Блок 2. Основы ООП

План занятия

- Принципы ООП
- Концепции ООП
- Принципы SOLID

Шесть принципов Алана Кэя

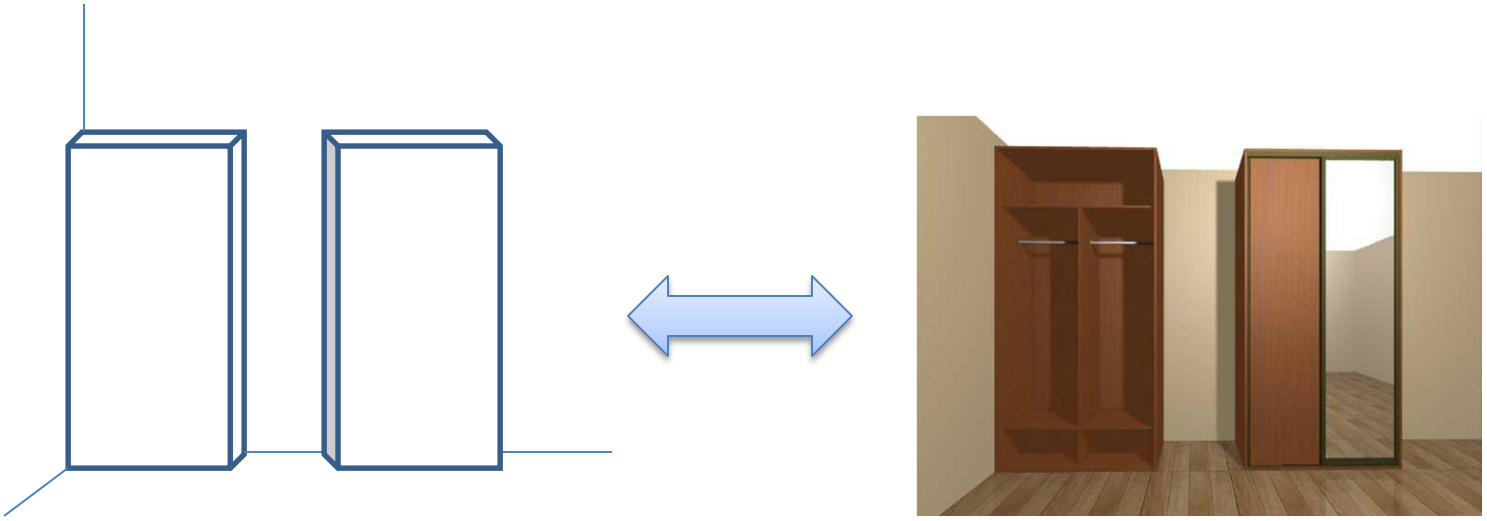
1. Всё является объектом;
2. Каждый объект является экземпляром класса;
3. Класс определяет поведение объекта;
4. Классы организованы в иерархию наследования;
5. Каждый объект обладает независимой памятью;
6. Вычисления производятся путём взаимодействия между объектами.

Alan Curtis Kay

Author of Smalltalk
Founder of OOP

1. Всё является объектом:

- Любая сущность может быть описана в виде законченного объекта;
- Набор свойств определяется исходя из потребностей приложения.



2. Каждый объект является экземпляром, или представителем класса:

- Класс выражает общие свойства объектов.

3. Класс определяет поведение объекта:

- Поведение, или функциональность объекта задаётся в классе;
- Все объекты, являющиеся экземплярами одного и того же класса, могут выполнять одни и те же действия.

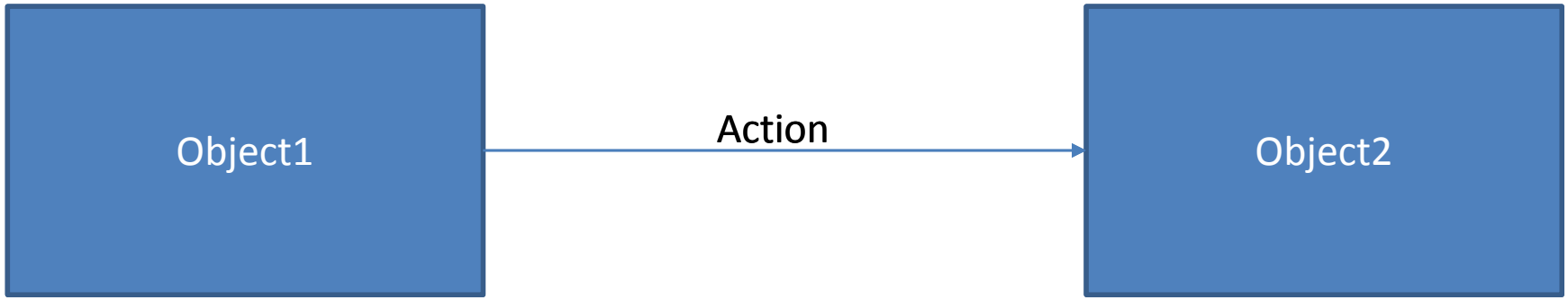
4. Классы организованы в иерархию наследования:

- Иерархия представляет собой единую древовидную структуру с общим корнем;
- Память и поведение, связанные с экземплярами определённого класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

5. Каждый объект обладает независимой памятью:

- Память объекта состоит из других объектов;
- Вложенные (агрегированные) объекты не имеют доступа к память основного (агрегата).

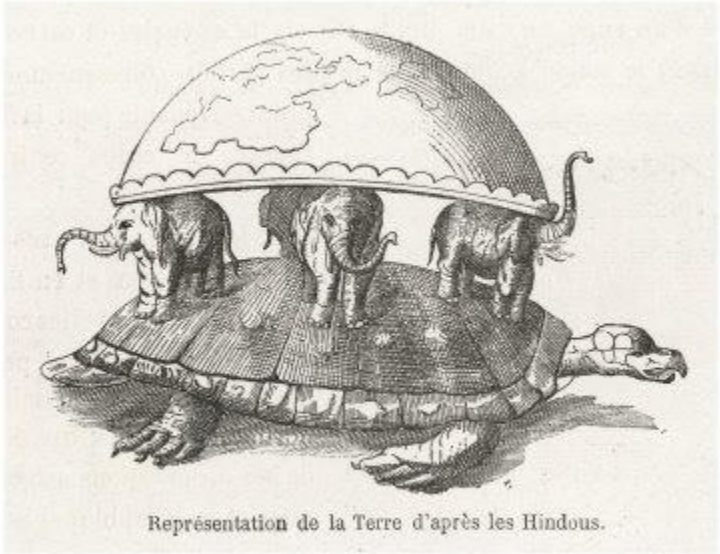
6. Вычисления осуществляются путём взаимодействия между объектами



Концепции ООП

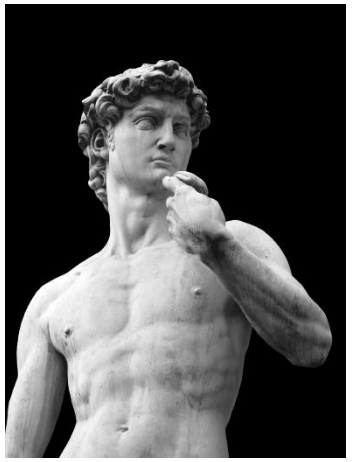
- Инкапсуляция
- Наследование
- Полиморфизм

- Абстракция

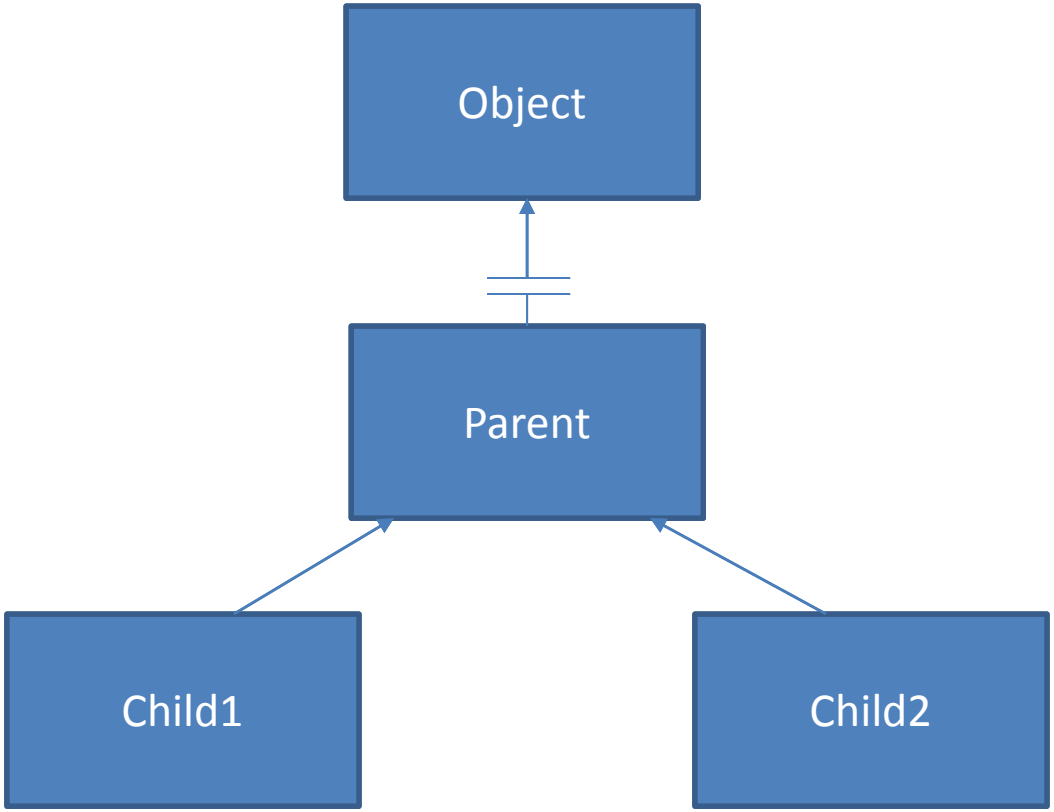


Абстракция (abstraction)

- Формирование объекта (класса) как совокупности свойств и поведения:
  - Выделяйте только те факторы, которые нужны для решения задачи;
  - Отсекайте всё лишнее.

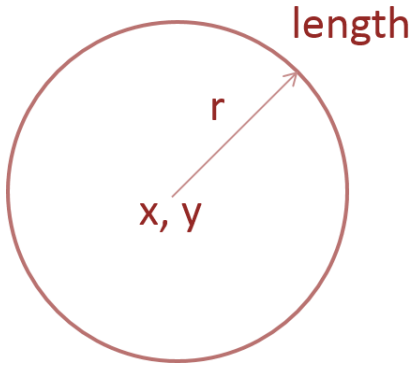


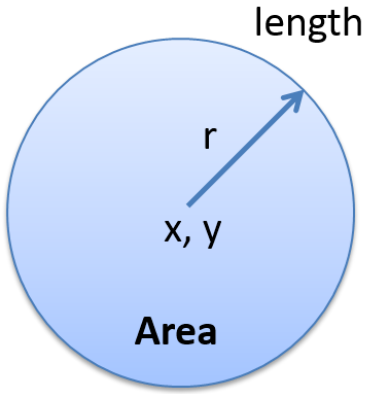
Возможность создания классов на основе других, ранее написанных



Правила наследования

- Наследование синонимично расширению:
  - Каждый объект потомка является расширенной, уточнённой версией предка;
  - Потомок не имеет права на ограничение характеристик и/или поведения предка.

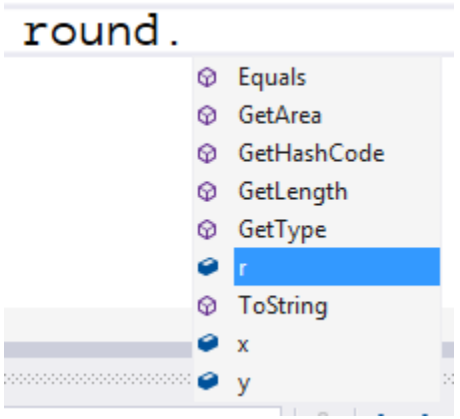




Пример наследования

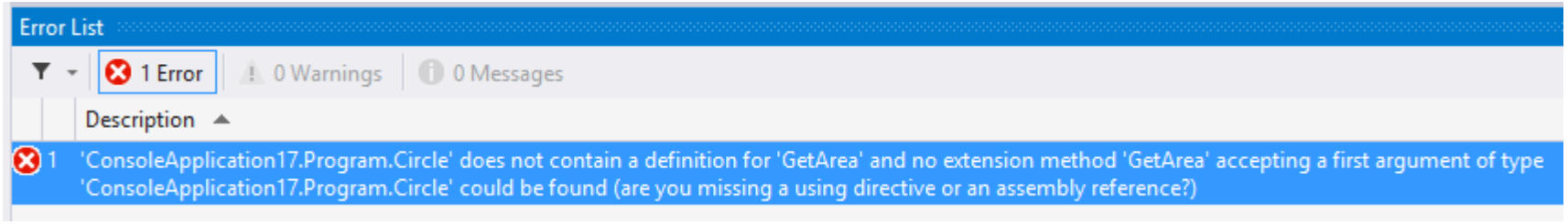
```
public class Circle
{
    public int x;
    public int y;
    public int r;

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


public class Round : Circle
{
    public double GetArea()
    {
        return Math.PI * r * r;
    }
}
```

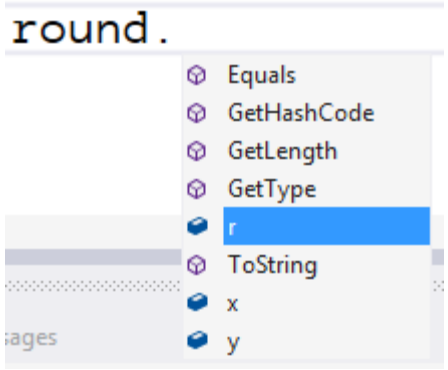
```
var round = new Round();
round.r = 5;
var length = round.GetLength();
var area = round.GetArea();
```



Совместимость ссылок при наследовании

```
Circle round = new Round();
round.r = 5;
var length = round.GetLength();
var area = round.GetArea();

area = ((Round)round).GetArea();
```





Агрегация

- Хранение объектов в полях других объектов;
- Рассматривается как альтернатива наследованию в ситуациях, когда наследование невозможно или нежелательно:
  - класс уже является потомком третьего класса;
  - класс должен отличаться от агрегируемого интерфейсом или поведением;
  - объект класса использует возможности ранее созданного объекта агрегируемого класса;
  - наследование от агрегируемого класса запрещено (модификатор **sealed** или скрыт конструктор).

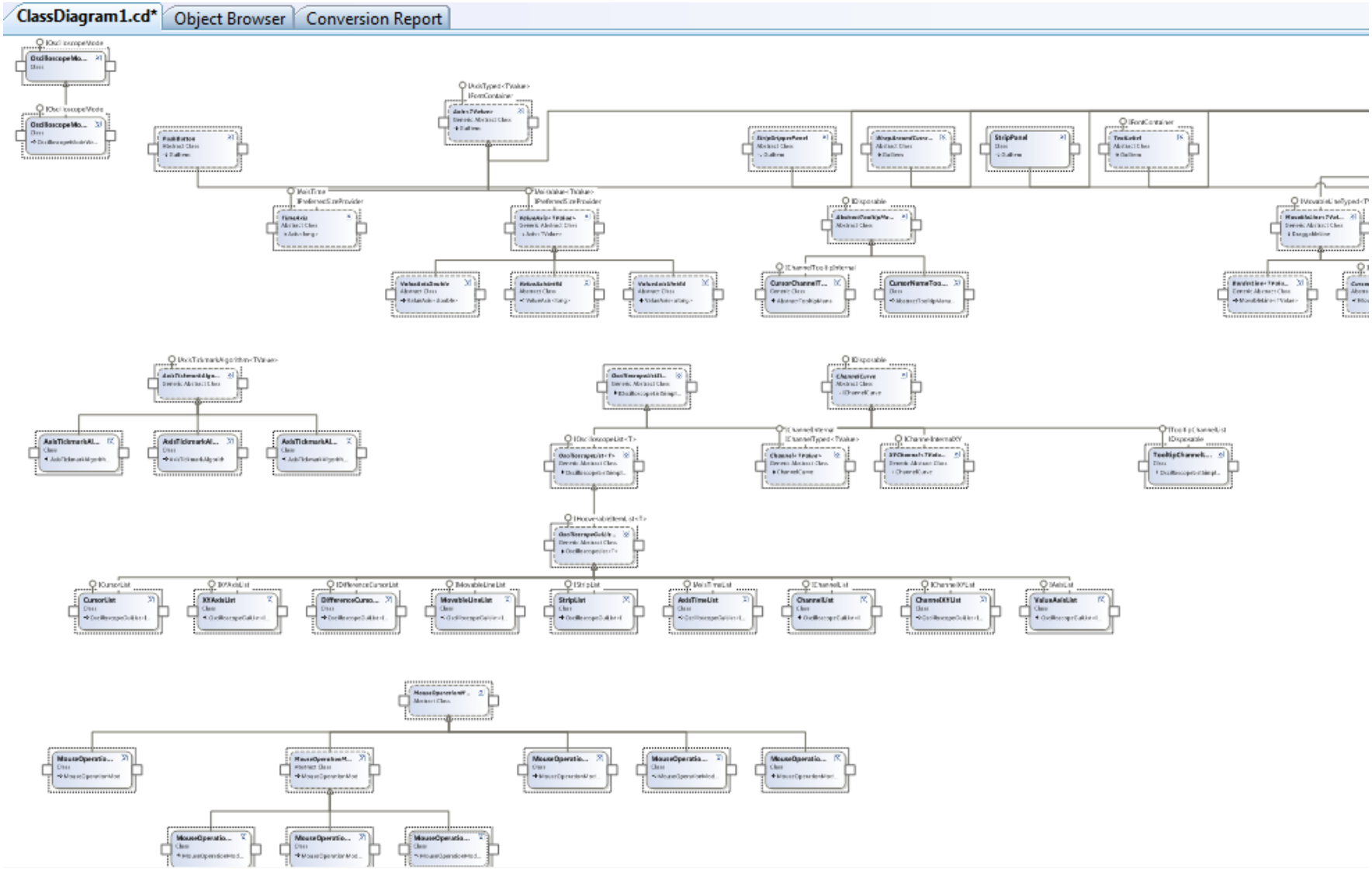
Пример агрегации

```
public class Ring
{
    public int x;
    public int y;
    public Round inner;
    public Round outer;

    public double GetArea()
    {
        return outer.GetArea() - inner.GetArea();
    }
}


var ring = new Ring()
{
    inner = new Round { r = 5 },
    outer = new Round { r = 10 },
};
var area = ring.GetArea();
```

Наследование или агрегация?

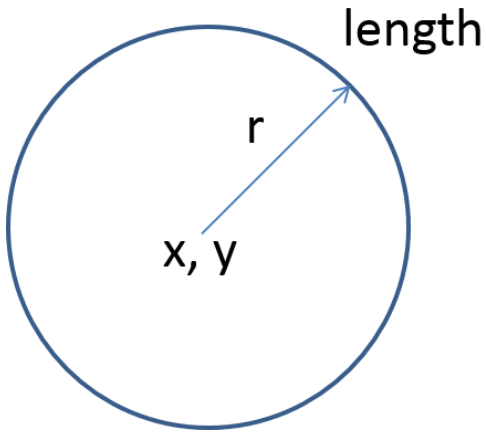
- Круг и окружность
- Квадрат и прямоугольник
- Прямоугольник и квадрат ☺
- Эллипс (вписанный в прямоугольник) и прямоугольник
- Круг и точка
- Шкаф и объект мебели

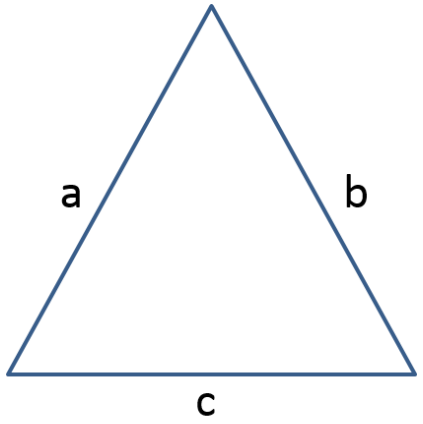
Фрагмент диаграммы классов реального проекта



Инкапсуляция

- Отделение внешнего интерфейса объекта от его внутренней реализации:
  - Пользователь не должен менять внутреннее состояние объекта;
  - Поля и методы делятся на внутренние (имплементация) и внешние (интерфейсные);





Задачи, решаемые инкапсуляцией

- Сокрытие реализации;
- Поддержание объекта в заведомо корректном состоянии;
- Решение проблемы связанных полей;
- Упрощение интерфейса.

Средства для реализации инкапсуляции

- Спецификаторы доступа
- Свойства
- Конструкторы
- Методы
- Индексаторы

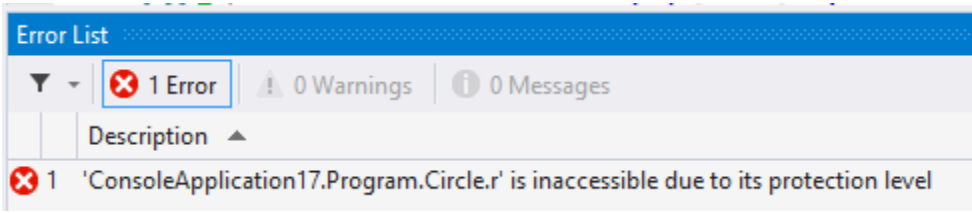
Спецификаторы доступа

- **private** — доступ только из текущего класса;
- **protected** — доступ только из текущего класса и из его потомков;
- **internal** — доступ только из классов текущей сборки (проекта);
- **protected internal** — доступ из классов текущей сборки, а также из потомков класса;
- **public** — полный неограниченный доступ из любой точки приложения.

Инкапсуляция поля парой методов

```
private int r;

public int GetRadius()
{
    return r;
}

public void SetRadius(int value)
{
    if (value > 0)
    {
        r = value;
    }
}
```

```
var circle = new Circle();
circle.r = 10;
circle.SetRadius(10);
var radius = circle.GetRadius();
```



Свойства

- Свойство — это пара методов, предназначенных для правильной инкапсуляции поля:
  - **get** (accessor, или getter) — предназначен для получения значения;
    - не должен приводить к исключениям;
    - не должен выполнять длительных вычислений;
  - **set** (mutator, или setter) — предназначен для установки значения;
    - должен производить валидацию передаваемого значения и генерировать исключение при необходимости;
    - не должен выполнять длительных вычислений.

Свойства в C# 1.0

[спецификатор] тип имя_свойства
{
    get { тело }
    set { тело }
}

```
private double r;

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

Свойства в C# 2.0

[спецификатор] тип имя_свойства
{
    [[спецификатор] get { тело }]
    [[спецификатор] set { тело }]
}

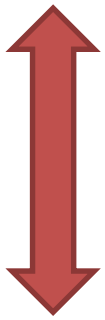
```
private double r;

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

Свойства в C# 3.0

```
private int r;

public int Radius
{
    get { return r; }
    set { r = value; }
}
```

```
public int Radius { get; set; }
```

```
private int r;

public int Radius
{
    get { return r; }
    private set { r = value; }
}
```

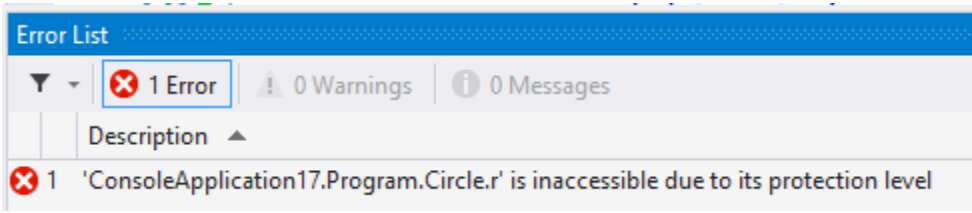


```
public int Radius { get; private set; }
```

Пример инкапсуляции поля свойством

```
private int r;

public int Radius
{
    get
    {
        return r;
    }
    set
    {
        if (value > 0)
        {
            r = value;
        }
    }
}
```

```
var circle = new Circle();
circle.r = 10;
circle.Radius = 10;
var radius = circle.Radius;
```



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);
```

Оптимизированное решение

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

Конструктор

- Специальный метод, предназначенный для выделения памяти под объект и инициализации его полей;

Деструктор и финализатор

- Деструктор в ООП — специальный метод, предназначенный для удаления объекта и освобождения занятой им памяти;
- Финализатор в ООП — специальный метод, вызывающийся непосредственно перед удалением объекта для освобождения связанных с ним ресурсов;
- Деструктор в C# использует синтаксис деструктора C++, однако по сути является финализатором;
- Если процесс завершится до того, как объектом займётся сборщик мусора, деструктор не будет вызван.

```
~Circle()
{
    // освобождение ресурсов
}
```

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;
}
```

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 innerR, int outerR)
        : base(outerR)
    {
        this.innerR = innerR;
    }
}
```

Отложенная загрузка (Lazy Loading)

```
public class MyClass
{
    private MyData myData;

    public MyClass()
    {
        myData = null;
    }

    public MyData Data
    {
        get
        {
            if (myData == null)
            {
                myData = DataLoader.Load();
            }
            return myData;
        }
    }
}
```

Сокрытие конструктора

```
var circle = Circle.Create(10);
```

```
public class Circle
{
    public int X { get; set; }

    public int Y { get; set; }

    public int Radius { get; private set; }

    // Наличие явно заданного (хоть и недоступного) конструктора
    // скрывает неявный конструктор по умолчанию
    private Circle()
    {
    }

    public static Circle Create(int radius)
    {
        if (radius <= 0)
        {
            throw new ArgumentException("Radius should be positive");
        }

        return new Circle { Radius = radius };
    }
}
```

Спецификатор static

- Статические поля

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

- Статические методы

```
static void Method1() { /* ... */ }
MyClass.Method1();
```

- Статические конструкторы

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

- Статические классы

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

Операторы

Унарные

+ - ! ~

++ --

true false

Бинарные

+ - * / %

& | ^

<< >>

== != < >
<= >=

Операции приведения типов

implicit

explicit

Индексаторы

[ ]

Унарные операторы

- Синтаксис:
  static TResult **operator** OpSign**(**MyClass obj**)**
  - TResult — тип результата
  - OpSign — знак оператора

```
static ResultClass operator +(MyClass obj)
{
    return new ResultClass(obj);
}
```

- Тип значения, возвращаемого операторами инкремента и декремента (++, --) должен совпадать с типом аргумента.
- Операторы **true** и **false** должны возвращать результат типа bool.

Бинарные операторы

- Синтаксис:
  static TResult **operator** OpSign**(**MyClass1 obj, MyClass2 obj**)**
  - TResult — тип результата
  - OpSign — знак оператора

```
static ResultClass operator *(MyClass obj1, MyClass obj2)
{
    return new ResultClass(obj1, obj2);
}
```

Операции приведения типов

- Неявное приведение

```
int i = 5;
double d = i;

static implicit operator ResultClass(MyClass obj)
{
    return new ResultClass(obj);
}
```

- Явное приведение

```
double d = 5.5;
int i = (int)d;

static explicit operator ResultClass(MyClass obj)
{
    return new ResultClass(obj);
}
```

Пример использования операторов приведения

- Вектор на плоскости описывается двумя компонентами: абсциссой и ординатой (X, Y);
- Вектор в пространстве описывается тремя координатами: абсциссой, ординатой и аппликатой (X, Y, Z);
- Любой вектор на плоскости может быть безболезненно преобразован в вектор в пространстве;
- Любой вектор в пространстве может быть преобразован в проекцию на плоскость с потерей аппликаты.

Пример применения операторов приведения

- Приведение типов без потери данных реализуется неявным приведением (implicit operator);
- Приведение типов с потерей данных реализуется явным приведением (explicit operator).

```
static void Main()
{
    var v2d = new Vector2D { X = 5, Y = 6 };
    var v3d = new Vector3D { X = 7, Y = 8, Z = 9 };

    Vector3D res3d = v2d;
    Vector2D res2d = (Vector2D)v3d;
}
```

Пример реализации операторов приведения

```
class Vector2D
{
    public int X { get; set; }
    public int Y { get; set; }
}

class Vector3D
{
    public int X { get; set; }
    public int Y { get; set; }
    public int Z { get; set; }

    public static implicit operator Vector3D(Vector2D v2d)
    {
        return new Vector3D { X = v2d.X, Y = v2d.Y };
    }

    public static explicit operator Vector2D(Vector3D v3d)
    {
        return new Vector2D { X = v3d.X, Y = v3d.Y };
    }
}
```

Индексаторы

- Гибрид свойства и пары методов с параметрами
  - От свойств: явно указываются блоки get и set, любой из них можно опустить
  - От методов: тип параметров и возвращаемого значения может быть любым

```
class Circle
{
    public Point this[double angle]
    {
        get { return new Point(Math.Cos(angle), Math.Sin(angle)); }
    }

    public double this[double x, double y]
    {
        get { return Math.Atan2(y, x); }
    }
}
```

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

Правила реализации интерфейса

- Реализация интерфейса синтаксически выглядит как наследование (через **:**);
- Класс обязан реализовать все сигнатуры реализуемого интерфейса одним из способов:
  - неявная реализация публичным (public) экземплярным членом класса;
  - явная реализация с указанием имени интерфейса;
- Допускается одновременная реализация нескольких интерфейсов.

Пример неявной и явной реализации интерфейса

```
interface IPrintable
{
    void Print();
}

class ImplicitImplementation : IPrintable
{
    public void Print()
    {
    }
}

class ExplicitImplementation : IPrintable
{
    void IPrintable.Print()
    {
    }
}
```

Состав интерфейсов

- Могут содержать:
  - Сигнатуры методов
  - Сигнатуры свойств
  - Сигнатуры индексаторов
  - События
- Не могут содержать:
  - Поля и константы
  - Конструкторы и деструкторы
  - Операторы
  - Статические члены
  - Реализации методов и т.п.
  - Явно указанные модификаторы доступа

Переменная интерфейсного типа

```
public void PrintAll(IPrintable[] objects)
{
    foreach (var item in objects)
    {
        item.Print();
    }
}
```

```
IPrintable object1 = new ExplicitImplementation();
IPrintable object2 = new ImplicitImplementation();

IPrintable[] objects = new IPrintable[2] { object1, object2 };

PrintAll(objects);
```

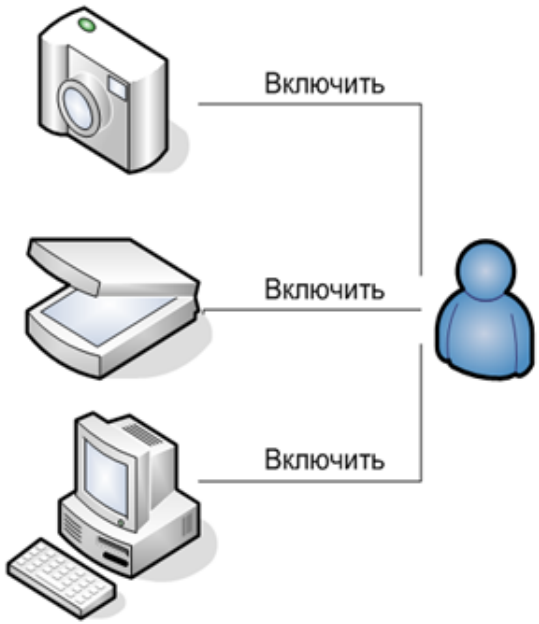
Наследование интерфейсов

- Интерфейсы могут **наследоваться** от других интерфейсов;
- Допустимо множественное наследование интерфейсов;
- Класс, **реализующий** интерфейс-потомок, должен реализовать и всех его предков.

```
interface ILocatable
{
    public int X { get; set; }
    public int Y { get; set; }
}

interface IGameObject : ILocatable, IPrintable
{
}
```

Полиморфизм

- Основная концепция: «Один интерфейс — множество реализаций»;
- Однотипные действия должны быть доступны через однотипный подход.



Виды полиморфизма

- Статический
  - Реализация выбирается на этапе компиляции.
  - Пример: перегрузка, перекрытие, обобщения.
- Динамический
  - Реализация выбирается на этапе выполнения.
  - Пример: переопределение.

Статический полиморфизм

- Перегруженные методы — методы с одним именем. Могут располагаться как в одном классе, так и в разных классах;
- Предполагается, что перегруженные методы выполняют схожие действия;
- Частный случай перегрузки — перекрытие методов. Возникает при наличии у класса-предка и класса-потомка метода с идентичной сигнатурой. Рекомендуется использование ключевого слова **new**.

Перегрузка в одном классе

```
class Rectangle
{
    private int width, height;

    public Rectangle()
        : this(1, 1)
    {
    }

    public Rectangle(int width, int height)
    {
        this.width = width;
        this.height = height;
    }
}
```

```
var rect1 = new Rectangle(3, 4);
var rect2 = new Rectangle();
```

Перекрытый метод

```
class Circle
{
    ...

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

class Ring : Circle
{
    ...

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

Вызов перекрытого метода

```
Ring r = new Ring(3, 5);

// r объявлен как Ring, поэтому вызывается Ring.GetLength
double totalLength = r.GetLength();

// r явно приводится к Circle, вызывается Circle.GetLength
double outerLength = ((Circle)r).GetLength();

Console.WriteLine("Total: " + totalLength);
Console.WriteLine("Outer: " + outerLength);
```

```
Total: 50.2654824574367
Outer: 31.4159265358979
```

Динамический полиморфизм

- **Переопределение** — определение новой реализации виртуального метода/свойства класса-предка в классе-потомке;
- Виртуальные методы/свойства помечаются ключевым словом **virtual**;
- Новые реализации методов/свойств помечаются ключевым словом **override**;
- По умолчанию новые реализации также являются виртуальными, если не указано ключевое слово **sealed**.

Пример переопределения

```
class Round
{
    ...

    public virtual string Name
    {
        get { return "Круг"; }
    }
}

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

Обращение к предыдущей реализации

- При переопределении метода/свойства/… есть возможность обратиться к предыдущей реализации при помощи префикса base.

```
class Ring : Round
{
    public int InnerRadius { get; set; }

    public override string Name
    {
        get
        {
            if (InnerRadius == 0) { return base.Name; }

            return "Кольцо";
        }
    }
}
```

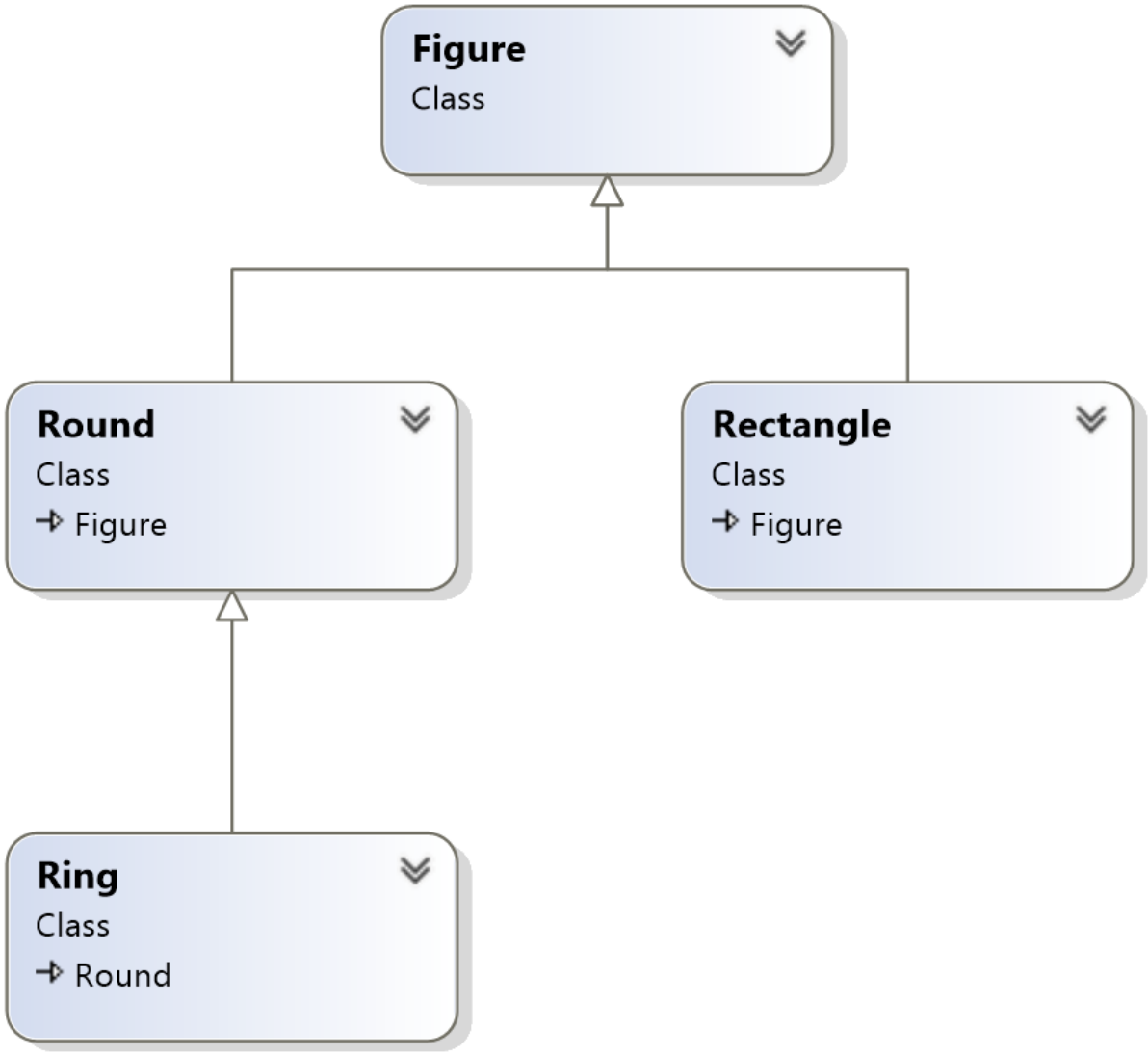
Векторный графический редактор

- Задача: разработать заготовку для простейшего векторного редактора, позволяющего создавать и выводить на экран следующие объекты:
  - прямоугольник;
  - круг;
  - кольцо;
- Какие проблемы могут возникнуть при разработке заготовки?

Векторный графический редактор

- Проблема:
  - Хранение объектов различных типов в одной коллекции (массиве);
- Способ решения:
  - Создание базового класса;
  - Использование ссылочной совместимости.

Дерево наследования



Позднее связывание

- На этапе компиляции ничего не известно о том, объект какого типа будет храниться по ссылке;
- Среда исполнения сама вызывает нужную реализацию метода в зависимости от конкретного типа объекта.

Абстрактные члены класса

- Это члены класса, не имеющие конкретной реализации в данном классе;
- Класс, содержащий хотя бы один абстрактный член, должен стать абстрактным;
- Объект абстрактного класса создать нельзя (можно использовать класс в качестве типа переменной для ссылочной совместимости);
- Потомки абстрактного класса должны либо реализовать (переопределить) его абстрактные члены, либо также стать абстрактными;
- Абстрактные члены класса и классы помечаются ключевым словом **abstract**.

Пример реализации абстрактного метода

- Реализация абстрактных методов в потомках происходит посредством переопределения с ключевым словом **override**, как и для виртуальных методов.

```
abstract public class Figure
{
    abstract public double Area { get; }
}

public class Round : Figure
{
    public int Radius { get; set; }

    public override double Area
    {
        get { return Math.PI * Radius * Radius; }
    }
}
```

Интерфейсы и абстрактные классы

- Класс может иметь только одного предка, но реализовывать сколько угодно интерфейсов.

```
public class Rectangle : Figure, IDrawable, IRotatable
{
    // ...
}
```

- Абстрактный класс может содержать реализованные методы/свойства, конструкторы и т.п.
- Абстрактный класс может являться потомком другого класса, в том числе и абстрактного.

Принципы SOLID от Роберта Мартина

S	SRP — Single Responsibility Principle
O	OCP — Open/Closed Principle
L	LSP — Liskov Substitution Principle
I	ISP — Interface Segregation Principle
D	DIP — Dependency Inversion Principle

S

SRP – Single Responsibility Principle

- Принцип единственной ответственности:

«Каждый модуль должен обладать только одним предназначением»

O

OCP – Open/Closed Principle

- Принцип открытости/закрытости:

«Сущности должны быть открыты для расширения, но закрыты для модификации»

L

Liskov Substitution Principle

- Принцип подстановки Барбары Лисков:

«Объекты классов должны легко заменяться на экземпляры любых своих потомков без нарушения функциональности»

I

ISP – Interface Segregation Principle

- Принцип разделения интерфейсов:

«Множество специфичных, ориентированных на конкретные задачи, интерфейсов лучше одного сверх-интерфейса»

D

DIP – Dependency Inversion Principle

- Принцип инверсии зависимостей:

«Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций»

«Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций»

Источники

- 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#»