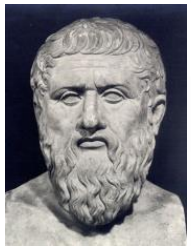
Информатика

Объектно-ориентированное программирование. Часть 2

© Марченко Антон Александрович 2017 г.
Абрамский Михаил Михайлович

ООП продолжается...



Заказали два приложения

- #1 Система управления договорами
- #2 Текстовая игра

Система управления договорами

Договор заключается с юр. лицами и физ. лицами
У договора есть предмет, сумма, сроки
Сроки и суммы можно изменять
У договора есть статус и ответственный сотрудник
У заказчиков физ. лиц: ФИО, паспортные данные, прописка
У юр. лиц – наименование, адрес, реквизиты, директор
Договора сохраняются в хранилище с возможностью поиска по заказчикам

Текстовая игра двух игроков

Игроки наносят друг другу удары по очереди
Игроки указывают силу удара от 1 до 9
С увеличением силы возрастает вероятность промаха
При успешном ударе у противника уменьшаются очки здоровья (hp)
Когда hp одного из игроков становится <=0, игрок проигрывает

Ранее по ООП…

Классы, объекты, поля, методы, опциональные параметры, перегрузка, абстракция, инкапсуляция, модификаторы, доступ к членам, this, свойства, автоматические свойства, конструкторы, статические поля и методы, статический конструктор

Разработанный ранее код #1

```
public class IndividualContract
{
    public string Subject { get; }
    public DateTime DueTo { get; set; }
    public double Cost { get; set; }
    public Individual Individual { get; }
    public Employee Responsible { get; set; }
}

public class Employee
{
    public string Name { get; }
    public Department Department { get; set; }
    public Employee Chief { get; set; }
}
```

Код поправим

```
public class Individual
{
    public string Name { get; }
    public PassportInfo PassportInfo
    { get; set; }
    public Address Address
    { get; set; }
}
```

Разработанный ранее код #2

```
class Player
{
    public int Hp { get; private set; }
    public string Name { get; }
    public string BattleCry { get; set; }

    public Player(string name, string battleCry = "This is Sparta!!!11")
    {
        Hp = 100;
        Name = name;
        BattleCry = battleCry;
    }
    public void ShoutBattleCry()
        => Console.WriteLine(Name + ": " + BattleCry);
}
```

Заказчик изменил требования

- Ребят, мне нужно хранить информацию о том, когда какой договор с физ. лицом был заведён. Ну как журнал такой

… через 5 минут…

- Да, и ещё мне надо, чтобы результаты игр между игроками сохранялись. Имена игроков, дата и кто победил. В принципе, тоже журнал

Действия менеджера проекта

- *Разработчик Сёма, разрабатываешь журнал договоров!*
- *Разработчик Соня, разрабатываешь журнал результатов игр!*
- *Обоим три дня на разработку, потом ваш код присоединяем к проекту*

Размышления Семёна

- Журнал договоров – **набор записей**, в каждом хранится ссылка на договор с физ.лицом и дата заключения
  - **Запись в журнале** – сущность с двумя полями (договор, дата). Поля можно инкапсулировать в свойства
  - **Журнал** – набор записей (массив, список)
    - Инкапсулируем его в отдельный класс
    - Нужно использовать коллекции (кое-что по ним уже знаем, остальное будет в следующем семестре)

Журнал договоров от Семёна

```
public class JournalEntry
{
    public IndividualContract IndividualContract { get; }
    public DateTime Date { get; }
    …
}
public class Journal
{
    public List<JournalEntry> JournalEntries { get; }
        = new List<JournalEntry>();
    public int JournalSize { get; private set; }
    …
}
```

Размышления Софьи

- **Журнал результатов игр** – **набор записей**, в каждой хранятся имена обоих игроков, дата игры, номер победителя (1 или 2)
  - **Запись в журнале** – сущность с четырьмя полями (имена игроков, дата игры, результат). Поля можно инкапсулировать в свойства
  - **Журнал** – набор записей (массив, список)

Журнал договоров от Софьи

```
public enum Winner { First,Second}
public class JournalEntry
{
    public string Player1 { get; }
    public string Player2 { get; }
    public DateTime Date { get; }
    public Winner Winner { get; }
    …
}
public class Journal
{
    public const int JournalCapacity = 3000;

    public JournalEntry[] JournalEntries
        = new JournalEntry[JournalCapacity];
    public int JournalSize { get; private set; }
    …
}
```

Через три дня

- PM: «Сёма, Соня, давайте код!»
- Разработчики дают код, и тут…
  - **У обоих разработчиков по два разных класса с одинаковыми названиями!**
    - Классы Journal почти совпадают
    - Но JournalEntry – совершенно разные

В каждой шутке доля…

- Разумеется, у нас 2 разных проекта
  - И конфликта имён нет
- Разумеется, если даже бы они были в одном проекте, можно было бы переименовать классы
  - GameJournalEntry, GameJournal
  - ContractsJournalEntry, ContractsJournal

Но…

- В масштабных проектах – несколько тысяч классов, которые писали разработчики разных групп и компаний со всего мира
- Разработчиков много, а названий сущностей мало
  - List, Connection, Entry, Reader, Writer
    - ❖Сущность называют так, как она себя ведёт. Но некоторые сущности ведут себя похожим образом

Пространства имён `namespace`

Чтобы не перепутать внешне одинаковые сущности, используют пространства имён

- Пространство имён – *логическое* объединение связанных объектов
- Может объединять пользовательские типы и вложенные пространства имён
- У пространства имён есть уникальное имя
- Полное имя класса: NamespaceId.ClassName

Директивы using

- Можно в коде всегда писать полные имена классов

```
System.IO.StreamReader sr = new System.IO.StreamReader("input.txt");
```

- Лень постоянно писать полные имена, поэтому используем директивы `using`

```
using System.IO;

…
StreamReader sr = new StreamReader("input.txt");
```

Глобальное пространство имён global::

Существует глобальное пространство имён (без указанного явно имени)

При переопределении типов можно обращаться к глобальному пространству имён во избежание конфликтов

```
class TestApp
{
    public class System { }
    const int Console = 7;
    const int number = 66;
    static void Main()
    {
        // Следующая строчка приведёт к ошибке.
        Console.WriteLine(number);

        // OK
        global::System.Console.WriteLine(number);
    }
}
```

Псевдонимы, static using

1. Можно обращаться к статическим членам класса без указания его имени (C# 6.0)

```
using static System.Math;
…
Console.WriteLine(Cos(0));
```

2. Можно указывать псевдонимы для классов

```
using printer = System.Console;
…
printer.WriteLine("Hello!");
```

Представим, что игра дописана

```
namespace Work.Projects.TextGame
{
    public class Game
    {
        public void Go()
        {
            Player p = new Player("Denis Popov");
            //..
        }
        static void Main()
        {
            new Game().Go();
            //Анонимный объект. Используется 1 раз.
        }
    }
}
```

Пространства имён

- В лекциях не будут указываться пространства имён в целях экономии места на слайдах и концентрации на важных деталях
- В проектах следует серьёзно подходить к организации кода, комментированию, документированию и пространствам имён!

Опять пришёл заказчик...

Заказчик: «*Чего так долго тянули с журналом! Все сроки сорвали! Быстрее дописывайте!...*

... а, кстати, увидел недавно в одной игре какой-то – там игроки не только ударяли, но и могли себя исцелять. Добавьте таких игроков, но обычных тоже оставьте.»

Анализ требований заказчика

- **Есть обычный игрок**
  - свойства: Hp, Name, BattleCry,
  - методы: Punch, BattleCry
- **Есть продвинутый игрок**
  - свойства: Hp, Name, BattleCry,
  - методы: Punch, BattleCry
  - новое свойство **HealPoints** – на сколько очков он может себя исцелять. При каждом исцелении уменьшается
  - новый метод **Heal(p)** – исцеление – увеличение Hp на p очков

Player

```
public class Player
{
    public int Hp { get; private set; }
    public string Name { get; }
    public string BattleCry { get; }
    public Player(string name, string battleCry)
    {
        Hp = 100;
        Name = name;
        BattleCry = battleCry;
    }
    public void shoutBattleCry(){…}
    public void Punch(Player p){…}
}
```

SmartPlayer

```
public class SmartPlayer
{
    public int Hp { get; private set; }
    public int HealPoints { get; private set; }
    public string Name { get; }
    public string BattleCry { get; }
    public SmartPlayer(string name, string battleCry)
    {
        Hp = 100;
        HealPoints = 20;
        Name = name;
        BattleCry = battleCry;
    }
    public void shoutBattleCry(){…}
    public void Punch(Player p){…}
    public void Heal(int p)
    {
        if (p <= HealPoints)
        {
            Hp += p;
            HealPoints -= p;
        }
    }
}
```

Вместе в одном проекте

```
public class Player
{
    public int Hp { get; private set; }
    public string Name { get; }
    public string BattleCry { get; }
    public Player(string name,
        string battleCry="This is Sparta!!!11")
    {
        Hp = 100;
        Name = name;
        BattleCry = battleCry;
    }
    public void shoutBattleCry(){…}
    public void Punch(Player p){…}
}
```

```
public class SmartPlayer
{
    public int Hp { get; private set; }
    public int HealPoints { get; private set; }
    public string Name { get; }
    public string BattleCry { get; }
    public SmartPlayer(string name, string battleCry)
    {
        Hp = 100;
        HealPoints = 20;
        Name = name;
        BattleCry = battleCry;
    }
    public void shoutBattleCry(){…}
    public void Punch(Player p){…}
    public void Heal(int p)
    {
        if (p <= HealPoints)
        {
            Hp += p;
            HealPoints -= p;
        }
    }}
```

SmartPlayer и Player

- Жуткое дублирование кода
- Любое изменение Player влечёт изменение SmartPlayer
- SmartPlayer может вести себя как Player, но не наоборот
- В SmartPlayer есть весь функционал Player, но обратное не верно

Принцип ООП #2: Наследование

- **Классы могут использовать готовую реализацию других классов, добавляя лишь то, чего не хватает в исходном (базовом, родительском классе)**
  - Концепция «повторного использования компонентов»
- По-английски – Inheritance
- Есть понятия «родительских» и «дочерних» классов («наследников»)
- Следует понимать как «расширение» или «уточнение»

Пример наследования

Родительский класс (предок, базовый класс, суперкласс) – **Человек**

Дочерний класс (потомок, производный класс, подкласс) – **Студент**

- студент является Человеком
  - может всё то, что может человек
  - содержит всю информацию, присущую человеку
- Человек не обязательно является Студентом
  - у студента есть данные (зачётка, студенческий) и методы (сдать экзамен, посетить лекцию), которых нет у произвольного человека

Игроки

- В нашем примере SmartPlayer – потомок Player
  - Может всё то же, что и Player, добавляет в Player новую информацию (HealPoints) и новый метод (Heal), а также *уточняет* конструктор
  - Player при этом не меняется (почти)

Наследование

```
public class SmartPlayer: Player
```

Класс SmartPlayer наследуется от Player.
Класс SmartPlayer расширяет Player

в Java – ключевое слово extends

Внутри SmartPlayer описываем то, чего нет у Player

Данные

Описываем только те поля и свойства, которые новые для SmartPlayer, остальное есть в Player

с ними будет небольшое веселье, но позже

```
public class SmartPlayer: Player
{
    public int HealPoints { get; private set; }
```

Добавим метод Heal

```
public class SmartPlayer: Player
{
    public int HealPoints { get; private set; }

    public void Heal(int p)
    {
        if (p <= HealPoints)
        {
            Hp += p;
            HealPoints -= p;
        }
    }
}
```

Не скомпилируется с ошибкой: Hp … метод set недоступен

Что?! Почему?! Я же в этом же классе работаю?

Или…?

Модификатор `private`

`private` разрешает прямой доступ только в базовом классе!

На наследниках это не работает!
Унаследованные private поля и методы недоступны и производных классов.

Что делать?

2 способа

1. Использовать public set и get – методы у свойства Hp в классе Player
   - Тогда игроки смогут увеличивать Hp в любое время. Не порядок!
2. Изменить модификатор set метода у Hp.
   - Чтобы потомки могли иметь доступ
   - Но только потомки! Извне – нет!

Модификатор доступа protected

Разрешает прямое обращение к членам класса из базового класса и всех его потомков

Свободнее private, жёстче чем public.

```
public class Player
{
    public int Hp { get; protected set; }
```

Теперь у SmartPlayer можно вызывать Hp-=p

Теперь вы знаете модификаторы private, public, protected

Конструктор SmartPlayer

```
public SmartPlayer(string name, string battleCry)
{
    Hp = 100;
    Name = name;
    BattleCry = battleCry;
    HealPoints = 20;
}
```

- **Решили вопрос с доступом, но:**
  - Конструктор почти полностью дублирует конструктор Player
  - Код не скомпилируется. «У класса Player нет конструктора без параметров»

Экземпляры классов-наследников

Представим, что создали экземпляр SmartPlayer:

```
SmartPlayer sp = new SmartPlayer(…);
```

- При создании экземпляра дочернего класса сначала неявно создаётся экземпляр родительского класса, а потом выполняется то, что связано со SmartPlayer
- Если создаётся (пусть и неявно) экземпляр базового класса, значит вызывается его конструктор

Так вот

```
public SmartPlayer(string name, string battleCry)
{
    Hp = 100;
    Name = name;
    BattleCry = battleCry;
    HealPoints = 20;
}
```

В дочернем конструкторе первым делом вызывается родительский

Если явно не указано, происходит попытка вызова конструктора без параметров – а его у Player нет, поэтому ошибка

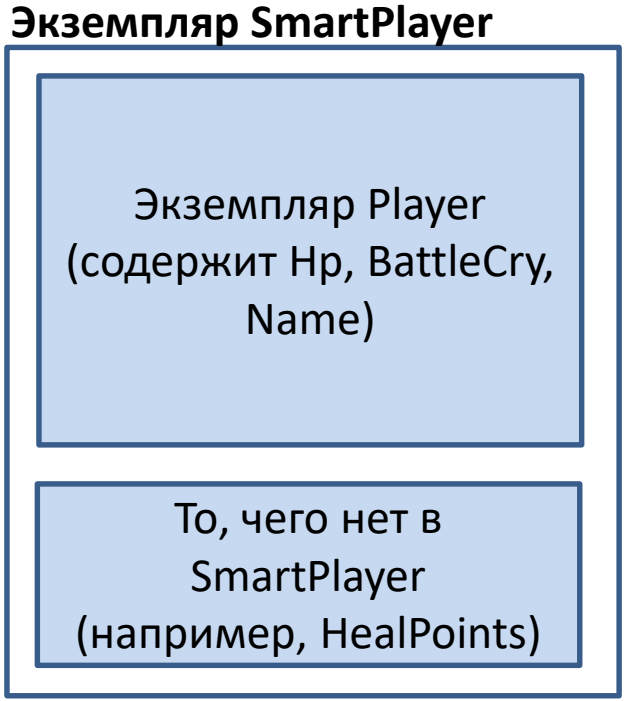
Явный вызов конструктора базового класса

`base`– обращение к экземпляру базового класса (ссылка, как и `this`)

Через `base` - не только обращение к экземпляру, но и вызов конструктора

```
public SmartPlayer(string name, string battleCry)
    :base(name,battleCry)
{
    HealPoints = 20;
}
```

Экземпляры классов-наследников в памяти



Объект подкласса может «прикинуться» объектом базового класса

В объекте SmartPlayer есть не только члены Player,
но и спрятанная сущность Player

base – ссылка на эту скрытую сущность

«Самый базовый» класс

- На самом деле Player – тоже наследник
- И любой класс в C# - наследник object (псевдоним/сокращение System.Object)
- object - корень иерархии классов. В нём даже есть свои собственные методы
  - Player можно привести к строке с помощью ToString, унаследованного от object

Изменение родительских методов в потомках

(На этом слайде намеренно не говорим про переопределение. Это не оно!)

Что хотим увидеть при выводе игрока на экран? Пусть будет имя и тип!

В Player добавим метод:

```
public string ToString()
{
    return "Player: "+Name;
}
```

```
Player p = new Player("Leonidas");
Console.WriteLine(p.ToString());
```

Это не перегрузка: совпадают и название и параметры

Соглашение об обозначениях

Есть объект (экземпляр класса)

- Все его public-методы назовём ~~публичным~~ интерфейсом объекта (класса)
- То, как именно работает метод, назовём конкретной реализацией/поведением объекта(класса)

Отцы и дети

- Интерфейс любого потомка включает в себя интерфейс родителя:
  - Так как любой экземпляр производного класса может быть экземпляром базового класса
- Но при одинаковых методах интерфейса у них может быть разная реализация

Восходящее преобразование

```
Player p = new SmartPlayer("Leonidas","This is Sparta!!!11");
Console.WriteLine(p.ToString()); //Что выведется? Player:… или SmartPlayer:…?
```

- Интерфейс p определяется Player (левая часть/ссылка/интерфейс)
  - p не может вызвать heal (т.к. у Player его нет)
- Сужаем интерфейс потомка до интерфейса родителя

Смысл

- В предыдущем примере нет никакой необходимости создавать дочерние классы, ограничивая их интерфейс родительским
- Но во многих примерах такая необходимость есть!

Телефон и смартфон

Класс **Phone**, метод **Call**

Класс **SmartPhone** наследует **Phone** и добавляет метод **TakePhoto**

Вы дарите смартфон вашему пожилому родственнику (бабушке, дедушке). Они его используют только как телефон, т.е.

```
Phone p = new SmartPhone();
```

- p.Call() – работает, т.к. есть у Phone
- p.TakePhoto() – не работает

Много детей

У одного родителя может быть несколько наследников

- Есть договоры с физ.лицами и юр.лицами
- Оба наследники общего класса Contract

```
public class CompanyContract
{
    public string Subject {get;}
    public DateTime DueTo {get;set;}
    public double Cost {get; set;}
    public Company Company {get;}
}
```

```
public class IndividualContract
{
    public string Subject { get; }
    public DateTime DueTo { get; set; }
    public double Cost { get; set; }
    public Individual Individual { get; }
    public Employee Responsible { get; set;}
}
```

Выстраиваем иерархию

```
public class Contract
{
    public string Subject { get; }
    public DateTime DueTo { get; set; }
    public double Cost { get; set; }
}
public class IndividualContract
    :Contract
{
    public Individual Individual { get; }
    public Employee Responsible { get; set; }
}
public class CompanyContract
    :Contract
{
    public Company Company { get; }
}
```

new Требование()

Заказчик:

«Слушай, хранилище договоров нужно и для физиков и для юриков. Один, общий. Сделай, генеральный бесится. Я вам срок сдачи оттяну

Ещё нужно, чтобы по каждому договору можно было получить текстовую инфу – предмет, сумма, срок + нужная инфа физ. или юр. лица»

Journal

Нельзя жёстко привязываться к типу договора
Будем хранить ссылки на экземпляры базового класса

```
public class JournalEntry
{
    public Contract Contract { get; private set; }
    public DateTime Date { get; }
}

public class Journal
{
    public JournalEntry[] JournalEntries { get; } = new JournalEntry[JournalCapacity];
    public int JournalSize { get; private set; }
}
```

Вспоминаем про объекты в памяти

- *Объект производного класса хранит экземпляр базового класса (base). И может «подыграть» в его качестве*
  - T.e. IndividualContract – это Contract
  - И CompanyContract – это Contract

Кстати!

```
public JournalEntry[] JournalEntries { get; }
      = new JournalEntry[JournalCapacity];
```

Ни одного экземпляра JournalEntry не создано!

Создан массив ссылок (изначально null) на экземпляры, но ни одного объекта.

Как быть с объектами?

Добавляем новый договор в хранилище

```
public class Journal
{
    // …
    public void Add(Contract contract, DateTime date)
    {
        JournalEntries[JournalSize++]
            =new JournalEntry(contract,date);
    }
}
```

В месте использования Journal

```
Journal journal = new Journal();
//..
IndividualContract ic1 = new IndividualContract(
    "Development", new DateTime(2016, 3, 15), 100000);

CompanyContract cc1 = new CompanyContract(
    "Awesome Development", new DateTime(2016, 3, 15), 500000);

// В ic1 добавляют физ.лицо, ответственного
// В cc1 добавляют юр.лицо

journal.Add(ic1, DateTime.Now);
journal.Add(cc1, DateTime.Now);
```

Что произошло?

- Восходящее преобразование!
- IndividualContract и CompanyContract ограничили по родительскому интерфейсу Contract
- *А как выводить информацию, специфичную для каждого типа договоров?*

Информация о договоре

```
public class IndividualContract
    :Contract
{
    //...
    public void PrintInfo()
    {
        Console.WriteLine(
            $"{Subject}. Due to {DueTo}. Money: {Cost}");
        Console.WriteLine(Individual);
        Console.WriteLine(Responsible);
    }
}
public class CompanyContract
    :Contract
{
    //...
    public void PrintInfo()
    {
        Console.WriteLine($"{Subject}. Due to {DueTo}. Money: {Cost}");
        Console.WriteLine(Company);
    }
}
```

Пытаемся привести в порядок

```
public class Contract
{
    public void PrintInfo()
    {
        Console.WriteLine($"{Subject}. Due to {DueTo}. Money: {Cost}");
    }
}
public class CompanyContract : Contract
{
    public void PrintInfo()
    {
        base.PrintInfo();
        Console.WriteLine(Company);
    }
}
public class CompanyContract : Contract
{
    public void PrintInfo()
    {
        base.PrintInfo();
        Console.WriteLine(Company);
    }
}
```

И вывести всю инфу о договорах

```
public class Journal
{
    public void PrintAllInfo()
    {
        foreach (var entry in JournalEntries)
            entry.Contract.PrintInfo();
    }
}
```

У нас три разных PrintInfo. Который вызовется?

Сокрытие методов

Специфичные для типа вызовы PrintInfo у договоров не сработают

Наши методы будут работать как нам нужно, только если обращаться к экземплярам по ссылке на их «настоящий» тип

Происходит ***сокрытие методов родительского класса*** в классах IndividualContract и CompanyContract

Сокрытие методов

Вот что происходит с PrintInfo:
(можно увидеть соответствующий warning при компиляции)

```
public class IndividualContract : Contract
{
    //...
    public new void PrintInfo()
    {
        base.PrintInfo();
        Console.WriteLine(Individual);
        Console.WriteLine(Responsible);
    }
}
```

Что нужно нам?

Сокрытие методов нам не подходит.

Нам нужно, чтобы при обращении через ссылку на экземпляр родительского класса вызывались специфичные для конкретных типов договоров методы PrintInfo

Нужно **переопределить** методы!

Переопределение

Переопределение – механизм, позволяющий через общий интерфейс базового типа вызывать методы экземпляров производного типа

- Метод базового класса должен быть **виртуальный (virtual)**
- Метод производного класса должен быть помечен как **переопределённый (override)**

Кстати, в Java все методы виртуальные. Им несколько проще со всем этим…

Что изменится в договорах?

```
public class Contract
{
    public virtual void PrintInfo()
    {
        Console.WriteLine($"{Subject}. Due to {DueTo}. Money: {Cost}");
    }
}

public class CompanyContract : Contract
{
    public override void PrintInfo()
    {
        base.PrintInfo();
        Console.WriteLine(Company);
    }
}
```

Связывание (binding)

- Присоединение тела метода к вызову
- contact.PrintInfo() – вызов
- методы PrintInfo() есть у нескольких классов иерархии

Early binding vs Late binding

Реализация вызываемого метода может определяться во время выполнения и во время компиляции

- *Для обычных методов и сокрытия – **раннее связывание***
- *Для виртуальных методов и переопределений – **позднее связывание***
  - ещё есть dynamic, но о нём сильно позже…

Виртуальные методы

- Для реализации позднего связывания виртуальных методов используются ***таблицы виртуальных методов***, в которой хранятся ссылки на методы, которые нужно вызывать
  - Производные классы заменяют записи в таблице на «свои» методы, чтобы при обращении к ним через ссылку на базовый класс вызывался специфичный для них метод

Третий принцип ООП: Полиморфизм

Разное понимание

- ad hoc полиморфизм – один интерфейс, множество реализаций
  - статический полиморфизм при сокрытии методов родительского класса
  - сюда иногда добавляют перегрузку
- Полиморфизм наследования
  - то, что было у нас
- Параметрический полиморфизм
  - обобщенное программирование, когда тип параметра – тоже параметр

Полиморфизм в ООП - это

- Возможность реализовывать уникальное поведение у нескольких производных классов при едином интерфейсе базового класса
- Позволяет единообразно работать с разными объектами с учётом специфики типов этих объектов

Вопросы?

e-mail: marchenko@it.kfu.ru