассируем стек
    {
     Console.WriteLine(e.Message);
     Console.WriteLine(e.StackTrace);
    }

     return 0;
}
```



Создание пользовательских исключений

```
// Это пользовательское исключение более подробно описывает
// ситуацию выхода машины  из строя
public class CarIsDeadException : System.Exception
{
    // С помощью этого исключения мы сможем получить имя несчастливой машины
    private string carName;
    public CarIsDeadException(){}
    public CarIsDeadException(string carName)
    {
     this.carName = carName;
    }
    // Замещаем свойство Exception.Message
    public override string Message
    {
     get
     {    string msg = base.Message;
          if (carName != null)
               msg += carName + " has bought the farm...";
          return msg;
     }
    }
}
```

Генерация пользовательского исключения

```
// Генерируем пользовательское исключение
public void SpeedUp(int delta)
{
    // Если машина вышла из строя, сообщаем об этом
    if (dead)
    {
     // Генерируем исключение
     throw new CarIsDeadException(this.petName);
    }
    else // Машина еще жива, можно разгоняться
    {
     currSpeed += delta;
     if (currSpeed >= maxSpeed)
     {
          dead = true;
     }
     else
          Console.WriteLine("\tCurrSpeed = {0}", currSpeed);
    }
}
```

Перехват пользовательского исключения

```
try
{
  for(int i = 0; i < 10; i++)
       buddha.SpeedUp(10);
}
catch (CarIsDeadException e)
{
   Console.WriteLine(e.Message);
   Console.WriteLine(e.StackTrace);
}
```

```
// Захват всех исключений без разбора
catch
{
   Console.WriteLine("Something bad happened…");
}
```

Создание пользовательских исключений (2-й вариант)

```
public class CarIsDeadException : System.Exception
{
 // Конструкторы для создания польз. сообщения об ошибке
 public CarIsDeadException(){}

 public CarIsDeadException(string message)
    : base(message) {}

 public CarIsDeadException(string message, Exception innerEx)
    : base(message, innerEx) {}

}
```

Генерация пользовательских исключений (2-й вариант)

```
public void SpeedUp(int delta)
{
  if (dead)
  {
  // Передаем имя машины и сообщение как аргументы конструктора
  throw new CarIsDeadException(this.petName + " is destroyed!");
  }
  else  // Машина еще жива, можно разгоняться
  {
  }
}
```

Обработка нескольких исключений

```
// Проверка параметров на соответствие условиям
public void SpeedUp(int delta)
{
    // Ошибка в принимаемом параметре? Генерируем системное исключение
    if (delta < 0)
    throw new ArgumentOutOfRangeException("Must be greater than zero");

    // Если машина вышла из строя — сообщить об этом
    if (dead)
    {
    // Генерируем исключение CarIsDeadException
    throw new CarIsDeadException (this.petName + " has bought the farm!");
    }
     ...
}
```

Обработка нескольких исключений

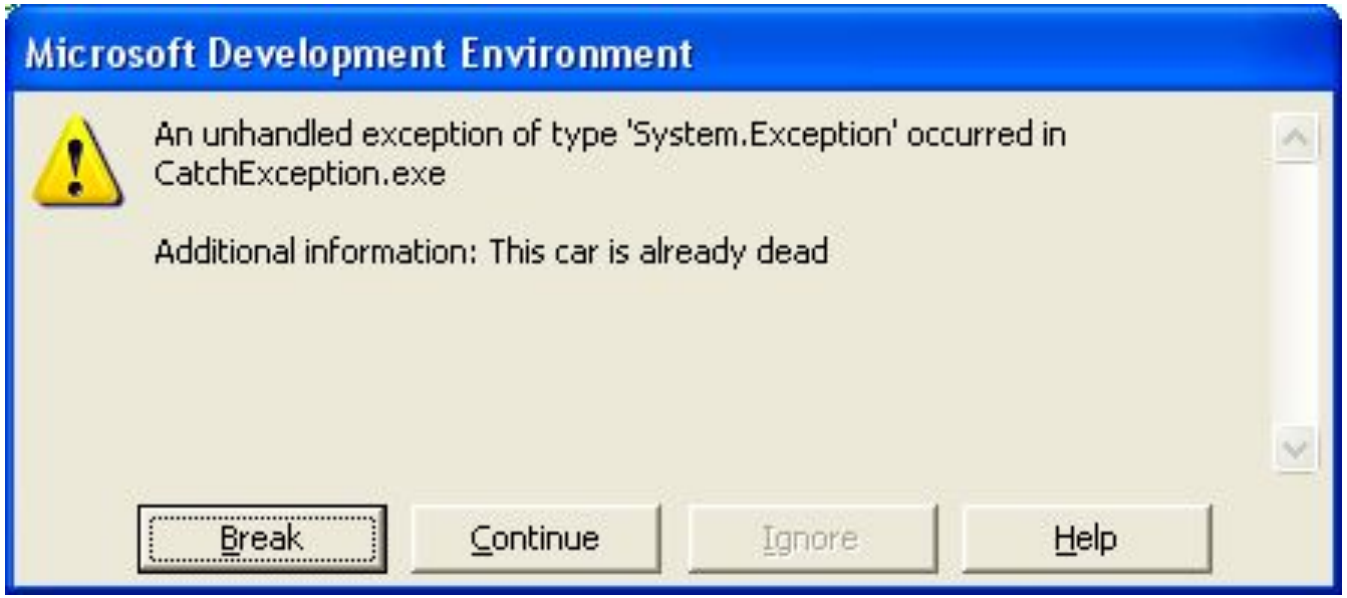
```
// Теперь мы готовы перехватить оба исключения
try
{
   for(int i = 0; i < 10; i++)
    buddha.SpeedUp(10);
}
catch (CarIsDeadException e)
{
   Console.WriteLine(e.Message);
   Console.WriteLine(e.StackTrace);
}
catch (ArgumentOfRangeException e)
{
   Console.WriteLine(e.Message);
   Console.WriteLine(e.StackTrace);
}
```

Блок finally

```
// Используем блок finally для закрытия всех ресурсов
public static int Main(string[] args)
{
    Car buddha = new Car("Buddha", 100, 20);
    buddha.CrankTunes(true);

    // Давим на газ
    try
    {// Разгоняем машину...
    }
    catch (CarIsDeadException e)
    {Console.WriteLine(e.Message);
     Console.WriteLine(e.StackTrace);
    }
    catch (ArgumentOutOfRangeException e)
    {Console.WriteLine(e.Message);
     Console.WriteLine(e.StackTrace);
    }
    finally
    {// Этот блок будет выполнен всегда — вне зависимости от того,
     // произошла ошибка или нет
     buddha.CrankTunes(false);
    }
    return 0;
}
```

Необработанное исключение



Не допускайте бесконечной генерации ошибок

```
try
{  // Разгоняем машину...
}
catch (CarIsDeadException e)
{
  // Код для реакции на захваченное исключение
  // В этом коде мы генерируем то же самое исключение.
  // При необходимости следует генерировать другое исключение

  throw e;

}
```

Жизненный цикл объектов

```
public static int Main(string[] args)
{
  // Помещаем объект класс Car в «кучу»
  Car c = new Car("Lada", 200, 100);
  …
  return 0;
  // Если c – единственная ссылка на объект, то
  // начиная с этого места он может быть удален
}
```

Контроль за свободной памятью

```
// Создаем объекты Car таким образом, чтобы отреагировать на
// возможную нехватку места  в управляемой куче
public static int Main(string[] args)
{
...
   Car yetAnotherCar;
   try
   {
    yetAnotherCar = new Car();
   }
   catch (OutOfMemoryException e)
   {
    Console.WriteLine(e.Message);
    Console.WriteLine("Managed heap is FULL! Running GC...");
   }
...
    return 0;
}
```

Финализация объекта

- System.Object.Finalize()
- Нельзя замещать
- Нельзя вызывать

```
public class Car : Object
{
 // Деструктор C# ?
 ~Car()
  {
    // Закрываем открытые ресурсы
  }
}
```

Методы удаления

```
public Car: IDisposable
{
  …
  // Этот метод пользователь может вызывать вручную
  public void Dispose()
  {
    // удаление наиболее значимых ресурсов
  }
}
```