ЛЕКЦИЯ 7

- Модуль 2
- 22.11.2023
- Отношения между классами

ЦЕЛИ ЛЕКЦИИ

- Познакомиться с отношениями между классами
- Разобраться с особенностями реализации этих отношений на языке C#

ОТНОШЕНИЕ НЕЗАВИСИМОСТИ

НЕЗАВИСИМОСТЬ КЛАССОВ

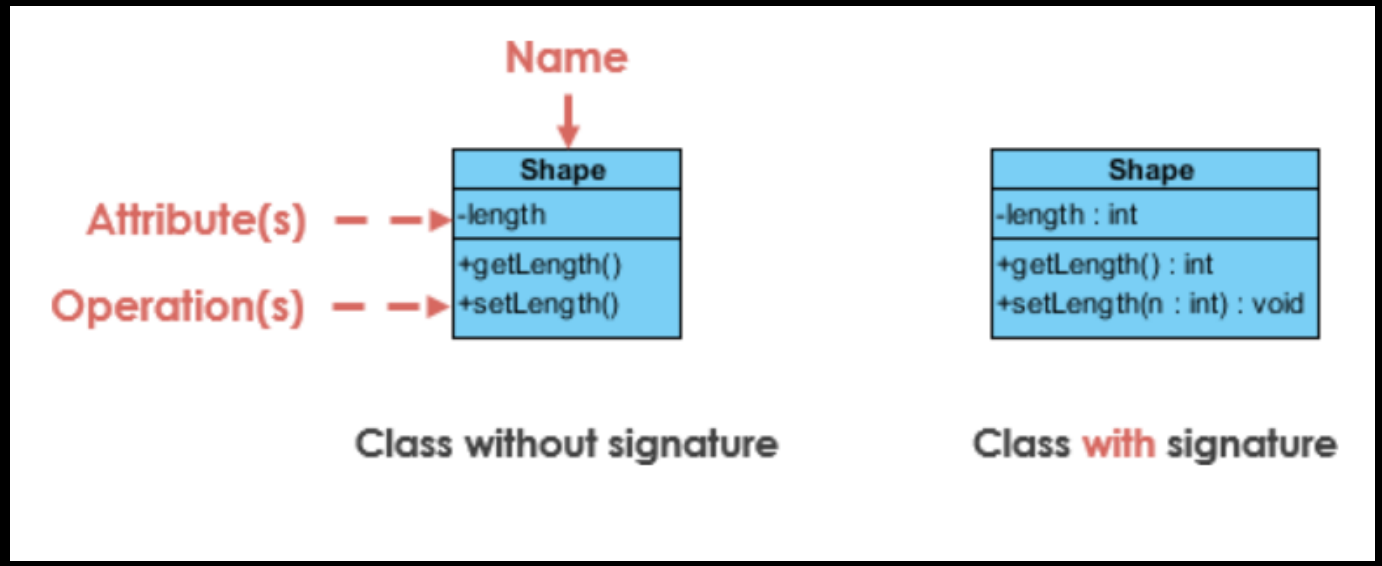
- Влечёт независимость порождаемых объектов этих классов

```
public class Cat {
    public string ToString() {
        return "Кошка гуляет сама по себе";
    }
}
```

```
public class Dog {
    public string ToString() {
        return "Собака - друг человека";
    }
}
```

```
class Program {
    static void Main() {
        Console.WriteLine((new Cat()).ToString());
        Console.WriteLine((new Dog()).ToString());
    }
}
```

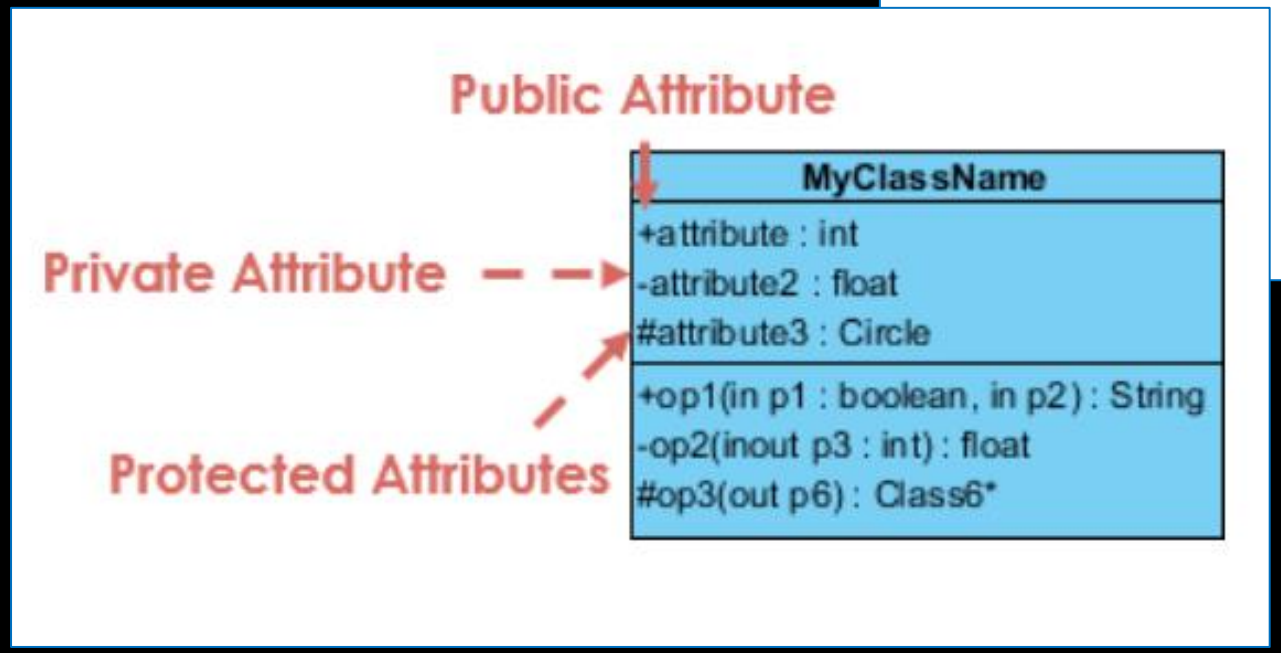
НЕМНОГО О НОТАЦИИ UML (1)

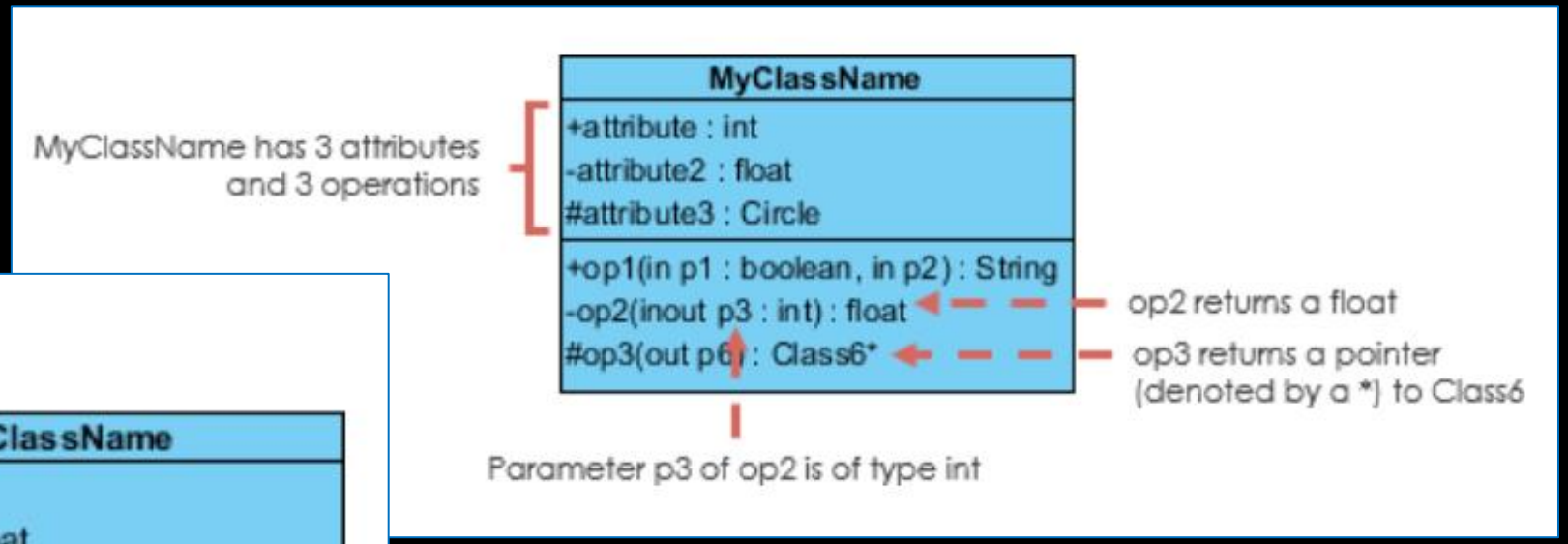


У каждого атрибута есть тип
У каждого элемента поведения есть сигнатура

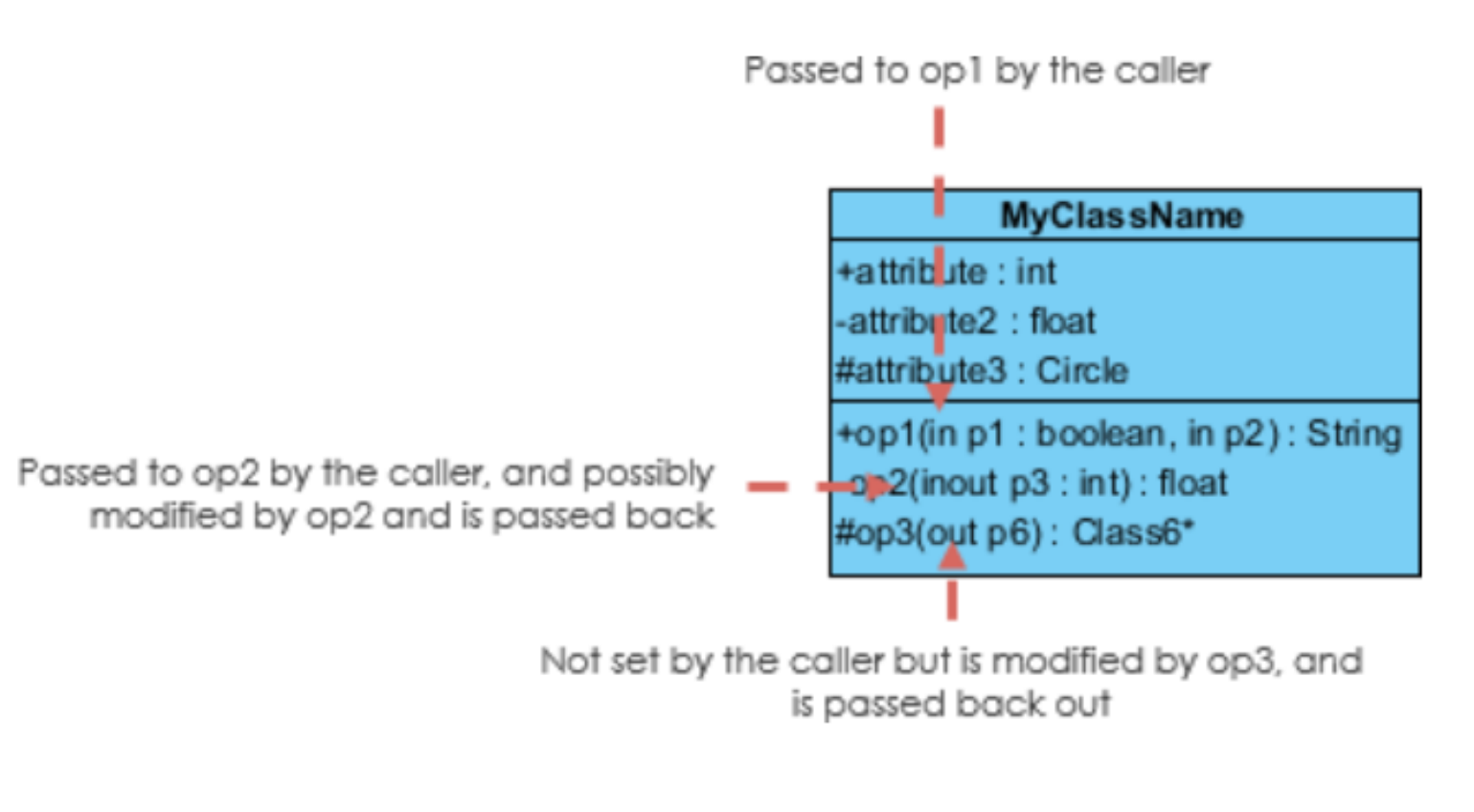
Attribute(s) – инкапсулируют состояние
Operation(s) - поведение

НЕМНОГО О НОТАЦИИ UML (2)

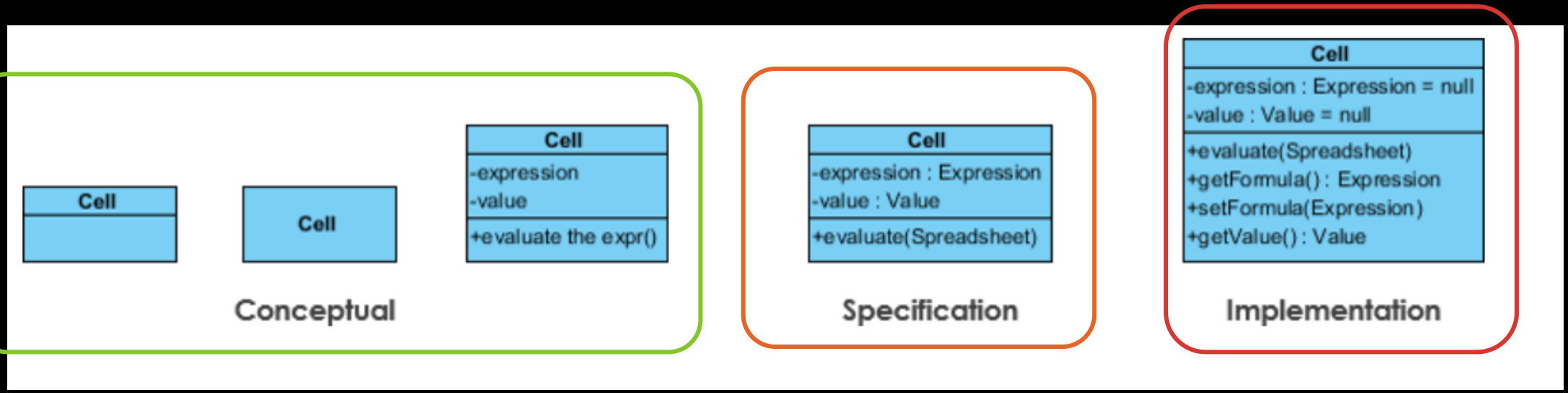




НЕМНОГО О НОТАЦИИ UML (3)

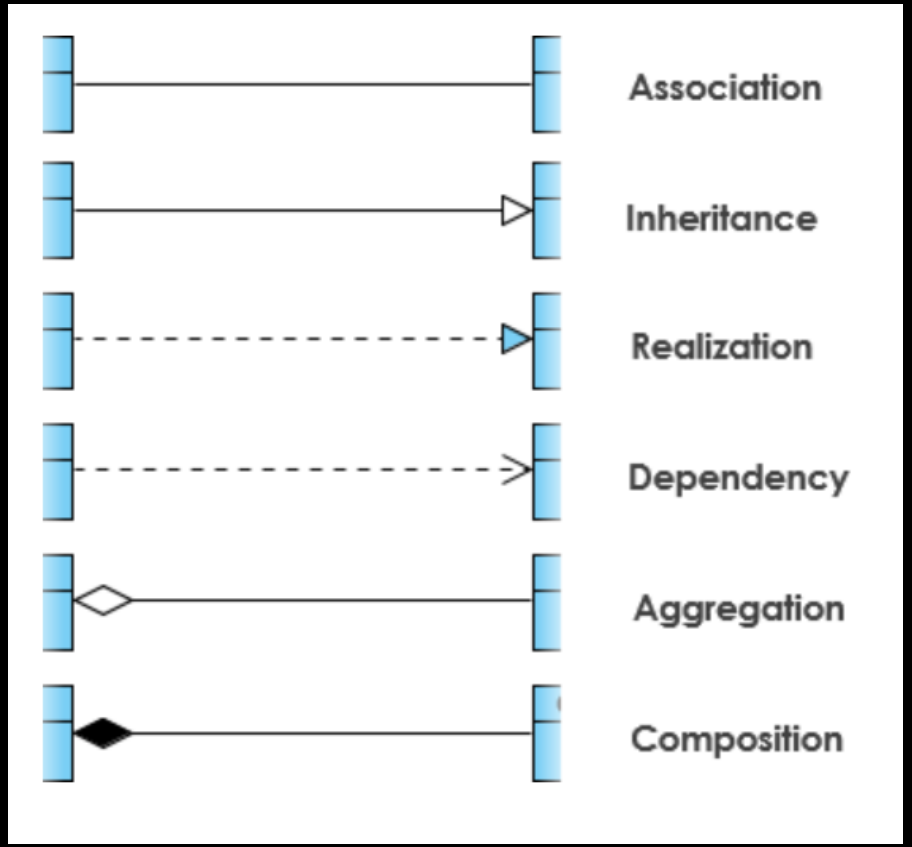


НЕМНОГО О НОТАЦИИ UML (4)



ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ

- Независимость
- Включение
  - Агрегация
  - Композиция
  - Ассоциация
- Вложение
- Наследование
- Инстанцирование



ОТНОШЕНИЕ ВКЛЮЧЕНИЯ

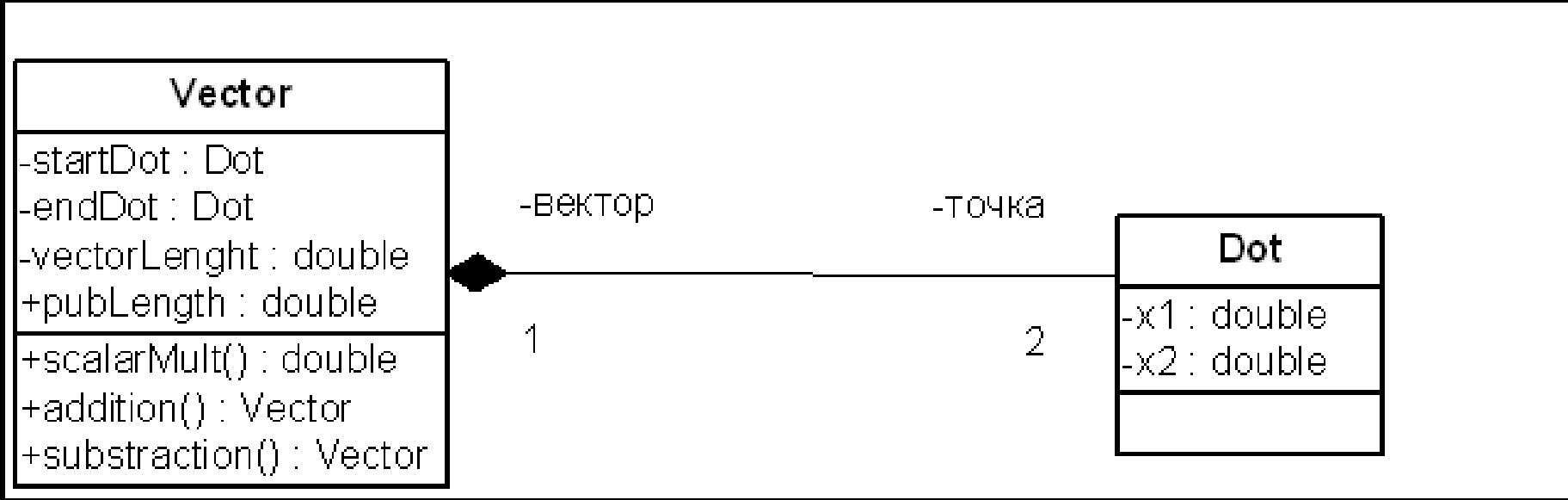
Агрегация и композиция

АГРЕГАЦИЯ (ВКЛЮЧЕНИЕ)

- **Агрегация** [aggregation] является отражением отношения «являться частью» или «часть/целое» (**HAS A**)
  - Один класс включается в состав другого
  - В классе объявляется поле, тип которого является другим классом
- **Композиция** (композитная агрегация) [composition] означает временную зависимость, когда агрегат не может существовать без своих компонентов

Между созданием/уничтожением включённого объекта агрегатом и присваиванием полю ссылки на уже существовавший ранее «внешний» объект – существенная разница

КОМПОЗИЦИЯ КЛАССОВ



ПРИМЕР РЕАЛИЗАЦИИ

```
public class Dot {
    public double X1 { get; set; }
    public double X2 { get; set; }
}
public class Vector {
    private Dot _startDot;
    private Dot _endDot;
    private double vectorLenght;
    public double pubLenght;
    public double ScalarMult() { return 0.0; }
    public Vector Addition() { return new Vector();}
    public Vector Substraction() { return new Vector(); }
}
```

Каким должен быть конструктор класса Vector?

```
public Vector (double x1, double x2, double x3, double x4)
{
    _startDot = new Dot();
    _startDot.X1 = x1;
    _startDot.X2 = x2;
    _endDot = new Dot();
    _endDot.X1 = x3;
    _endDot.X2 = x4;
} // composition
```

```
public Vector(Dot start, Dot end)
{
    _startDot = start;
    _endDot = end;
} // aggregation
```

https://replit.com/@olgamaksimenkova/AggregVSComposConstructor

ТИП «ТОЧКА НА ПЛОСКОСТИ»

1) Класс + Свойство

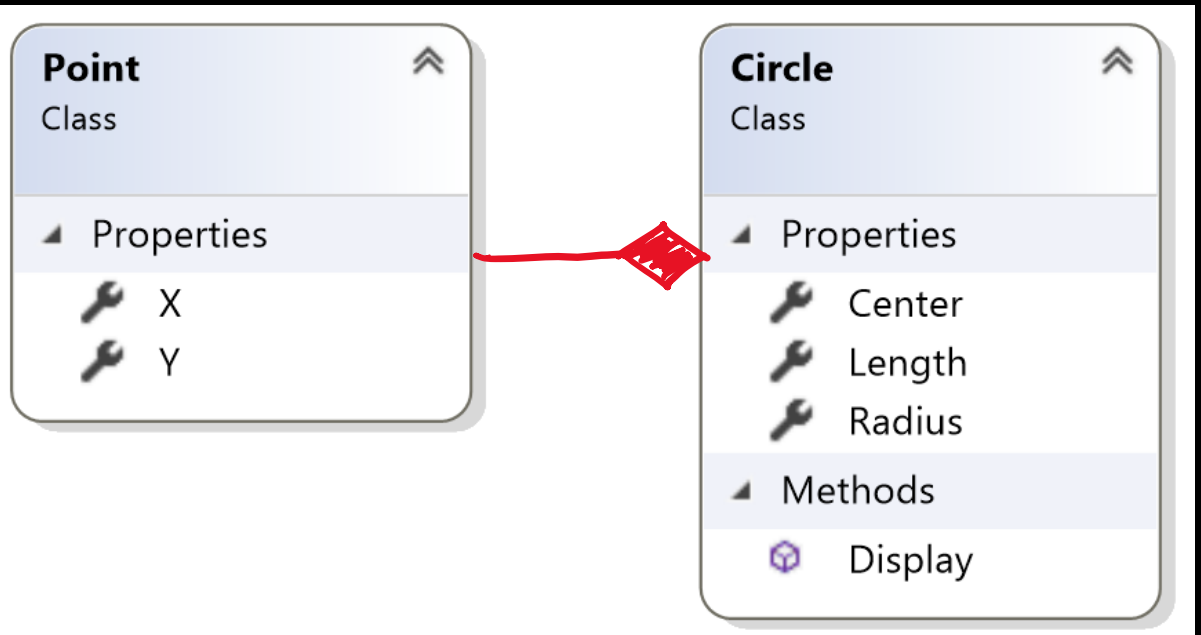
```
public class Point
{
    private double _x, _y;
    public double X { get { return _x; } init { _x = value; } }
    public double Y { get { return _y; } init { _y = value; } }
}
```

2) Класс + Авто. Свойство

```
public class Point
{
    public double X { get; init; }
    public double Y { get; init; }
}
```

КОМПОЗИЦИЯ КЛАССОВ

```
public class Point
{
    public double X { get; set; }
    public double Y { get; set; }
}

public class Circle
{
    public double Radius { get; set; }
    public double Length => 2 * Radius * Math.PI;
    public Point Center { get; set; } = new Point();

    public void Display()
       => Console.WriteLine($"Center: X = {Center.X}, Y = {Center.Y}; " +
                            $"Radius = {Radius}, Length = {Length, 6:f2}");
}
```



ЭКЗЕМПЛЯР КЛАССА CIRCLE ПРИ КОМПОЗИЦИИ

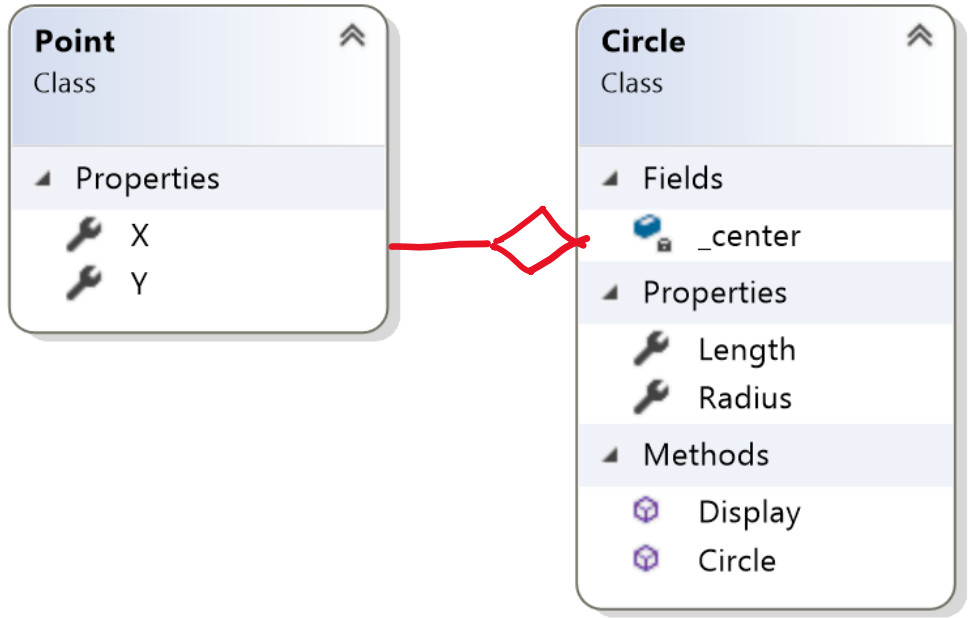
```
Circle circle = new();
circle.Center.X = 10;
circle.Center.Y = 20;
circle.Radius = 3.0;
circle.Display();
```

Вывод:
Center: X = 10, Y = 20; Radius = 3, Length = 18.85

АГРЕГАЦИЯ КЛАССОВ

Теперь точка передаётся в конструктор, её жизненный цикл **не контролируется** объектом Circle.

```
public class Circle
{
    public double Radius { get; set; }
    public double Length => 2 * Radius * Math.PI;

    private Point _center;
    public Circle(Point point, double radius)
        => (_center, Radius) = (point, radius);

    public void Display()
        => Console.WriteLine($"Center: X = {_center.X}, Y = {_center.Y}; " +
                             $"Radius = {Radius}, Length = {Length,6:f2}");

}
```



ЭКЗЕМПЛЯР КЛАССА CIRCLE ПРИ АГРЕГАЦИИ

```
Point pt = new Point();
pt.X = 10;
pt.Y = 20;
Circle circle = new Circle(pt, 10);
circle.Display();
```

Вывод:
Center: X = 10, Y = 20; Radius = 10, Length = 62.83

АГРЕГАЦИЯ VS. КОМПОЗИЦИЯ

Уровень отличия	**Агрегация**	**Композиция**
Общие	Отличается от композиции тем, что не является владельцем своих частей	Способ «обертки» отдельных типов или объектов, как одного общего
Отношения	Отношение HAS-A между агрегатом и его частями	Объект-композит всегда включает объекты-части
UML	Обозначается пустым ромбом у типа-композита	Обозначается закрашенным ромбом у типа-композита
Жизненный цикл	Объекты-части агрегата имеют собственное времени жизни	Объекты-части композита собственного времени жизни не имеют
Ассоциация	Агрегация или «слабая ассоциация»	«Сильная агрегация» или «Сильная ассоциация

АССОЦИАЦИЯ

- **Ассоциация** [*association*] – взаимное включение классов, то есть двунаправленная связь между ними
  - **Множественность/кратность ассоциации** – число объектов, участвующих в ассоциации с каждой стороны
    - **1:1** (один к одному) «зачётная книжка – студент»
    - **1:n** (один ко многим) «учебная группа – студенты»
    - **M:N** (многие ко многим) «спортивные секции – студенты»

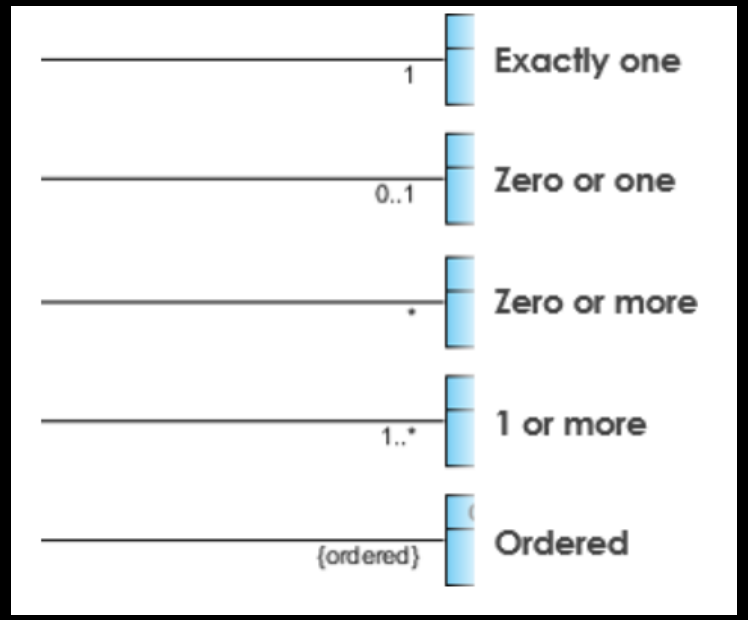
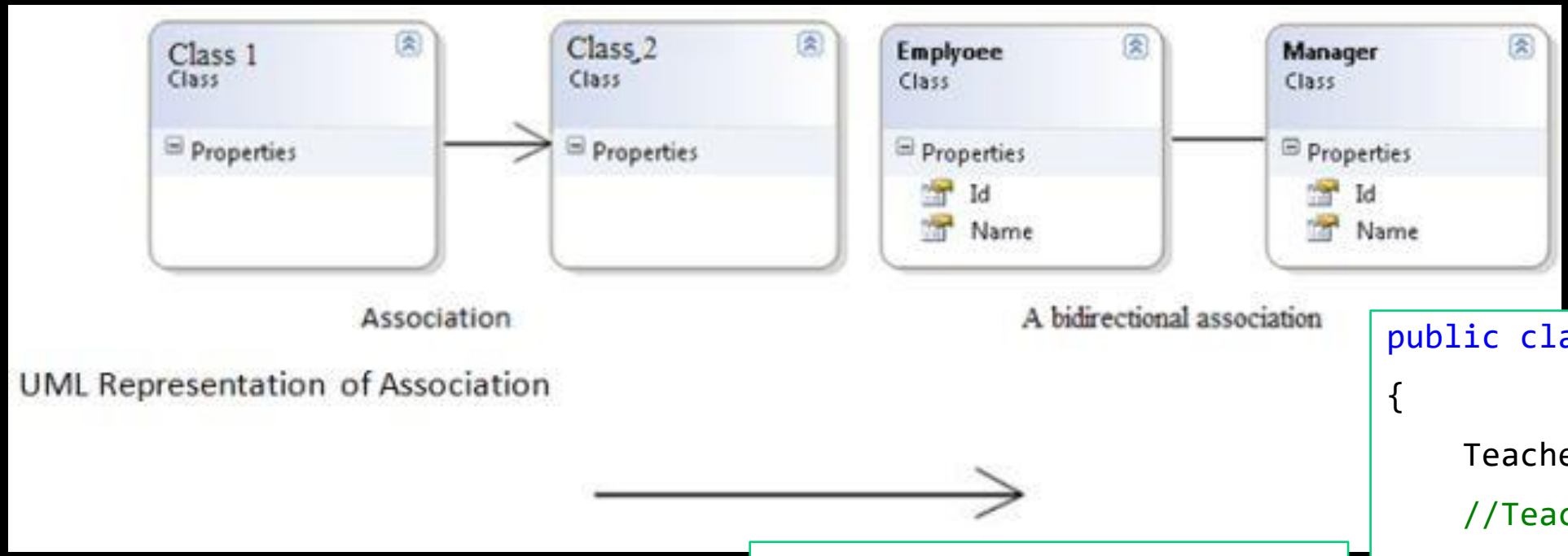


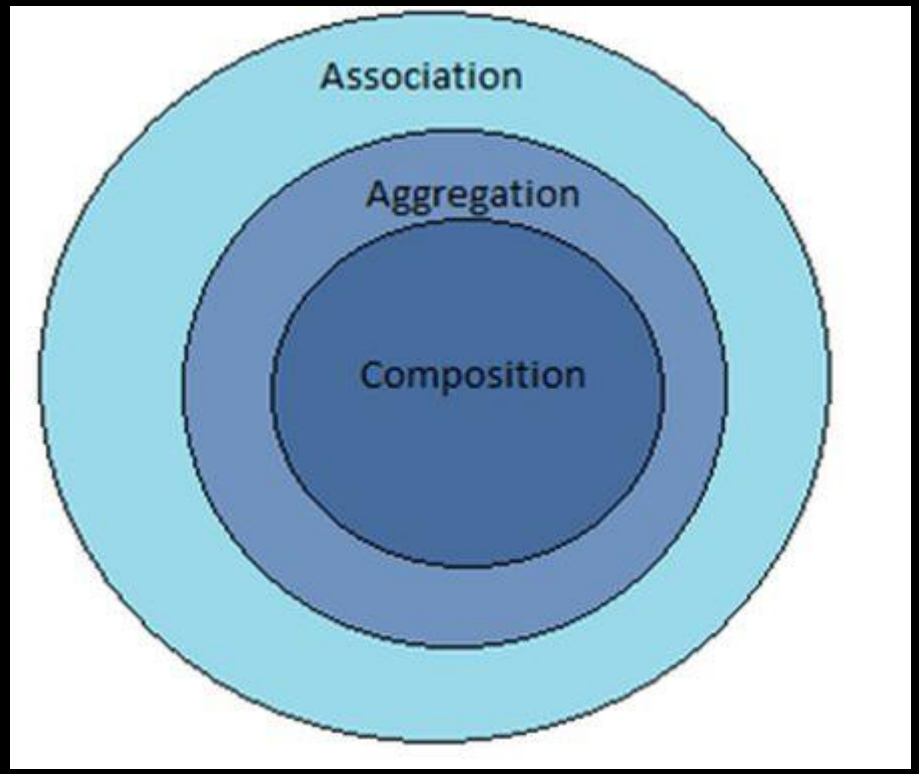
ДИАГРАММА АССОЦИАЦИИ

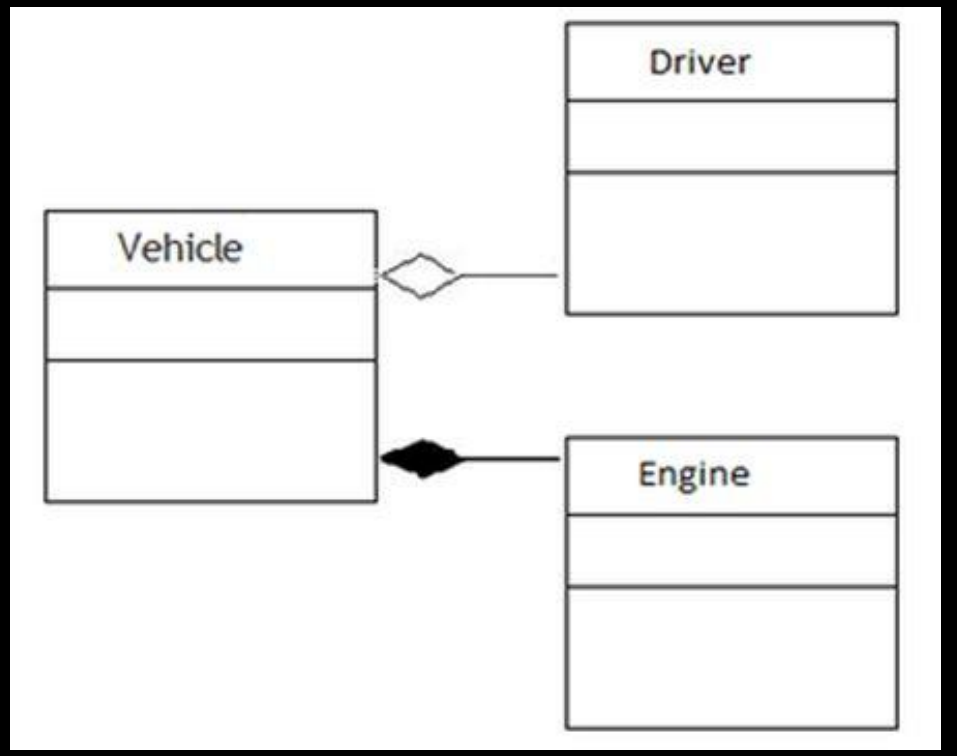



```
public class Teacher
{
    Assist _profAssist;
   // Assist[] _assists;
}
```

```
public class Assist
{
    Teacher _employeeTeacher;
    //Teacher[] _emplTeachers;
}
```

АССОЦИАЦИЯ, АГРЕГАЦИЯ, КОМПОЗИЦИЯ

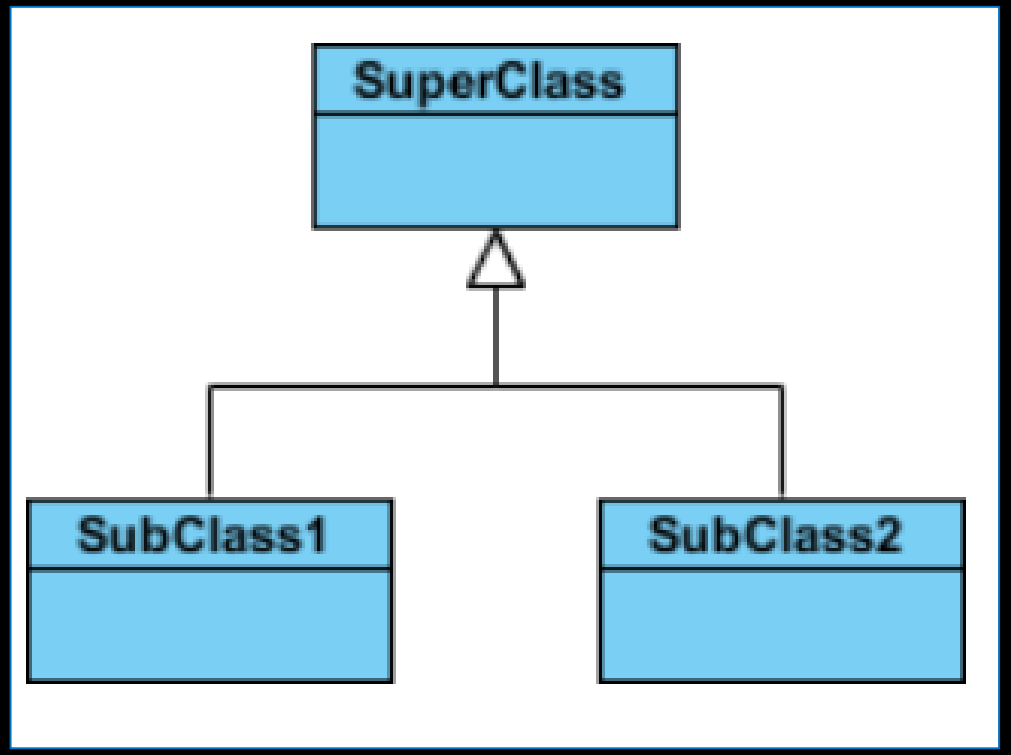




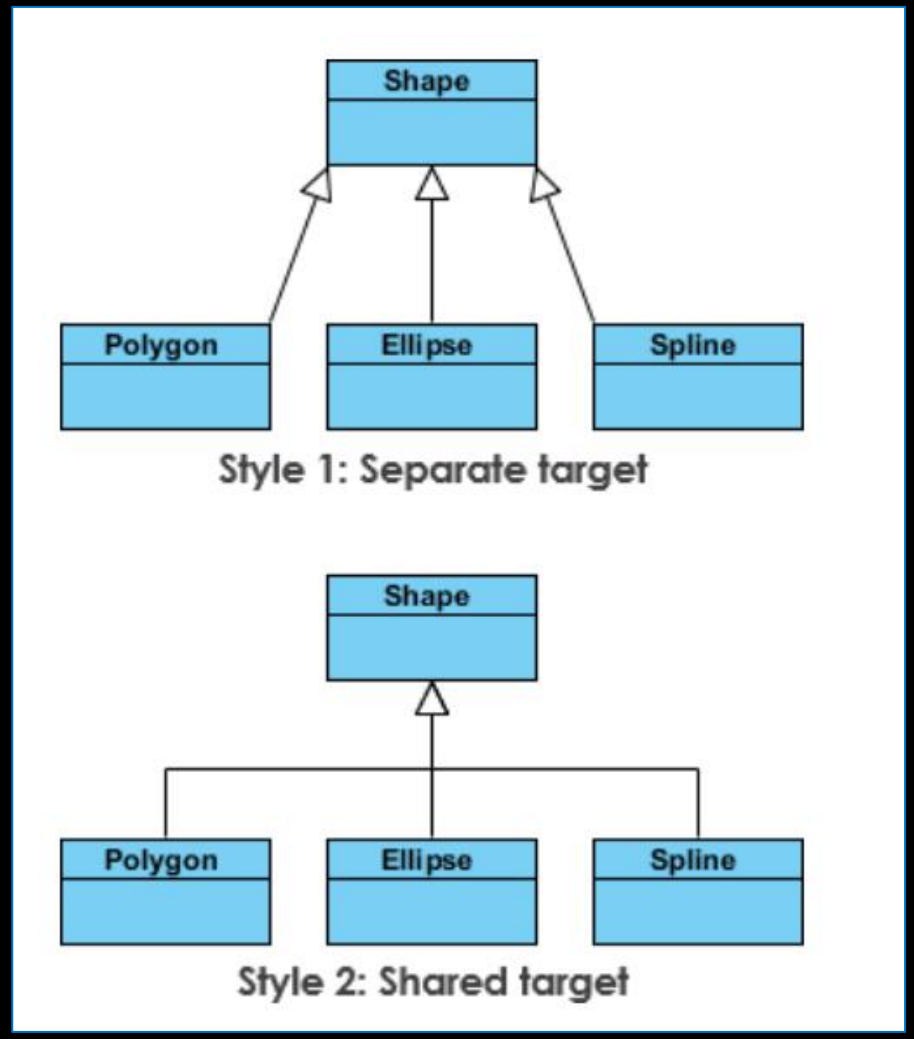
НАСЛЕДОВАНИЕ

- **Наследование** [inheritance] – отражение отношения «являться экземпляром» или «частное/общее» («Это есть») (IS A)

Наследование отражает иерархию классов как иерархию **обобщения** или как иерархию **специализации**



НАСЛЕДОВАНИЕ



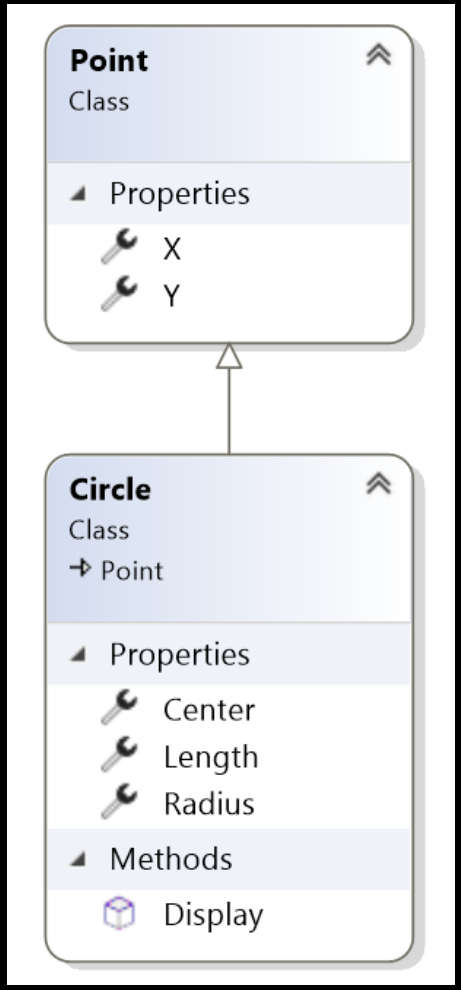
- **Обеспечивает** не только наследование интерфейса, как обязательство предоставить «не худший интерфейс», но и наследования реализации
- **Вводит** правила совместимости типов производного класса и базового класса
- **Позволяет** единообразно сформулировать понятие полиморфизма за счёт переопределения виртуальных методов
- **Открывает** производному классу секреты реализации базового класса, то есть **нарушает инкапсуляцию**

НАСЛЕДОВАНИЕ ОТ КЛАССА POINT

```
public class Circle : Point
{
    public double Radius { get; set; }
    public double Length => 2 * Radius * Math.PI;
    public Point Center
    {
        get => new Point() { X = this.X, Y = this.Y };
        set { X = value.X; Y = value.Y; }
    }
    public void Display()
        => Console.WriteLine($"Center: X = {X}, Y = {Y}; " +
                             $"Radius = {Radius}, Length = {Length,6:f2}");
}
```

ЭКЗЕМПЛЯР КЛАССА CIRCLE – НАСЛЕДНИКА POINT

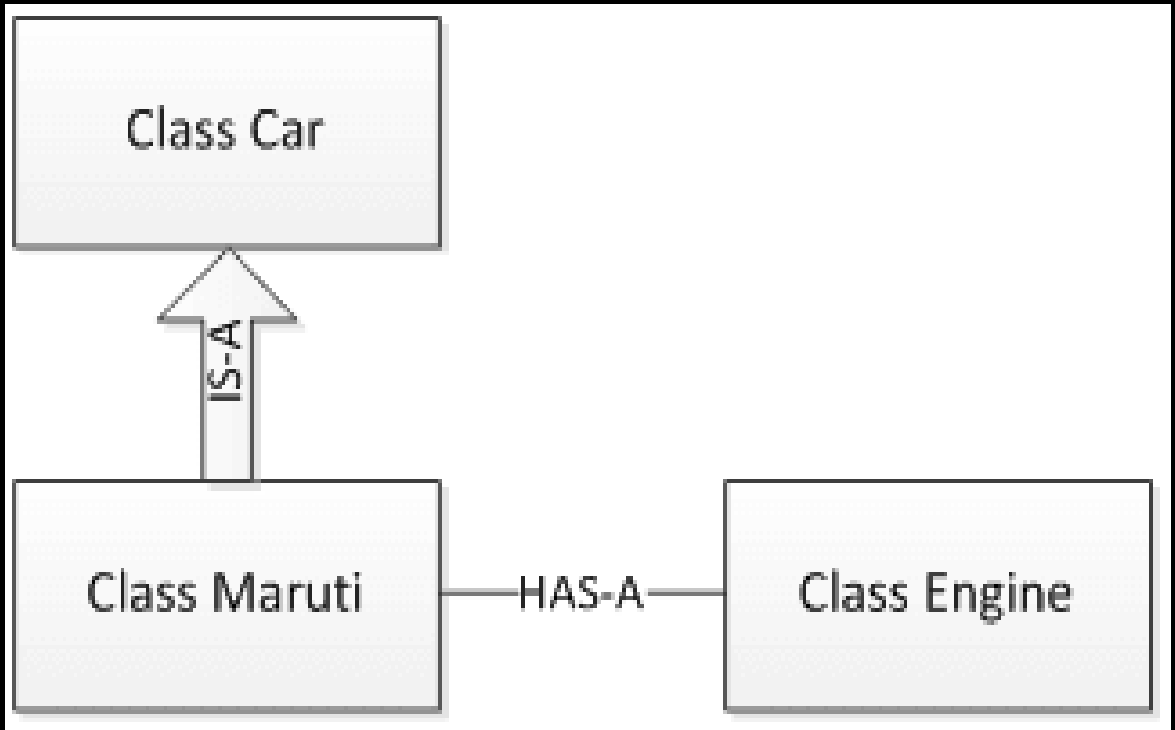
```
Circle circle = new Circle();
circle.X = 24;          // Свойство из Point.
circle.Y = 10;          // Свойство из Point.
circle.Radius = 2;      // Свойство из Circle.
circle.Display();
circle = new Circle();
circle.Display();
```



Вывод:

Center: X = 24, Y = 10; Radius = 2, Length = 12.57
Center: X = 0, Y = 0; Radius = 0, Length = 0.00

НАСЛЕДОВАНИЕ VS АГРЕГАЦИЯ \ КОМПОЗИЦИЯ



- Отношение IS-A основано на наследовании, которое может быть реализовано наследованием класса или наследованием интерфейса
- Отношение HAS-A основано на композиции и повышает продуктивность повторного использования кода

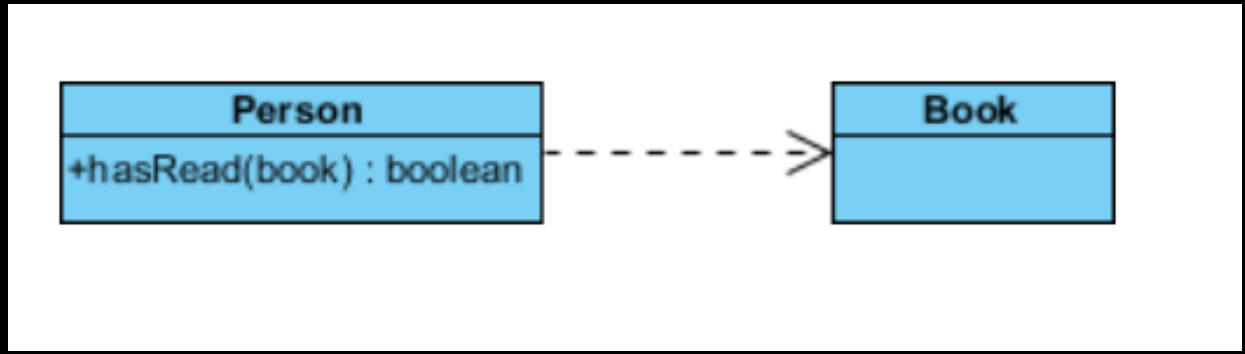
НАСЛЕДОВАНИЕ VS АГРЕГАЦИЯ \ КОМПОЗИЦИЯ

- Модифицировать типы, входящие в агрегации проще, чем связанные наследованием
  - Изменение базового типа влияет на все классы-наследники
- Composition is dynamic binding (run-time binding) while Inheritance is static binding (compile time binding)
- Наследование упрощает добавление классов-наследников по сравнению с классами-агрегатами за счёт полиморфизма
  - Если код использует интерфейс базового типа, то он без модификаций сможет работать и любым новым наследником
  - Агрегации являются мощным средством проектирования при использовании совместно с интерфейсами (о них будем говорить позже)
- Модификация интерфейса класса-агрегата всегда проще, чем типа, входящего в иерархию наследования
- При любом отношении модифицировать внутреннюю структуру типа проще, чем его интерфейс

Не стоит использовать наследование только для организации повторного использования кода или острого желания пользоваться полиморфизмом. Если отношение IS A отсутствует или надумано лучше использовать агрегацию (композицию) с купе с интерфейсами

ЗАВИСИМОСТЬ

- **Зависимость** [*dependency*] – наиболее общее отношение между классами, указывающее лишь на то, что один класс зависим от особенностей реализации другого класса

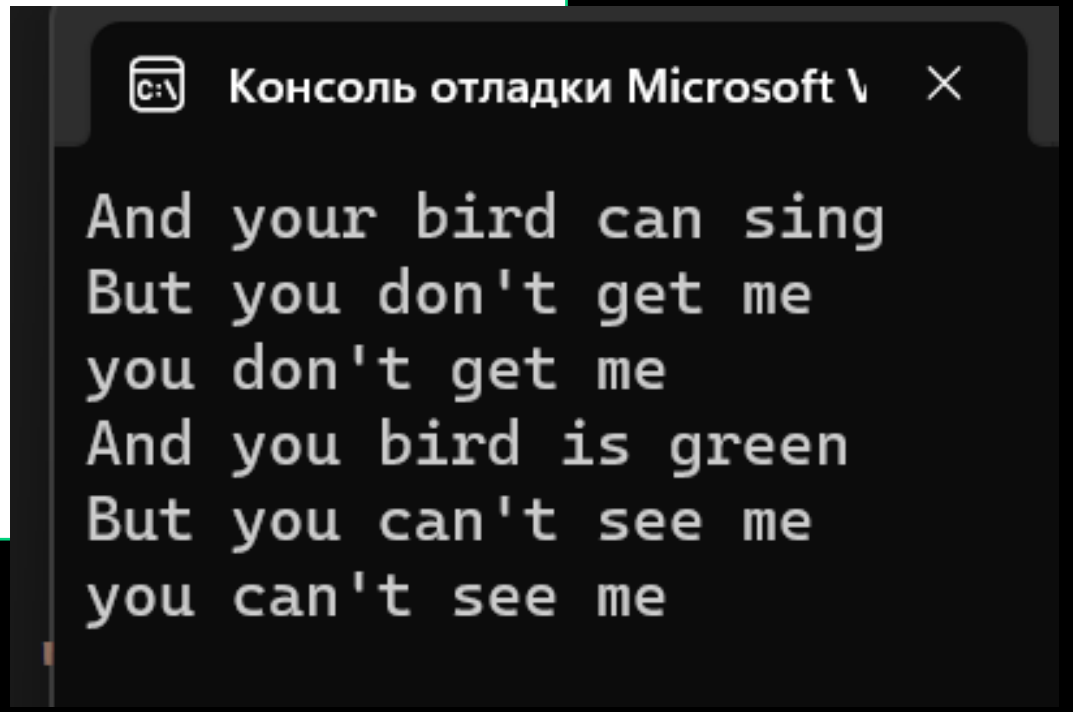


Использование [*use*] – один класс при реализации своих методов обращается к интерфейсу другого класса

```
class Words
{
    string[] _words;
    public Words(params string[] words ) => _words = words;
    public List<string> GetValidStrings (LenghtValidator f)
    {
        List<string> result = new List<string>();
        for (int i = 0; i < _words.Length; i++)
        {
            if (f.IsValid(_words[i]))
            {
                result.Add(_words[i]);
            }
        }
        return result;
    }
}
```

```
class LenghtValidator
{
    int _length;
    public LenghtValidator(int length = 0)
    {
        if ( length < 0 )
            throw new ArgumentOutOfRangeException("length");
        _length = length;
    }
    public bool IsValid(string value)
    {
        if( value == null )
            throw new ArgumentNullException("value");
        if (value.Length > _length) return false;
        return true;
    }
}
```

```
string[] test = { "You tell me that you've got everything you want",
"And your bird can sing",
"But you don't get me",
"you don't get me",
"You say you've seen the seven wonders",
"And you bird is green",
"But you can't see me",
"you can't see me"};
Words words = new Words(test);
LenghtValidator lv = new LenghtValidator(25);

foreach(string s in words.GetValidStrings(lv))
{
    Console.WriteLine(s);
}
```



ИНСТАНЦИРОВАНИЕ

- **Инстанцирование** [*instancing*], данное отношение кроме связывания класса и его экземпляров может связывать и один класс с другим классом при наличии в языке обобщённых [*generic*] классов

ВКЛЮЧЕНИЕ / ВЛОЖЕНИЕ

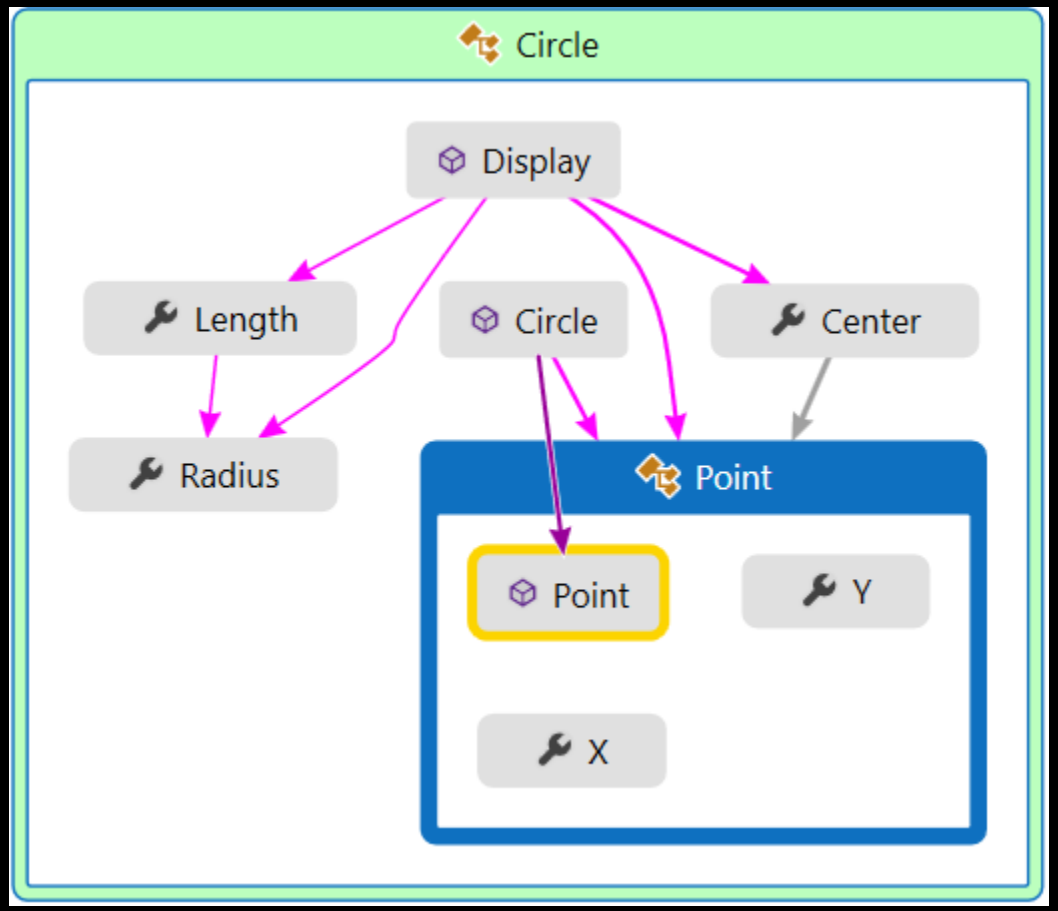
```
public class Circle
{
    public Point Center { get; set; } = new Point(); // центр окружности

    public double Radius { get; set; }
    public double Length => 2 * Radius * Math.PI;

    public void Display()
        => Console.WriteLine($"Center: X = {Center.X}, Y = {Center.Y}; " +
                             $"Radius = {Radius}, Length = {Length,6:f2}");
    public class Point
    {
        public double X { get; set; }
        public double Y { get; set; }
    }
}
```

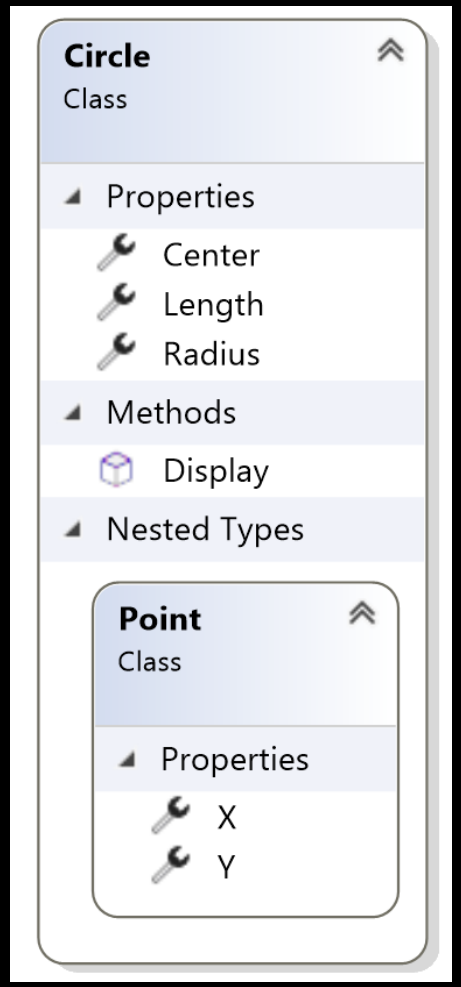
ЭКЗЕМПЛЯР КЛАССА CIRCLE ПРИ ВЛОЖЕНИИ КЛАССА POINT

```
Circle circle = new Circle();
circle.Center.X = 100;
circle.Center.Y = 200;
circle.Radius = 30.0;
circle.Display();
```

Вывод:
Center: X = 100, Y = 200; Radius = 30, Length = 188.50

ДИАГРАММА ВЛОЖЕНИЯ КЛАССОВ





НАСЛЕДОВАНИЕ ОТ КЛАССА С PROTECTED-ЧЛЕНАМИ

```
// Базовый класс круга.
public class Disk
{
    protected double radius;
    protected Disk(double radius) => this.radius = radius;
    protected double Area => radius * radius * Math.PI;
}
```

```
// Класс кольцо.
public class Ring : Disk
{
    // Радиус внутренней окружности:
    new double radius;
    public Ring(double outerRadius, double innerRadius)
                : base(outerRadius) { radius = innerRadius; }
    public new double Area => base.Area - Math.PI * radius * radius;
    public void Print()
        => Console.WriteLine($"Ring: Outer radius = {base.radius:f2}, " +
                             $"Inner radius = {radius:f2}, Area = {Area:f3}");
}
```

ЭКЗЕМПЛЯР КЛАССА RING – НАСЛЕДНИКА DISK

```
Ring ring = new Ring(10.0, 4.0);
ring.Print();
```

Вывод:
Ring: Outer radius = 10.00, Inner radius = 4.00, Area = 263.894

ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

- Буч, Гради, Роберт А., Энгл, Майкл У., Янг, Бобби Дж., Коналлен, Джим, Хьюстон, Келли А. Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е изд. – М.: ООО «И.Д. Вильямс», 2008.
- Init (справочник по C#) (https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/init)
- How to: Add class diagrams to projects (https://learn.microsoft.com/en-us/visualstudio/ide/class-designer/how-to-add-class-diagrams-to-projects?view=vs-2022)
- Understanding Association, Aggregation, Composition and Dependency relationship (https://www.dotnettricks.com/learn/oops/understanding-association-aggregation-composition-and-dependency-relationship)
- https://www.c-sharpcorner.com/UploadFile/ff2f08/association-aggregation-and-composition/
- https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/
- Использование диаграммы классов UML при проектировании и документировании программного обеспечения (https://habr.com/ru/articles/572234/)
- Полное руководство по диаграмме классов UML (https://www.cybermedian.com/ru/a-comprehensive-guide-to-uml-class-diagram/)