ООП–(**Объектно-ориентированное программирование)** — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

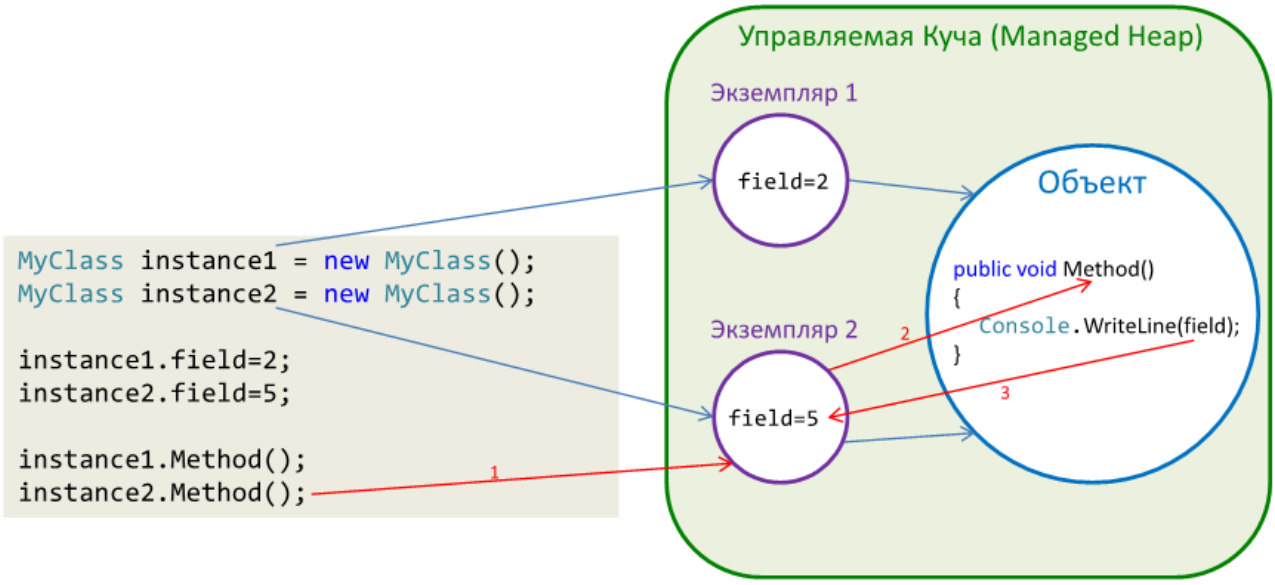
Класс—это конструкция языка, состоящая из ключевого слова `class`, идентификатора (имени) и тела.

Класс может содержать в своем теле : поля, методы, свойства и события.

Поля определяют состояние, а ***методы*** поведение будущего объекта.

Объекты содержат в себе статические поля и все методы.

Экземпляры содержат не статические поля.



Модификаторы доступа – `private` и `public` определяют видимость членов класса.

Никогда не следует делать поля открытыми, это плохой стиль. Для обращения к полю, рекомендуется использовать методы доступа.

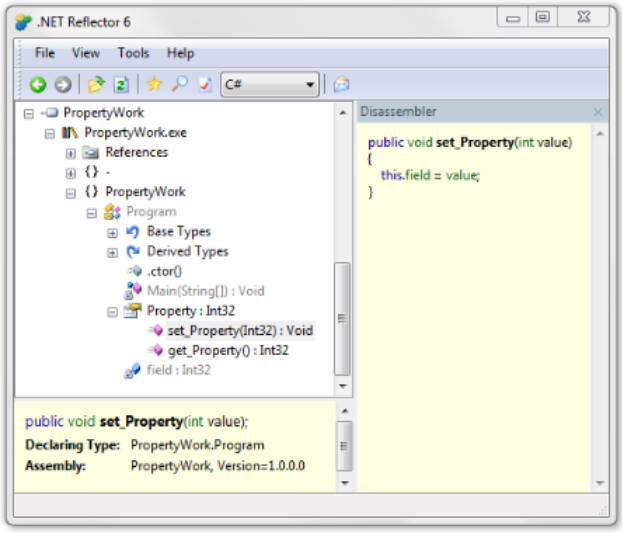
Свойство это конструкция языка C#, которая заменяет собой использование обычных методов доступа.

Работа со свойством экземпляра напоминает работу с полями экземпляра.

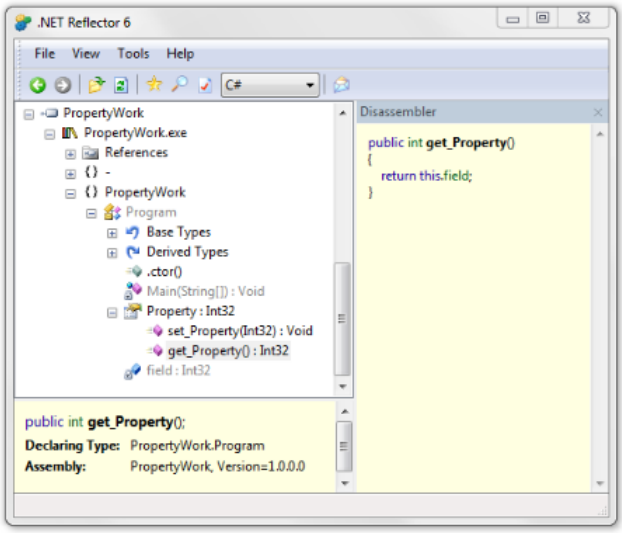
Свойство состоит из имени, типа и тела. В теле задаются методы доступа, через использование ключевых слов `set` и `get`.

Метод `set` автоматически срабатывает тогда, когда свойству пытаются присвоить значение. Это значение представлено ключевым словом `value`.

Метод `get` автоматически срабатывает тогда, когда мы пытаемся получить значение.



Метод доступа `set`



Метод доступа `get`

Метод доступа `get`–используется для получения значения из переменной.

Метод доступа `set`-используется для записи значения в переменную.

```
int field;

public int Property
{
   get
   {
      return field;
   }
}
```

Свойство только для чтения

```
int field;

public int Property
{
   set
   {
      field = value;
   }
}
```

Свойство только для записи

Конструктор класса—специальный метод, который вызывается во время построения класса.

Конструкторы по умолчанию

```
public MyClass()
{
}
```

Пользовательские конструкторы

```
public MyClass (int arg)
{
}
```

Если в теле класса не определен явно ни один конструктор, то всегда используется «невидимый» конструктор по умолчанию.
Имя конструктора всегда совпадает с именем класса. Конструкторы не имеют возвращаемых значений.

Задача конструктора по умолчанию – инициализация полей значениями по умолчанию.

Задача пользовательского конструктора – инициализация полей предопределенными пользователем значениями.

Если в классе имеется пользовательский конструктор, и при этом требуется создавать экземпляры класса с использованием конструктора по умолчанию, то конструктор по умолчанию должен быть определен в теле класса явно, иначе возникнет ошибка на уровне компиляции.

Конструкторы, вызывающие другие конструкторы

Один конструктор может вызывать другой конструктор того же класса, если после сигнатуры вызывающего конструктора поставить ключевое слово `this` и указать набор параметров, который должен совпадать по количеству и типу с набором параметров вызываемого конструктора.

Вызывающий конструктор

```
public Point(string name)
             : this(300, 400)
{
    this.name = name;
}
```

Вызываемый конструктор

```
public Point(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

Auto-Implemented Properties

Автоматически реализуемые свойства это более лаконичная форма свойств, их есть смысл использовать, когда в методах доступа `get` и `set` не требуется дополнительная логика.

При создании автоматически реализуемых свойств, компилятор создаст закрытое, анонимное резервное поле, которое будет доступно с помощью методов `get` и `set` свойства.

```
public class MyClass
{
    public string Name { get; set; }
    public string Book { get; set; }
}
```

Создание экземпляра класса по сильной ссылке

```
MyClass instance = new MyClass();
instance.Method();
```

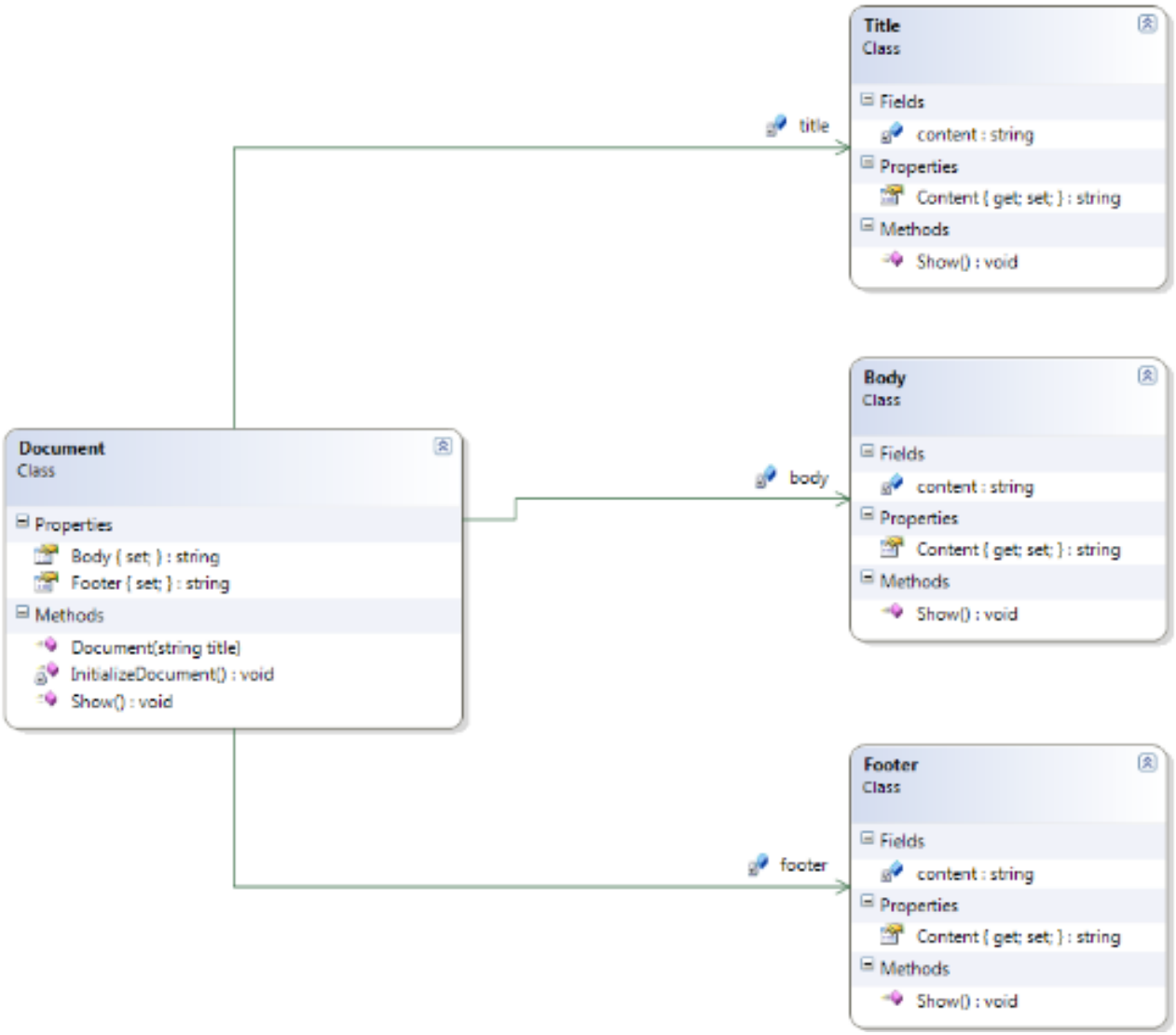
Создание экземпляра класса по слабой ссылке

```
new MyClass().Method();
```

Первая парадигма ООП

Инкапсуляция *(инкапсуляция вариаций)* –

Техника сокрытия частей Объектно-Ориентированных программных систем.



Partial classes

В C# реализована возможность разделить создание класса или метода (структуры, интерфейса) между двумя или более исходными файлами или модулями. Каждый исходный файл содержит определение типа или метода, и все части объединяются при компиляции приложения.

```
partial class PartialClass
{
    public void MethodFromPart1()
    {
    }
}
```

```
partial class PartialClass
{
    public void MethodFromPart2()
    {
    }
}
```

```
static void Main()
{
    PartialClass instance = new PartialClass();

    instance.MethodFromPart1();
    instance.MethodFromPart2();
}
```

Unified Modeling Language

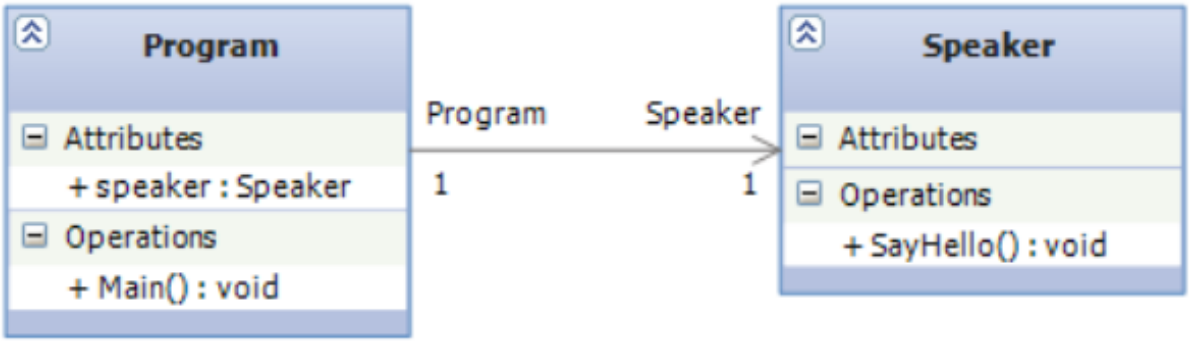
UML (*Unified Modeling Language* — унифицированный язык моделирования) — язык графического описания для объектного моделирования в области разработки программного обеспечения.

UML – был создан для того, что бы участники процесса создания программного обеспечения смогли строить модели для визуализации системы, определения её структуры и поведения, сборки системы и документирования решений, принимаемых в процессе разработки.

Class diagrams

Диаграммы классов используются для изображения классов, а также связей между ними. Самым важным является показ классов и связей между ними с различных сторон таким способом, что бы передать наиболее важный смысл.

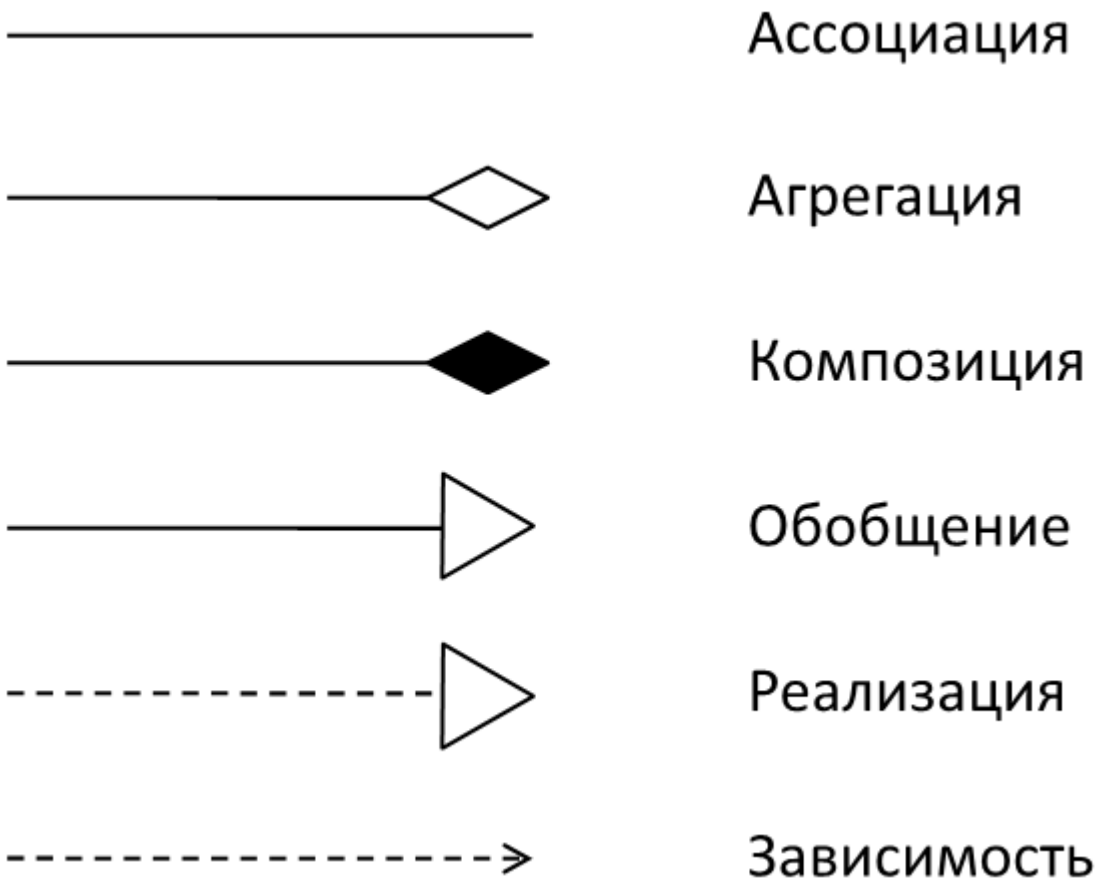
Диаграмма классов представляет собой статическую модель системы. Диаграмма классов не описывает поведение системы, или то, как взаимодействуют экземпляры классов.



MyClass
field : int
Method() : int
Обязанности: -Вернуть строку “Hello!” - …

```
class MyClass
{

    int field;


    int Method() { return 777; }


    // TODO: Вернуть строку "Hello!"

}
```

Связи отношений между классами



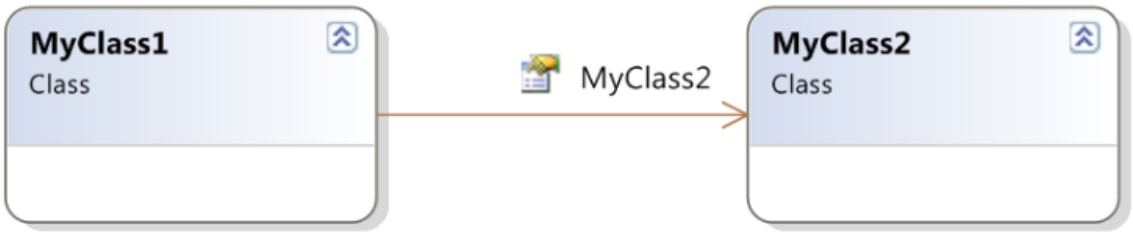
• **Ассоциация** показывает, что объекты одной сущности (класса) связаны с объектами другой сущности.

• **Агрегация** — это разновидность ассоциации при отношении между целым и его частями.(Отношение типа: «Я знаю о… и без этого могу существовать»). Одно отношение агрегации не может включать более двух классов (контейнер и содержимое).

• **Композиция** — более строгий вариант агрегации. Композиция имеет жёсткую зависимость времени существования экземпляров класса-контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено. (Отношение типа: «Я знаю о… и без этого не могу существовать»)

• **Зависимость** — это слабая форма отношения использования, при котором изменение в спецификации одного влечёт за собой изменение другого, причем обратное не обязательно.

Направленная (Однонаправленная)




```
class MyClass1
{

     public MyClass2 MyObj;

}
```

```
class MyClass2
{



}
```

Двунаправленная (Ненаправленная)


```
class MyClass1
{

     public MyClass2 myObj;

}
```

```
class MyClass2
{

     public MyClass1 myObj;

}
```

Рефлексивная Ассоциация

```
class MyClass
{

      public MyClass myObj;

}
```