# Web-разработка на C# и платформе Microsoft .NET

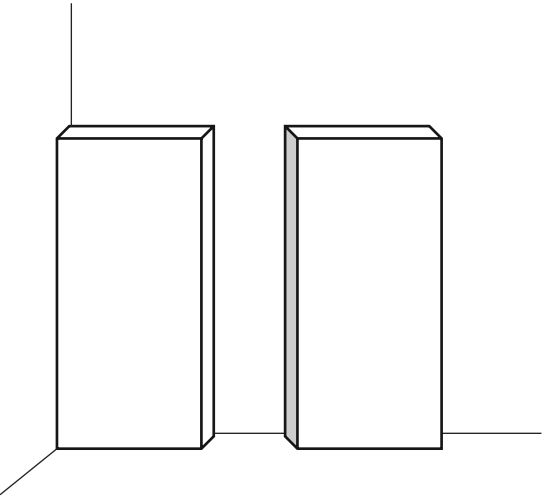
ООП – часть 1

# План занятия

- Принципы ООП
- Абстракция
- Инкапсуляция
- Наследование
- Полиморфизм

# Шесть принципов Алана Кэя

## 1. ВСЁ ЯВЛЯЕТСЯ ОБЪЕКТОМ.



# Шесть принципов Алана Кэя

**2. КАЖДЫЙ ОБЪЕКТ ЯВЛЯЕТСЯ ПРЕДСТАВИТЕЛЕМ (ЭКЗЕМПЛЯРОМ) КЛАССА, КОТОРЫЙ ВЫРАЖАЕТ ОБЩИЕ СВОЙСТВА ОБЪЕКТОВ.**

# Шесть принципов Алана Кэя

**3. В КЛАССЕ ЗАДАЁТСЯ ПОВЕДЕНИЕ (ФУНКЦИОНАЛЬНОСТЬ) ОБЪЕКТА. ТЕМ САМЫМ ВСЕ ОБЪЕКТЫ, ЯВЛЯЮЩИЕСЯ ЭКЗЕМПЛЯРАМИ ОДНОГО КЛАССА, МОГУТ ВЫПОЛНЯТЬ ОДНИ И ТЕ ЖЕ ДЕЙСТВИЯ.**

# Шесть принципов Алана Кэя

**4. КЛАССЫ ОРГАНИЗОВАНЫ В ЕДИНУЮ ДРЕВОВИДНУЮ СТРУКТУРУ С ОБЩИМ КОРНЕМ, НАЗЫВАЕМУЮ ИЕРАРХИЕЙ НАСЛЕДОВАНИЯ. ПАМЯТЬ И ПОВЕДЕНИЕ, СВЯЗАННОЕ С ЭКЗЕМПЛЯРАМИ ОПРЕДЕЛЁННОГО КЛАССА, АВТОМАТИЧЕСКИ ДОСТУПНЫ ЛЮБОМУ КЛАССУ, РАСПОЛОЖЕННОМУ НИЖЕ В ИЕРАРХИЧЕСКОМ ДЕРЕВЕ.**

# Шесть принципов Алана Кэя

## 5. КАЖДЫЙ ОБЪЕКТ ИМЕЕТ НЕЗАВИСИМУЮ ПАМЯТЬ, КОТОРАЯ СОСТОИТ ИЗ ДРУГИХ ОБЪЕКТОВ.

# Шесть принципов Алана Кэя

**6. ВЫЧИСЛЕНИЯ ОСУЩЕСТВЛЯЮТСЯ ПУТЁМ ВЗАИМОДЕЙСТВИЯ (ОБМЕНА ДАННЫМИ) МЕЖДУ ОБЪЕКТАМИ, ПРИ КОТОРОМ ОДИН ОБЪЕКТ ТРЕБУЕТ, ЧТОБЫ ДРУГОЙ ОБЪЕКТ ВЫПОЛНИЛ НЕКОТОРОЕ ДЕЙСТВИЕ.**

**ОБЪЕКТЫ ВЗАИМОДЕЙСТВУЮТ, ПОСЫЛАЯ И ПОЛУЧАЯ СООБЩЕНИЯ.**

**СООБЩЕНИЕ – ЭТО ЗАПРОС НА ВЫПОЛНЕНИЕ ДЕЙСТВИЯ, ДОПОЛНЕННЫЙ НАБОРОМ АРГУМЕНТОВ, КОТОРЫЕ МОГУТ ПОНАДОБИТЬСЯ ПРИ ВЫПОЛНЕНИИ ДЕЙСТВИЯ.**

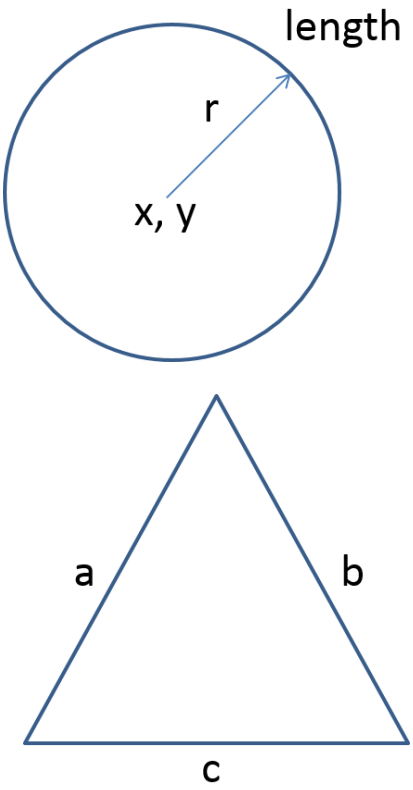
# Абстракция

- Выделяйте только те факторы, которые нужны для решения задачи
- Отсекайте всё лишнее




# Инкапсуляция

- Пользователь не должен менять внутреннее состояние объекта
- Поля и методы делятся на внутренние и интерфейсные



# Спецификаторы

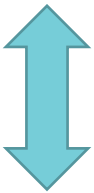
| | Классы | Методы | Поля |
|---|---|---|---|
| private | ✓ | ✓ | ✓ |
| protected | ⊘ | ✓ | ✓ |
| internal | ✓ | ✓ | ✓ |
| protected internal | ⊘ | ✓ | ✓ |
| public | ✓ | ✓ | ✓ |
| | | | |
| static | ✓ | ✓ | ✓ |
| sealed | ✓ | ✓ | ⊘ |
| abstract | ✓ | ✓ | ⊘ |
| virtual | ⊘ | ✓ | ⊘ |
| override | ⊘ | ✓ | ⊘ |
| new | ⊘ | ✓ | ⊘ |

# Поля и свойства

- Поле – переменная, хранящая значение.
- Свойство – пара методов, предназначенных для правильной инкапсуляции поля.
  - Accessor, или getter (get) не должен выполнять длительных вычислений
  - Mutator, или setter (set) должен проверять значения, передавать уведомления. Значение передаётся через ключевое слово value.

# Новый синтаксис свойств в C# (>3.0)

```
private double r;
public double R
{
        get { return r; }
        set { r = value; }
}
```



```
public double R { get; set; }
```

# Свойства - это методы

```
CallDateTime start = new CallDateTime(
        DateTime.Now.Year, DateTime.Now.Month,
        DateTime.Now.Day, DateTime.Now.Hour,
        DateTime.Now.Minute, DateTime.Now.Second);

// ..... идёт звонок .....

CallDateTime finish = new CallDateTime(
        DateTime.Now.Year, DateTime.Now.Month,
        DateTime.Now.Day, DateTime.Now.Hour,
        DateTime.Now.Minute, DateTime.Now.Second);
```

# Более верное решение

```
DateTime date = DateTime.Now;
CallDateTime start = new CallDateTime(
        date.Year,
        date.Month,
        date.Day,
        date.Hour,
        date.Minute,
        date.Second);
```

# Методы

- Конструкторы
- Деструкторы
- Статические методы
- Операторы
- Переопределение методов

# Конструкторы

Автоматический конструктор

```
Circle c = new Circle();
```

Явно заданный конструктор по умолчанию

```
public Circle()
{
        x = y = 0;
        r = 1;
}
```

Конструктор с параметрами

```
public Circle(double r)
{
        x = y = 0; this.r = r;
}
```

# Спецификатор “static”

Статические поля

```
static int x;
MyClass.x = 5;
```

Статические методы

```
static int Method() { ... }
MyClass.Method();
```

Статические конструкторы

```
static MyClass()
{ ... }
```

Статические классы

```
static class MyClass { ... }
```

# Конструкторы

Автоматический конструктор

```
Circle c = new Circle();
```

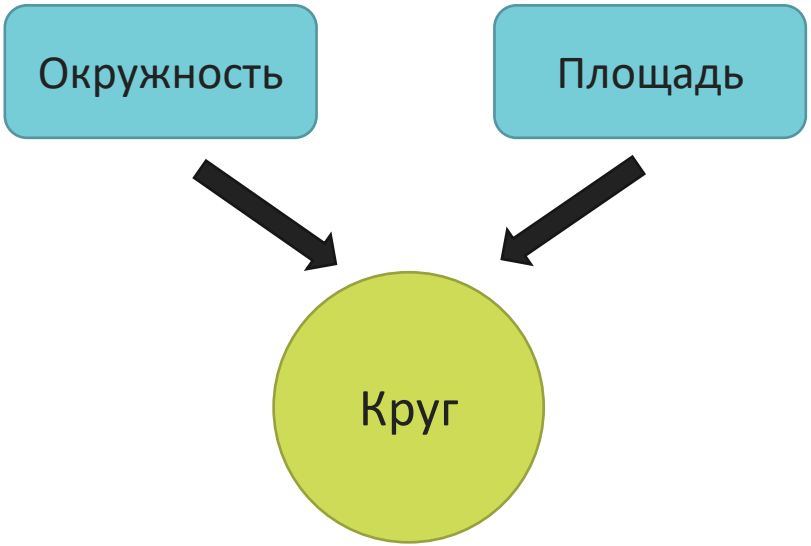
Явно заданный конструктор по умолчанию

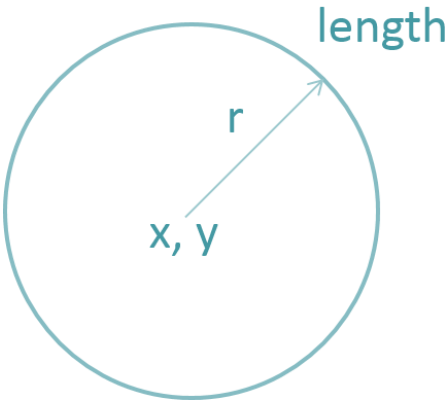
```
public Circle()
{
        x = y = 0;
        r = 1;
}
```

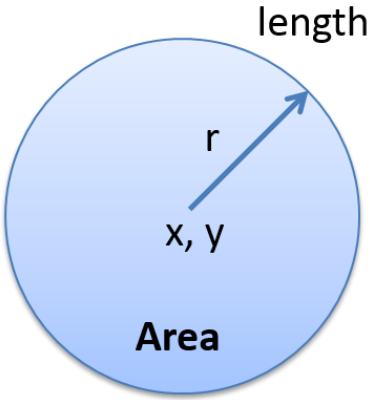
Конструктор с параметрами

```
public Circle(double r)
{
        x = y = 0;
        this.r = r;
}
```

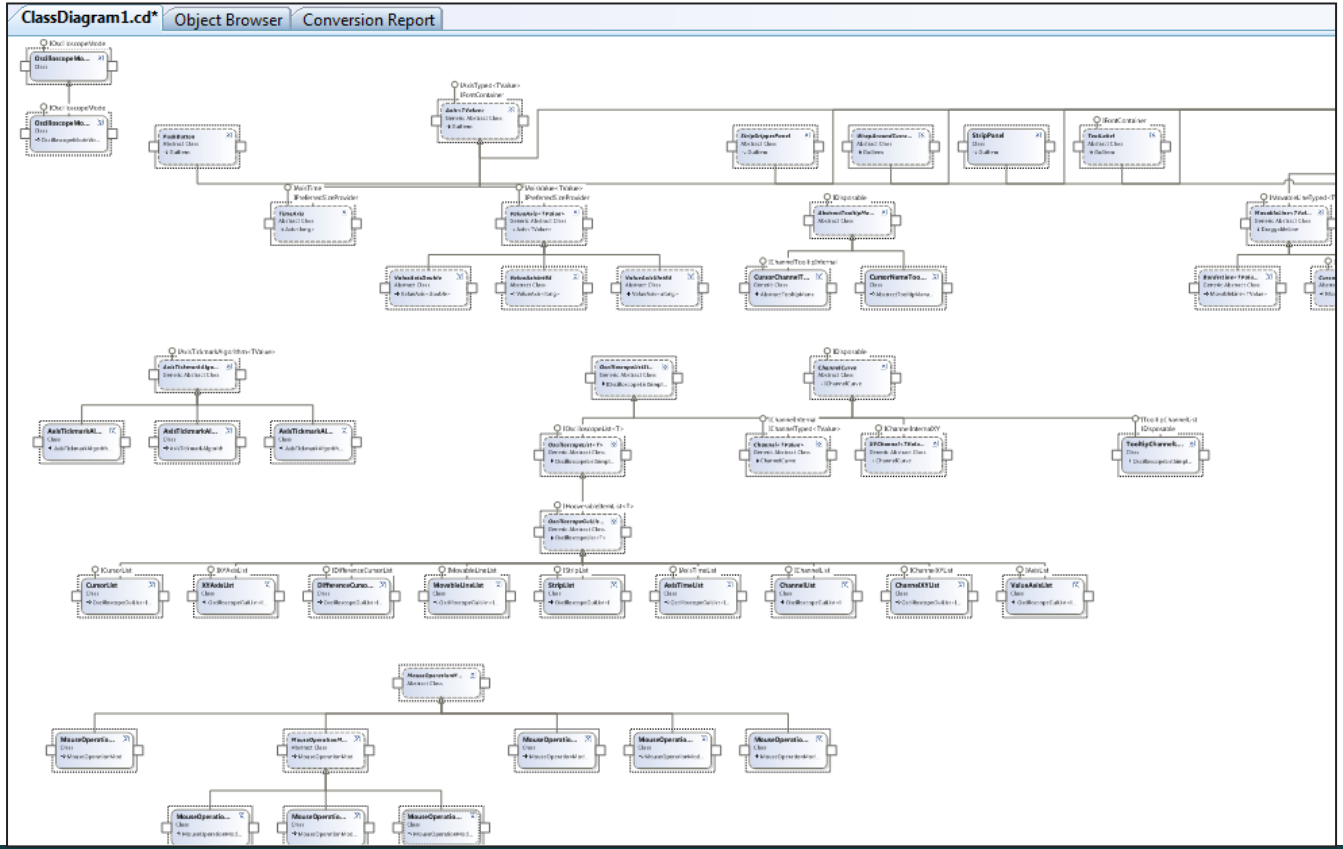
# Наследование



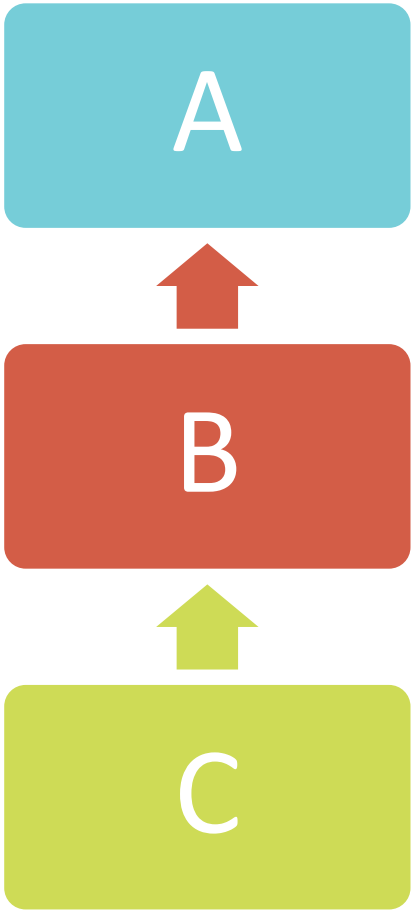



# Диаграмма классов реального проекта (фрагмент)



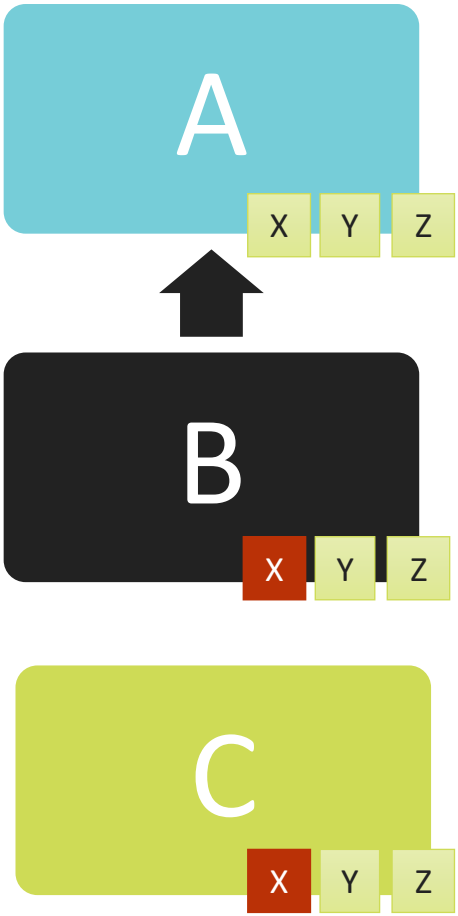
# Ключевые термины

- Наследование – inheritance
- Базовый класс, родительский класс, суперкласс – base class, parent class
- Наследник, потомок, производный класс – derived class, child class
- Прямые и косвенные предки
- Одиночное и множественное наследование



# Спецификатор “protected”

```
class A
{
        private int X;
        protected int Y;
        public int Z;
        public void DoSomething() { ... }
}
class B : A
{
        public void DoSomething() { ... }
}
class C : B
{
        public void DoSomething() { ... }
}
```



# Конструкторы при наследовании

```
class Circle
{
    private int x, y, r;
    public Circle()
    {
        x = y = r = 0;
    }
    public Circle(int r)
        : this()
    {
        this.r = r;
    }
}
```

```
class Ring : Circle
{
    private int innerR;
    public Ring()
    {
    }
    public Ring(int r, int ir)
        : base(r)
    {
        innerR = ir;
    }
}
```

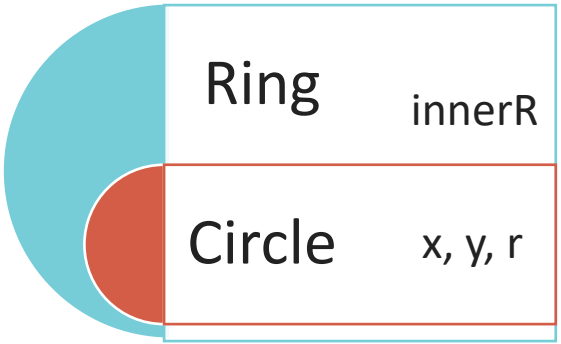
# Перекрытие методов

```
class Circle
{
   public double GetLength()
   {
      return 2 * Math.PI * r;
   }
}
```

```
class Ring : Circle
{
   public new double GetLength()
   {
      return base.GetLength() + 2 *
Math.PI * innerR;
   }
}
```

# Совместимость ссылок при наследовании

```
Circle c = new Ring();
c.R = 5;
c. InnerR = 4; // ошибка!
((Ring)c).InnerR = 4;
```



# Проблемы разработки крупных проектов

Большая команда

Разный уровень разработчиков

Очень много разного кода

# Понятие контрактов

Контракт (договор) – соглашение двух или более лиц, устанавливающее, изменяющее или прекращающее их права и обязанности.

# Интерфейсы

Синтаксис:

```
[спецификатор] interface имя {
        тип имя_метода1 (список_параметров);
        тип имя_метода2 (список_параметров);
        // ...
}
```

Пример

```
interface ISeries {
        double GetCurrent();
        bool MoveNext();
        void Reset();
}
```

# Реализация интерфейсов

```
class ArithmeticalProgression : ISeries
{
    double start, step; int index;
    public ArithmeticalProgression(double start, double step)
    {  this.start = start;
       this.step = step;
       this.index = 1;
    }
    public double GetCurrent()
    { return start + step * index; }
    public bool MoveNext()
    { index++; return true; }
    public void Reset()
    { index = 1; }
}
```

# Реализация интерфейсов

```
class List : ISeries
{
   ...
   public double GetCurrent()
   {
      return current.value;
   }
   public bool MoveNext()
   {
      current = current.next;
      return true;
   }
   public void Reset()
   {
      current = first;
   }
}
```

# Ссылки на интерфейсы

```
static void PrintSeries(ISeries series)
{
   series.Reset();
   for (int i = 0; i < 5; i++)
   {
      Console.WriteLine(series.GetCurrent());
      series.MoveNext();
   }
}
```

```
static void Main(string[] args)
{
   ISeries ser1 = new
ArithmeticalProgression(2, 2);
   PrintSeries(ser1);
   List lst = new List();
   lst.Add(5); lst.Add(8); lst.Add(6);
lst.Add(3); lst.Add(1); lst.Add(8);
   PrintSeries(lst);
}
```

# Интерфейсные свойства и индексаторы

```
interface ISeries
{
    ...
    double StartElement { get; set; }
    double Current { get; }
    double this[int index] { get; }
}
```

# Наследование интерфейсов

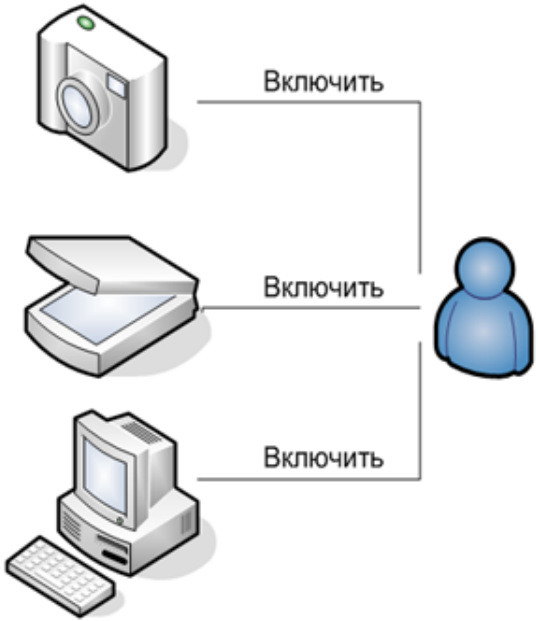
```
interface ISeries
{
    double GetCurrent();
    bool MoveNext();
    void Reset();
}

interface IIndexable
{
    double this[int index] { get; }
}

interface IIndexableSeries : ISeries, IIndexable
{
}
```

# Ограничение при использовании интерфейсов

- ~~Интерфейс не может содержать реализации методов.~~ Уже может, но пользоваться надо с осторожностью.
- Интерфейс не может содержать поля, конструкторы, деструкторы и операторы.
- Ни один член интерфейса не может быть объявлен статическим.
- Все методы интерфейса неявно получают модификатор доступа public. Явное изменение уровня доступа невозможно.

# Полиморфизм

Основная концепция:
«Один интерфейс – множество реализаций».
Одинаковые по сути действия должны располагаться в одноимённых методах.



# Виды полиморфизма

## Статический полиморфизм

- Реализация становится известной на этапе компиляции
- Перегрузка и перекрытие

## Динамический полиморфизм

- На этапе компиляции становится известна группа функций, нужная выбирается на этапе выполнения
- Переопределение

# Статический полиморфизм

- Перегруженные методы – методы с одним именем.
- Могут располагаться как в одном классе, так и в разных классах.

# Переопределение в разных классах

```
class Round
{
  ...
  public double GetArea()
  {
    return Math.PI * r * r;
  }
}
class Rectangle
{
  ...
  public int GetArea()
  {
    return width * height;
  }
}
```

# Использование перегрузки

```
Round round = new Round(10);
Rectangle rect = new Rectangle(10, 10);

Console.WriteLine("Площадь круга = {0}",
round.GetArea());
Console.WriteLine("Площадь прямоугольника = {0}",
rect.GetArea());
```

# Перекрытие

- **Перекрытие** (или сокрытие)– определение новой реализации **невиртуального** метода/свойства класса-предка в классе-потомке.
- Применяется ключевое слово **new**.
- Является частным случаем перегрузки в разных классах.

# Пример перекрытия

```
class Round
{
    public double GetArea()
    {
        return 0;
    }
}
```

```
class Ring : Round
{
    double innerR;

    public new double GetArea()
    {
        return 0;
    }
 }
```

# Динамический полиморфизм

**Переопределение** – определение новой реализации виртуального метода/свойства класса-предка в классе-потомке.
Применяется ключевое слово **override**.

# Пример переопределения

```
class Round
{
    public virtual string Name
    {
        get
        {
            return "Круг";
        }
    }
}
```

```
class Ring : Round
{
    public override string Name
    {
        get
        {
            return "Кольцо";
        }
    }
}
```

# Векторный графический редактор

- Задача: разработать заготовку для простейшего векторного редактора, позволяющего создавать и выводить на экран следующие объекты:
  - прямоугольник
  - круг
  - кольцо

# Векторный графический редактор

## Проблема

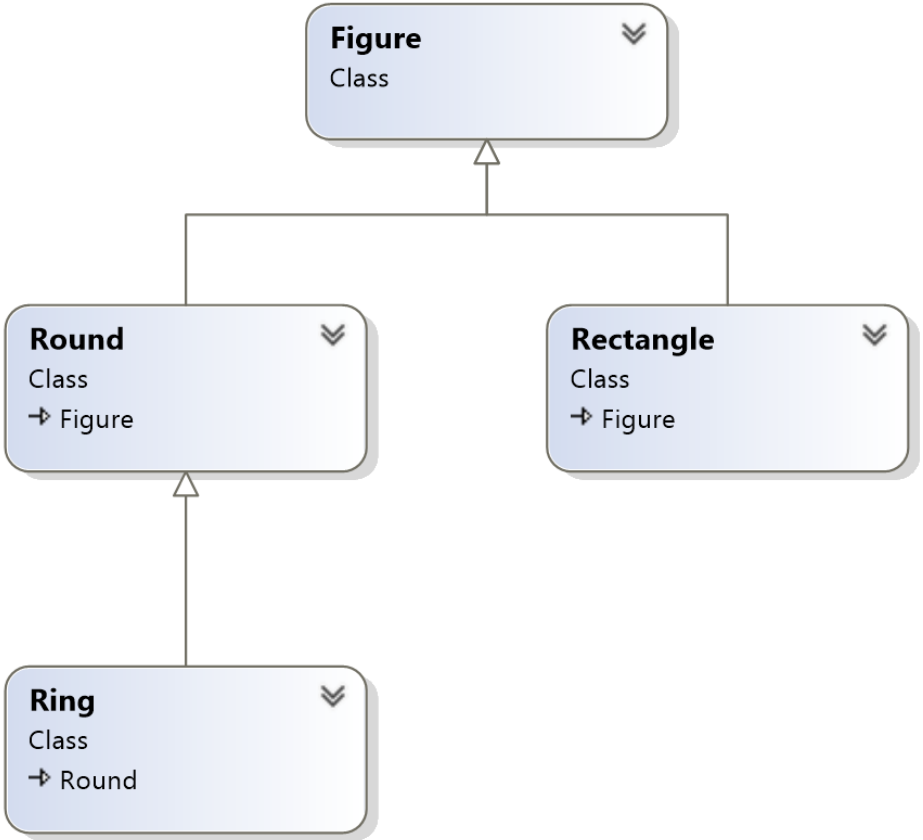
Хранение объектов различных типов в одной коллекции

## Способ решения

Создание базового класса

Использование ссылочной совместимости

# Дерево наследования



# Пример

```
class Figure { }

class Rectangle : Figure
{
   double width, height;
   public Rectangle(double width, double height)
   {
      this.width = width;
      this.height = height;
   }
   public void Draw()
   {
      Console.WriteLine("Это прямоугольник со сторонами {0} и {1}", width,
height);
   }
}
```

# Создание фигур

```
Figure[] fig = new Figure[10];
for (int i = 0; i < fig.Length; i++)
{
   switch (randomGenerator.Next(3))
   {
      case 0:
         fig[i] = new Rectangle(10, 10); break;
      case 1:
         fig[i] = new Round(10); break;
      case 2:
         fig[i] = new Ring(10, 5); break;
   }
}
```

# Отрисовка фигур (статический полиморфизм)

```
for (int i = 0; i < fig.Length; i++)
{
    if (fig[i] is Rectangle)
    {
        ((Rectangle)fig[i]).Draw();
    }
    else if (fig[i] is Ring)
    {
        ((Ring)fig[i]).Draw();
    }
    else if (fig[i] is Round)
    {
        ((Round)fig[i]).Draw();
    }
}
```

# Позднее связывание

- На этапе компиляции ничего не известно о том, объект какого класса будет храниться по ссылке.
- При вызове метода указывается только семейство методов.
- Среда исполнения сама вызывает нужный метод после того, как во время работы программы выяснится, объект какого класса лежит по ссылке.

# Реализация позднего связывания

- Между классами должно быть установлено отношение наследования.
- Метод, который предполагается сделать виртуальным, должен быть объявлен в базовом классе с ключевым словом virtual.
- В потомках этот метод должен быть переопределён с ключевым словом override.

# Применение позднего связывания

Базовый класс

```
class Figure
{
        public virtual void Draw()
        {
        }
}
```

Вывод фигур

```
for (int i = 0; i < fig.Length; i++)
{
        fig[i].Draw();
}
```

# Применение позднего связывания

Потомки

```
class Rectangle : Figure
{
        ......
        public void Draw()
        {
                Console.WriteLine("Это прямоугольник со сторонами {0} и {1}", width, height);
        }
}
```

# Абстрактные классы

- Создать объект абстрактного класса нельзя.
- Абстрактные классы могут содержать нереализованные абстрактные методы.
- В абсолютном большинстве случаев классы, обладающие потомками, должны быть абстрактными.

# Абстрактные классы

Объявление

```
abstract class имя_класса
{
    ......
}
```

Пример

```
abstract class Figure
{
    public virtual void Draw() { }
}
```

# Абстрактные методы

Абстрактным называется метод, не имеющий тела.
Могут располагаться только в абстрактных классах и интерфейсах.
Должны быть перегружены с реализацией в потомке.

```
abstract class Figure
{
    public abstract virtual void Draw()
}
```

# Что почитать

- Эндрю Троелсен – “C# и платформа .NET”
- Кристиан Нейгел, Билл Ивьен, Джей Глинн, Морган Скиннер, Карли Уотсон – “C# 2005 и платформа .NET 3.0”
- Роберт Мартин, Мика Мартин – “Принципы, паттерны и методики гибкой разработки на языке C#”

# СПАСИБО ЗА ВНИМАНИЕ