№ 2232 МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ «МИСиС»

Кафедра автоматизированных систем управления

С.Э. Мурадханов
А.И. Широков

Информатика и программирование

Объектно-ориентированное программирование (на основе языка C#)

Учебник

Рекомендовано редакционно-издательским
советом университета

МИСиС

Москва 2015

УДК 681.3
М91

Р е ц е н з е н т
канд. техн. наук, доц. *И.Н. Лесовская*
(Национальный исследовательский университет «Высшая школа экономики»)

Мурадханов С.Э.

М91 Информатика и программирование : объектно-ориентированное программирование (на основе языка C#) : учеб. / С.Э. Мурадханов, А.И. Широков. – М. : Изд. Дом МИСиС, 2015. – 309 с.
ISBN 978-5-87623-801-6

Объектно-ориентированная парадигма (ОПП) является фундаментом разработки современных программных комплексов. Изучающая такие средства дисциплина имеет как теоретическую составляющую (объектно-ориентированный подход к анализу предметной области), так и практическую (синтез информационной системы с использованием инструментов объектно-ориентированной разработки).

Реализация программных систем на основе ООП может быть выполнена различными языковыми средствами. В учебнике представлены теоретические знания и практические навыки, необходимые для разработки на основе модели объектно-ориентированного программирования приложений в среде Visual Studio на языке программирования C#.

Учебник предназначен для студентов специальностей 230100 «Информатика и вычислительная техника», 231300 «Прикладная математика», 230700 «Прикладная информатика», 220700 «Автоматизация технологических процессов и производств».

УДК 681.3

ISBN 978-5-87623-801-6

© С.Э. Мурадханов,
А.И. Широков, 2015

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Учебник продолжает тему использования языка C# при разработке различных программных систем*. В нем описываются методология и способы использования языка программирования C# при разработке объектно-ориентированных программных систем. Приобретаемые при изучении изложенного здесь материала знания и навыки являются достаточными для применения объектно-ориентированного подхода на основе языка программирования C# и платформы Microsoft .Net Framework при решении практических задач (см. приложение).

Представленный материал предназначен для получения студентами знаний и навыков в области информационных технологий. Также он может быть полезен для всех желающих освоить язык программирования C# самостоятельно. Учебник содержит достаточно полную и систематизированную информацию по языковым конструкциям C#, методологии и способам их применения при реализации объектно-ориентированного подхода к разработке сложных программных систем.

Язык программирования C# нельзя изучать «линейно» – «от аксиом к теоремам, описаниям и задачам», поэтому изложение материала (объектно-ориентированное программирование на языке C#) будет проходить «по спирали». Некоторые понятия, использованные в той или иной программе-примере, имеют краткое описание, но в последующих разделах они будут полностью рассмотрены и объяснены.

---

* См. учебник: Мурадханов С.Э., Широков А.И. Информатика и программирование: Основы разработки программ на языке C#: Учеб. – М.: Изд. Дом МИСиС, 2013. – 304 с.

1. КЛАССЫ, ОБЪЕКТЫ, ЭЛЕМЕНТЫ, СОЗДАНИЕ ПРИЛОЖЕНИЙ

Объектно-ориентированное программирование и проектирование построено на классах. Программную систему, выстроенную в объектном стиле, можно рассматривать как совокупность классов, возможно, объединенных в проекты, пространства имен, решения (как это делается при программировании в Visual Studio .Net Framework). Класс – это обобщенное понятие, определяющее характеристики и поведение некоторого множества объектов, называемых экземплярами класса. «Классический» класс содержит данные, определяющие свойства объектов класса, и методы, определяющие их поведение.

Все классы библиотеки .Net Framework, а также все классы, которые создает программист в среде .Net Framework, имеют одного общего предка – класс object. У этого класса две различные роли: модуля и типа данных. Состав класса и его размер определяются не архитектурными соображениями, а той абстракцией данных (видом представления), которую должен реализовать класс. Объектно-ориентированная разработка программной системы основана на стиле, называемом проектированием от данных. *Проектирование* системы сводится к поиску абстракций данных, подходящих для конкретной задачи. Каждая из таких абстракций реализуется в виде класса, которые и становятся модулями – архитектурными единицами построения системы. В основе класса лежит абстрактный тип данных.

1.1. C# и ООП: классы, объекты, элементы, создание приложений

В ООП основными элементами программы являются объекты. ***Объекты*** – это программные конструкции, включающие набор логически связанных свойств (данных) и методов. Объекты – автономные «сущности», предоставляющие свою функциональность (функциональные возможности) другим компонентам среды приложения, изолируя от них свое внутреннее устройство. Объекты создаются на основе шаблонов, которые называются ***классами*** и являются экземплярами этих классов. Библиотека .Net платформы FCL предоставляет большой набор уже созданных классов, которые можно применять для использования объектов в приложениях. В приложениях можно

создавать классы, требующиеся для описания решаемой задачи. Описание класса «Автомобиль»:

```
class Автомобиль
{
   // описание данных
   // описание методов
}
```

1.1.1. Классы, объекты, элементы

Классы

В ООП классы – это «*шаблоны*» (*чертежи*) объектов. Они определяют все элементы объекта: свойства и поведение (методы), а также задают начальные значения для создаваемых объектов, если это необходимо.

Класс – механизм, задающий структуру (поля данных) всех однотипных объектов и их функциональность (механизм, определяющий все методы, относящиеся к объектам). Среди данных и методов класса (объекта) могут существовать такие, которые принадлежат не единичному объекту, а всем объектам класса (данные могут меняться у любых объектов класса, метод класса может оперировать с любыми объектами класса).

Класс может представлять собой:

- ***контейнер*** для методов класса и данных класса, принадлежащих всем объектам класса;
- ***«трафарет»*** («*шаблон*»), позволяющий создавать конкретные объекты (экземпляры) класса.

Класс *«Студент группы № курса»* может содержать:

- поля (данные) *объекта*: ФИО, оценки, книги из библиотеки;
- методы (данные) *объекта*: сдать экзамен, получить/сдать книги из библиотеки;
- поля (данные) *класса*: номер курса, даты экзаменов, количество предметов;
- методы (данные) *класса*: перевести группу на следующий курс – изменятся все данные класса (не все объекты останутся в измененном классе – не обязательно все студенты будут переведены на следующий курс).

Различие между данными и методами объектов и данными и методами их класса часто используется в C#. В определении (объявле-

нии) класса его данные и методы, принадлежащие всем объектам класса, описываются модификатором **static** (статический).

Поля и методы, как класса, так и формируемых с его помощью объектов, называются *членами класса*. При создании экземпляра класса в памяти создается копия этого класса. Созданный таким образом экземпляр класса называют ***объектом***. Для создания экземпляра класса используется специальная операция **new**:

```
// Объявление переменной типа Автомобиль
Автомобиль Auto; // переменная – это не объект класса!
// Создание экземпляра класса Автомобиль
// и сохранение ссылки на него в переменной myAuto
Auto = new Автомобиль();
```

При создании экземпляра класса выделяется блок оперативной памяти, в который записывается копия данных, и адрес этого блока присваивается переменной (переменная Auto хранит ссылку). Экземпляры класса не зависят друг от друга и являются отдельными программными конструкциями. Можно создавать произвольное число копий класса, которые могут существовать одновременно. Продолжая параллель с реальным миром, можно сказать: если считать конкретный автомобиль объектом, то чертежи автомобиля представляют собой класс Автомобиль. По чертежу можно сделать сколько угодно автомобилей. Если один из автомобилей будет работать не так, как все, это никак не повлияет на остальные автомобили.

У класса *две различные роли*: *модуля* и *типа данных*.

Класс – это модуль, архитектурная единица построения программной системы. Модульность построения – основное свойство программных систем. В ООП программная система, построенная по модульному принципу, состоит из классов, являющихся основным видом модуля. Модуль может не представлять собой содержательную единицу – его размер и содержание определяются архитектурными соображениями (можно построить монолитную систему, состоящую из одного модуля, – она может решать ту же задачу, что и система, состоящая из многих модулей).

Вторая роль класса не менее важна. ***Класс – это тип данных*** для реализации некоторой абстракции данных, характерной для задачи, в результате решения которой создается программная система. С этих позиций классы – не просто «кирпичики», из которых строится система. Каждый кирпичик имеет важную содержательную начинку. Сравним современный дом, построенный из обычных кирпичей, и

дом будущего, где каждый кирпич выполняет определенную функцию: один следит за температурой, другой – за составом воздуха в доме. ООП-система представляет собой дом будущего. Состав класса определяется не архитектурными соображениями, а той абстракцией данных, которую должен реализовать класс (в класс Account, реализующий абстракцию «Банковский счет», нельзя добавить поля из класса Car, реализующего абстракцию «Автомобиль»).

Разработка ООП-системы основана на стиле, называемом проектированием от данных. Проектирование системы сводится к поиску абстракций данных, подходящих для конкретной задачи. Каждая из таких абстракций реализуется в виде класса; эти классы и становятся модулями – архитектурными единицами построения системы. В основе класса лежит абстрактный тип данных. В хорошо спроектированной объектно-ориентированной системе каждый класс играет *обе роли* – каждый модуль системы имеет определенную смысловую нагрузку (типичная ошибка – рассматривать класс только как архитектурную единицу, объединяя под оболочкой класса разнородные поля и функции, в результате чего становится непонятно, какой же тип данных задает этот класс).

Объекты и элементы. Объекты состоят из элементов, к которым относят *поля*, *свойства*, *методы* и *события*, представляющие данные и функциональность объекта. ***Поля*** содержат данные объекта; ***свойства*** – предоставляют управляемый способ доступа к данным объекта; ***методы*** определяют действия, которые объект способен выполнять; ***события*** уведомляют заинтересованных пользователей (другие классы, которые могут принимать («подписываться на») эти события), если в объекте происходит что-то важное.

Свойства объектов класса Автомобиль, например такие как *Цвет*, *Модель*, *Расход_топлива*, являются данными, описывающими состояние объекта, а ***методы*** этих объектов *Нажать_акселератор*, *Переключить_передачу*, *Повернуть_руль* описывают функциональность автомобиля. ***Методы*** позволяют реализовать поведение объекта. ***События*** представляют собой уведомления о важных происшествиях в объекте (количество бензина стало ниже заданной величины или объект класса Двигатель может уведомить объект класса Автомобиль о событии *Перегрев_двигателя*). В этом случае описание класса Автомобиль выглядит следующим образом:

```
class Автомобиль
{  // описание свойств
```

```
    public string Модель;
    public float Расход_топлива;
    private int Число_цилиндров;
    // описание методов
    public void Повернуть_руль(){...}
    private void Регулировка_датчика(){...}
    // описание события
   event Событие Перегрев_двигателя;
}
```

Для доступа к полям, свойствам, методам, элементам объектов используется специальная операция **точка (.)**:

```
string Модель; // Объявление переменной Модель типа string
Модель = Auto.Модель; // определение модели автомобиля
Auto.Повернуть_руль(); // поворот руля автомобиля
```

К элементам объекта имеется доступ не из всех частей программы (из методов данного класса или других классов, из других сборок). Возможность использования элементов класса задается (явно или неявно) с помощью указания режима доступа. Режимы доступа: *private* – элемент, для которого он задан, можно использовать только в методах того класса, где он описан (закрытые элементы); *public* – элемент можно использовать в других классах (открытые элементы).

1.1.2. Инкапсуляция, вложенность, наследование, полиморфизм

Объект – программная конструкция, представляющая некоторую сущность. В нашей повседневной жизни сущностями, или объектами, например, можно считать автомобили, велосипеды, настольные компьютеры, банковский счет. Каждый объект обладает определенной функциональностью и свойствами. Объект представляет собой завершенную функциональную единицу, содержащую все данные и предоставляющую всю функциональность, необходимую для решения задачи, для которой он предназначен. Описание объектов реального мира при помощи программных объектов называют абстрагированием.

Инкапсуляция – отделение реализации объекта (его внутреннего содержания) от способа взаимодействия с ним. Другие объекты приложения взаимодействует с рассматриваемым объектом посредством имеющихся у него открытых *public*-свойств и методов, которые со-

ставляют его интерфейс. В общем виде под интерфейсом понимается открытый способ взаимодействия между разными системами. Если интерфейс класса меняется, то приложение сохраняет способность к взаимодействию с его объектами, даже если в новой версии класса его реализация значительно изменится. Объекты могут взаимодействовать друг с другом только через свои открытые методы и свойства, поэтому объект должен предоставлять доступ только к тем свойствам и методам, которые пользователям необходимы.

Интерфейс не должен открывать доступ к внутренним данным объекта, поэтому поля с внутренними данными объекта обычно объявляют с модификатором *private*. У объектов класса Автомобиль, которые могут взаимодействовать с объектами класса Водитель через открытый интерфейс, открытыми объявлены только методы *Ехатъ_вперед*, *Ехать_назад*, *Повернуть* и *Остановиться* – их достаточно для взаимодействия объектов классов Водитель и Автомобиль. У объекта класса Автомобиль может быть вложенный объект класса Двигатель, но он будет закрыт для объектов класса Водитель, которому будут открыты лишь методы, требуемые для управления автомобилем. Можно заменить вложенный объект класса Двигатель, и взаимодействующий с ним объект класса Водитель «не заметит» замены, если она не нарушит корректную работу интерфейса.

Классы разных объектов, как и в реальном мире, связаны между собой. Выделяются два основных типа взаимосвязи классов: *вложенность* и *наследование*. ***Вложенность*** – включение объектов одного класса в объекты другого класса. ***Наследование*** – описание одного класса на основе другого класса. Объекты одних классов могут *включать* объекты других классов в качестве своих полей и предоставлять к ним доступ, как и к другим своим элементам. Иерархия вложенности объектов друг в друга называется *объектной моделью* (object model). Объект класса Автомобиль, который сам по себе является объектом, состоит из ряда вложенных объектов, таких как объект класса Двигатель, четырех объектов класса Колесо, объект класса Трансмиссия. Компоновка вложенных объектов непосредственно определяет работу объекта класса Автомобиль. Например, поведение объектов Автомобиль, у которых свойство *Число_цилиндров* вложенного объекта Двигатель равно соответственно 4 и 8, будет различным. В свою очередь у вложенных объектов могут быть собственные вложенные объекты. Например, объект Двигатель (который является вложенным объектом объекта Автомобиль) может иметь вложенные

объекты Свеча_зажигания. Для получения сведений о марке свечей зажигания автомобиля можно записать следующее выражение:

Auto.Свеча_зажигания.Марка;

Один класс может быть описан на основе уже имеющегося описания другого класса. В этом случае между классами задается отношение *наследования*. Наследование позволяет создавать новые классы на основе существующих, при этом новые классы обладают всей функциональностью старых и при необходимости могут модифицировать их. Класс, объявленный на основе некоторого (базового) класса, называется производным, или классом-потомком. У любого класса может быть только один прямой предок – его базовый класс (*base class*). У производного класса окажется тот же набор элементов, что и у базового, но, при необходимости, к производному классу разрешается добавлять дополнительные элементы. Можно также изменить реализацию членов, унаследованную от базового класса, переопределив их. На основе описания класса Транспортное_средство (базовый класс), можно описать класс Автомобиль (производный класс), на основе которого, в свою очередь, можно описать классы Грузовик, Пассажирский_автомобиль и Спортивный_автомобиль (рис. 1.1).

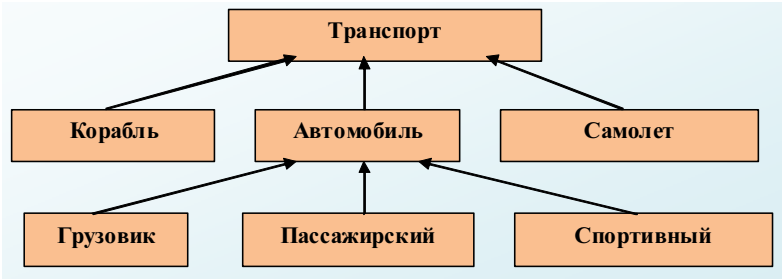


Рис. 1.1. Схема наследования классов

Пример описания производного класса Автомобиль:

```
class Автомобиль : Транспорт
{// описание свойств
   public string Модель;
   public float Расход_топлива;
   private int Число_цилиндров;
  // описание методов
```

```
    public void Повернуть_руль() { }
    private void Регулировка_датчика() { }
   // описание события
    event Событие Перегрев_двигателя;
}
```

В производном классе сохраняется функциональность (поведение), определенная в базовом классе. Кроме того, в производных классах могут описываться новые элементы и переопределяться методы, описанные в базовых классах. Для указания возможности доступа к наследуемым элементам используются следующие режимы доступа: *public*, *protected*.

Полиморфизм состоит в том, что одни и те же открытые интерфейсы можно по-разному реализовать в разных классах. Другими словами, полиморфизм позволяет вызывать методы и свойства объекта независимо от их реализации. Объект класса Водитель взаимодействует с объектом класса Автомобиль через открытый интерфейс. Если другой объект Грузовик или Гоночный_автомобиль поддерживает такой открытый интерфейс, то объект класса Водитель сможет взаимодействовать и с ними (управлять ими), невзирая на различия в реализации интерфейса. Основных подходов к реализации полиморфизма два: через интерфейсы и через наследование.

Реализация полиморфизма с помощью интерфейсов. Интерфейс (interface) – это соглашение, определяющее набор открытых методов, реализованных классом. Интерфейс определяет список методов класса, но не описывают реализацию классов. В объекте допустимо реализовать несколько интерфейсов, а один и тот же интерфейс можно реализовать в разных классах.

Описание интерфейса возможности управления некоторыми объектами (имена интерфейсов обычно начинаются с буквы I):

```
interface IУправлять
{
   int Ехать(...);
   float Повернуть(...);
   bool Остановиться(...);
}
```

Если класс реализует интерфейс, то в нем должны быть описаны все методы этого интерфейса:

```
class Автомобиль : Транспорт, IУправлять
{
   int Ехать(...) {<реализация метода>};
   float Повернуть(...){<реализация метода>};
   bool Остановиться(...){<реализация метода>};
   // описание других элементов ...
}
```

Любые объекты, в которых реализован интерфейс, способны взаимодействовать друг с другом с его помощью. Интерфейс **IУправлять** можно реализовать и в других классах, таких как Грузовик, Автопогрузчик или Катер. В результате эти объекты получат возможность взаимодействия с объектом класса Водитель. Объект класса Водитель находится в полном неведении относительно реализации интерфейса, с которым он взаимодействует, ему известен лишь сам интерфейс.

Реализация полиморфизма через наследование. Производные классы сохраняют все характеристики своих базовых классов и способны взаимодействовать с другими объектами под видом *экземпляров базового класса* (переменным базового типа можно присваивать ссылки на объекты производных классов):

```
Автомобиль Auto; // переменная – это не объект класса!
Спортивный_автомобиль sportAuto = new Спортивный_автомобиль();
// можно присвоить, так как есть наследование
Auto = sportAuto;
```

В этом случае можно работать с объектом производного класса так, как если бы он был объектом базового класса.

1.1.3. Создание приложений C#

C# – структура приложения. Платформа .Net и язык C# полностью соответствуют принципам и методологии ООП – программа (программная система) состоит только из описанных разработчиком типов данных, а также использует типы данных, расположенных в библиотеке FCL и других сборках, на которые делаются ссылки (рис. 1.2).

> **Примечание.** *Так как программа на языке C# – это просто набор объявлений разных типов, то изучение данного языка состоит в изучении того, как создавать и использовать типы (встроенные, библиотечные, пользовательские).*

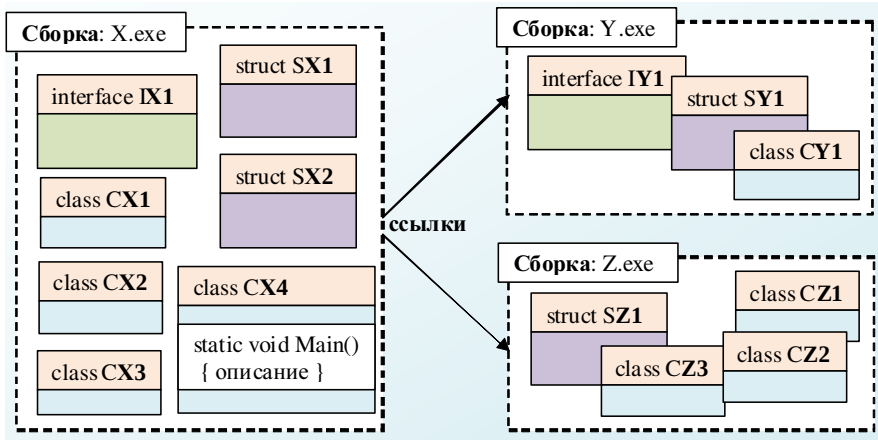


Рис. 1.2. Распределение пользовательских типов программ по сборкам

В платформе .Net под типами понимаются классы, структуры, интерфейсы, перечисления и делегаты. Программа – это набор типов, в основном – классов. Выполнение программы заключается в создании объектов этих классов и в вызове методов этих объектов для манипулирования их данными. Любая (даже самая простая) программа на языке C# состоит из классов (в самом простом варианте – включает один класс). ***Один из классов программы на языке C# должен обязательно включать статический открытый метод – static Main(), с выполнения которого начинается работа любого приложения C#.***

Общая схема описания приложения в виде набора пользовательских типов представлена на рис. 1.3. Основные части программы, такие как пространства имен и классы, состоят из заголовков и блоков. В заголовке указывается тип элемента программы и его имя, например namespace Name_nnn или class CX1. Блок образуется с помощью пары фигурных скобок (**{}**), между которыми записываются объявления различных типов, либо элементы классов, либо набор операторов языка C#.

C#-программа состоит из описаний пользовательских типов (в основном классов):

- описания классов состоят из описания полей (переменных) и методов;
- описание переменных состоит из указания типа и имени переменной;

- описание методов состоит из описания локальных переменных и набора операторов;
- оператор состоит из набора ключевых слов и выражений;
- выражения состоят из переменных и констант, связанных знаками операций.

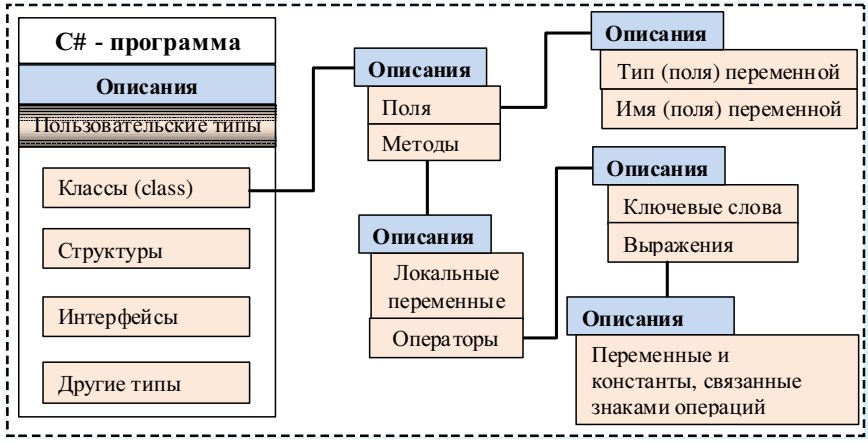


Рис. 1.3. Общая структура программы (приложения)

Примечание. *В C# большинство операторов заканчивается символом «точка с запятой» (;). Несколько операторов могут быть записаны в одной строке, или один оператор может быть записан в нескольких строках. Операторы могут быть объединены в блоки с помощью фигурных скобок ({}). В текст программы могут быть вставлены комментарии (компилятор их не обрабатывает). Комментарий, расположенный на одной строке, начинается с двух последовательных символов «косая черта» (//), а комментарий, расположенный на нескольких строках, начинается с символов /* и заканчивается символами */. Для всех элементов языка C#, таких как переменные, поля, методы, классы и т.п., задаются имена – идентификаторы.*

C# позволяет осуществлять разработку приложений с консольным (консольные приложения) и графическим интерфейсом (Windows, WPF, Web-приложения).

Пример 1.1. ***Консольное приложение для вычисления площади круга***

```
 1 - using System;
 2 - namespace ConsoleApplication
 3 - {
 4 - class Program
 5 -  {
 6 -   static void Main()
 7 -   {
 8 -       Console.Write("Введите радиус круга: ");
 9 -       string  s = Console.ReadLine();
10 -       double r = Convert.ToDouble(s);
11 -       double p = Math.PI * r * r;
12 -       Console.WriteLine("Площадь круга = {0}", p);
13 -       Console.ReadLine();
14 -       return;
15 -    }
16 -  }
17 - }
```

Результат выполнения

```
Введите радиус круга: 10
Площадь круга =314,159265358979
```

В 1-й строке текста простой программы указано, какое пространство имен (в данном случае System) будет использоваться. Пространства имен – это способ объединения в группы связанных классов. Для каждой программы задается свое пространство имен с помощью ключевого слова namespace, с которым будут связываться все описанные в программе классы. Директива **using** указывает на пространство имен, которое должен просматривать компилятор для поиска описания классов, у которых не задано полное имя. Причиной использования директивы **using** в приведенной программе является то, что в ней используются классы из библиотеки **FCL**, которые включены в пространство имен System. Директива using System позволяет ссылаться на классы System.Console и System.Convert как на Console и Convert, без указания пространства имен. В стандартном пространстве имен System содержатся базовые типы данных платформы .Net. Функциональность языка C# в значительной степени основывается на базовых классах платформы .Net. ***Сам язык C# не имеет встроенных операторов ввода-вывода, также как и встроенных типов, а использует базовые типы, описанные в библиотеке FCL платформы .Net.***

Во 2-й строке программы объявляется новое пространство имен ConsoleApplication, в котором описывается один класс Program. Полным именем данного класса является ConsoleApplication.Program. Весь код программы C# должен содержаться только внутри классов. Описание классов в C# состоит из ключевого слова class, за которым стоит название класса и пара фигурных скобок (блок). Весь код, связанный с классом, должен быть записан между этими фигурными скобками. В классе Program описан только один метод (6-я строка) с именем Main(). Данный метод (этот метод C# соответствует методу main() в языках C++ и Java) запускается автоматически (является точкой входа) при запуске созданной компилятором программы на выполнение. Он может возвращать целое значение (int) или не возвращать ничего (void). Описание методов в C# имеет следующую структуру:

[модификаторы] тип_результата имя_метода ([параметры])
{ // содержание метода }

В квадратных скобках указывают необязательные элементы описания. Модификаторы используются для задания некоторых особенностей методов, например таких как «откуда и как данный метод может вызываться». В примере 1.1 программы используются два модификатора: public и static. Модификатор (режим доступа) public означает, что данный метод может быть доступен из методов любых классов. Модификатор static указывает, что данный метод не связан с конкретным экземпляром класса и может быть вызван без использования ссылки на экземпляр. Это важно при запуске программы на выполнение без создания экземпляра конкретного класса. В данном примере задан тип результата void (это означает, что никакого результата нет) и для метода Main() не описаны передаваемые параметры.

Первый оператор метода Main() (см. пример 1.1) выполняет вывод подсказки с помощью метода Write() класса Console (данный метод выводит строку текста на экран в консольное окно, но не переводит курсор на начало следующей строки):

Console.Write("Введите радиус круга:");

Для ввода данных пользователя используется метод ReadLine() класса Console для получения данных с клавиатуры (при этом сразу же объявляется переменная s типа string):
string s = Console.ReadLine();

Пользователь может ввести число (для отделения дробной части используется запятая, а не точка, как в коде программы, например: 2,5) и нажать клавишу [Enter]. Метод ReadLine() возвращает текстовую строку (тип string). Для того чтобы преобразовать значение текстовой строки s в вещественное значение для переменной r (которая также объявляется в данной строке как тип double) используется метод ToDouble(s) класса Convert:

double r = Convert.ToDouble(s);

Полученное значение радиуса используется для вычисления площади круга, которое сохраняется в переменной p:

double p = Math.PI * r * r;

(Для вычисления используется значение числа π, которое можно получить из статического класса System.Math библиотеки FCL.)

Для вывода вычисленного значения используется метод WriteLine() со строкой форматирования:

Console.WriteLine("Площадь круга = {0}", p);

Для того чтобы консольное окно не закрылось автоматически после выполнения программы, используется метод ReadLine(), который приостанавливает выполнение программы до нажатия клавиши [Enter]:

Console.ReadLine();

После этого выполняется оператор **return**, который вызывает завершение работы метода (обычно данный оператор возвращает результирующее значение, но так как для данного метода указан тип void в заголовке, то никакого значения не возвращается – в этом случае данный оператор можно было бы и не записывать).

Создание выполняемой программы. Инструментом преобразования исходного текста программы на языке C# в выполняемый модуль для платформы .Net является компилятор. Можно составить программу, записать ее с помощью стандартной программы «Блокнот» в текстовый файл и преобразовать в сборку с помощью компилятора csc.exe:

csc.exe <имя файла с программой> [опции компиляции]

Основные опции компилятора:

- /target:[exe | winexe | library | module] – тип создаваемого модуля: exe – консольное приложение (по умолчанию); winexe – Windows-

приложение; library – библиотека классов (без метода Main); module – модуль (в модуль не добавляется декларация);

- /reference:<список файлов> – имена сборок (собственных или из FCL), на которые будут делаться ссылки в создаваемом модуле; основные сборки FCL (такие как System.dll подключаются по умолчанию и их указывать не требуется);
- /out:<имя файла> – имя создаваемого модуля, если оно не совпадает с именем входного файла.

Для упрощения и автоматизации разработки программ на языке C# используется интегрированная система разработки Visual Studio .Net Professional (VS Professional) или Visual C# Express Edition (VS EE). Эти системы используют компилятор языка C# и предоставляют большое количество средств автоматизации создания и отладки программ.

Система разработки Visual C# Express Edition позволяет создавать консольные и Windows-приложения на языке C#. Система Visual Studio .Net Professional позволяет создавать консольные, Windows- и Интернет-приложения на разных языках. Разработка приложений в этих системах основано на понятии ***проект***, под которым понимается множество файлов с описаниями классов, ссылками на используемые сборки, с другими типами данных, а также с параметрами для запуска компиляции. Все файлы проекта хранятся в одной специально создаваемой папке. Кроме понятия проекта используется понятие ***решения***. *Решение* содержит один или несколько *проектов*, ресурсы, необходимые этим *проектам*, возможно, дополнительные файлы, не входящие в *проекты*. Один из *проектов решения* должен быть указан как ***стартовый проект***. Выполнение *решения* начинается со *стартового проекта*. *Проекты* одного *решения* могут быть *зависимыми* или *независимыми*. *Стартовый проект* должен иметь точку входа – класс, содержащий статический метод с именем **Main()**, которому автоматически передается управление в момент запуска *решения* на выполнение. В уже имеющееся *решение* можно добавлять как новые, так и существующие *проекты*. Для каждого решения создается папка, в которой для каждого проекта решения создаются собственные подпапки с результатами компиляции приложения.

Проект – это основная единица, с которой работает разработчик. Он выбирает тип *проекта*, а Visual Studio создает «скелет» *проекта* в соответствии с выбранным типом. Рассмотрим сопряженные понятия – *решение (solution)*, *проект (project)*, *пространство имен (namespace)*, *сборка (assembly)* – как результат работы компилятора Visual Studio с

позиций программиста, работающего над *проектом*, и с позиций CLR, компилирующей PE-файл в исходный код процессора.

С точки зрения программиста, компилятор создает ***решение***, с точки зрения CLR – ***сборку***, содержащую PE-файл. Программист работает с *решением*, CLR – со *сборкой*.

Решение содержит один или несколько *проектов*, ресурсы, необходимые этим *проектам*, возможно, дополнительные файлы, не входящие в *проекты*. Один из *проектов решения* должен быть выделен и назначен *стартовым проектом*. Выполнение *решения* начинается со *стартового проекта*. *Проекты* одного *решения* могут быть *зависимыми* или *независимыми*. Например, все *проекты* одного раздела данного учебника могут быть для удобства собраны в одном *решении* и иметь общие свойства. Изменяя *стартовый проект*, получаем возможность перехода к нужному примеру. Следует отметететить, что *стартовый проект* должен иметь точку входа – класс, содержащий статическую процедуру с именем Main, которой автоматически передается управление в момент запуска *решения* на выполнение. В уже имеющееся *решение* можно добавлять как новые, так и существующие *проекты*. Один и тот же *проект* может входить в несколько *решений*.

Проект состоит из классов, собранных в одном или нескольких *пространствах имен*. *Пространства имен* позволяют ***структурировать*** *проекты*, содержащие большое число классов, объединяя в одну группу близкие классы. Если над *проектом* работает несколько исполнителей, то, как правило, каждый из них создает свое *пространство имен*. Помимо структуризации, это дает возможность присваивать классам имена не задумываясь об их уникальности. В разных *пространствах имен* могут существовать одноименные классы. *Проект* – это основная единица, с которой работает программист. Он выбирает тип проекта, а Visual Studio создает скелет проекта в соответствии с выбранным типом.

Создание проекта консольного приложения. Для создания проекта, в котором будет создаваться консольное приложение, нужно выполнить команду меню **File → New → Project** и выбрать вид проекта – Console Application (**Файл → Создать → Проект → Консольное приложение**), задать имя проекта HelloAplication и указать папку, в которой будет храниться проект (рис. 1.4). Если принять эти установки, то среда Visual Studio создаст решение, имя которого совпадает с именем проекта.

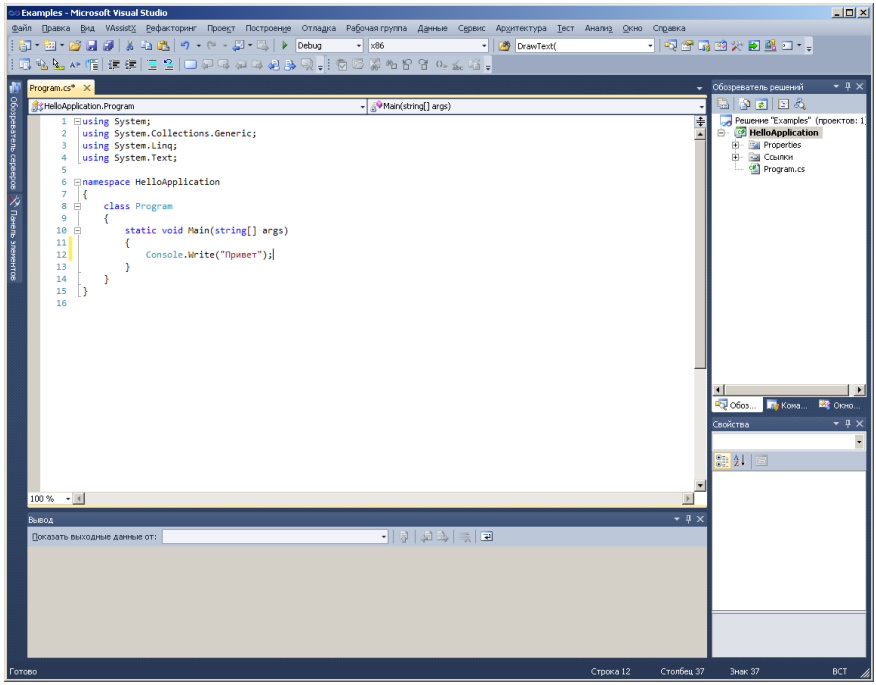


Рис. 1.4. Окно создания нового проекта

На рис. 1.5 показано, как выглядит это решение в среде разработки. Интегрированные системы разработки являются многооконными, настраиваемыми и обладают большим набором возможностей. В окне *Solution Explorer* (Обозреватель решений) представлена структура создаваемого решения. В окне *Properties* (Свойства) можно увидеть свойства выбранного элемента программы. В окне *Редактора текстов* (центральное окно) отображается выбранный документ, в данном случае программный код класса *проекта* – HelloAplication.**Program** (в этом окне можно отображать и другие документы, список которых показан в верхней части окна). В окне *Output* (под окном *Редактора*) отображаются результаты компиляции программы (панель Error List), результаты ее выполнения (панель Output) или результаты поиска (панель Find Symbols Results). Построенное *решение* содержит единственный заданный *проект* – HelloAplication. Создаваемый *проект* включает в себя логические папки со свойствами проекта и со ссылками на подключенные *пространства имен* из библиотеки FCL и файла с расширением .cs. Файл со стандартным именем Program

является построенным по умолчанию классом, который задает точку входа – метод Main() (в данном случае пустой).

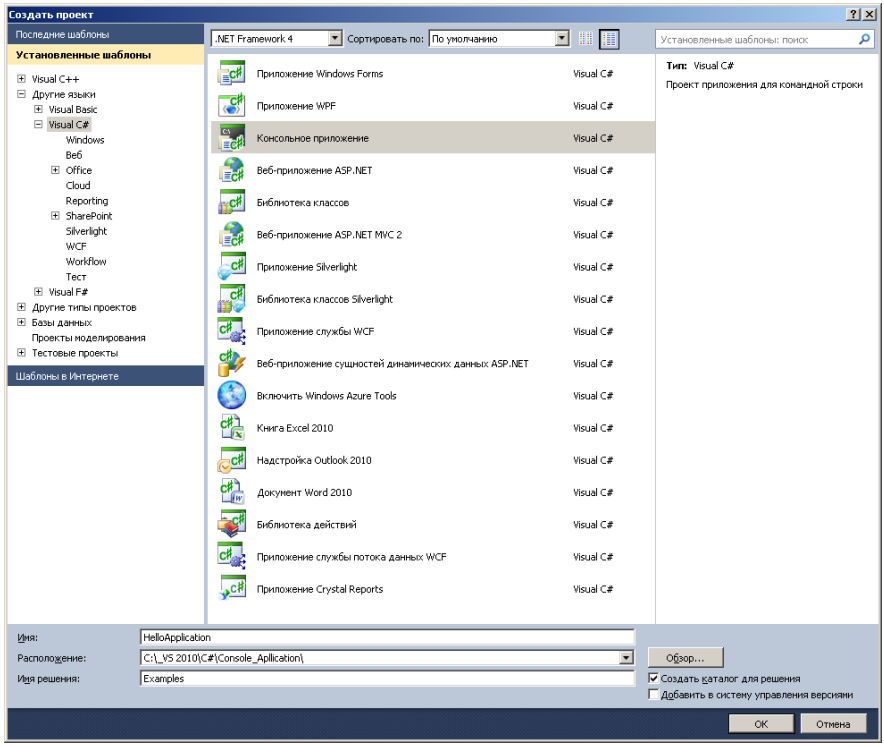


Рис. 1.5. Окно среды разработки и построенное консольное приложение

Класс *проекта* включен в *пространство имен*, которое по умолчанию имеет то же имя, что и *решение*, и *проект*. При создании нового *проекта* автоматически создается специальная структура – *решение*, которая включает *проект*, содержащий *пространство имен*, в котором описан класс с методом Main, являющимся точкой входа в программу. Для простых *решений* такая структура представляется избыточной, но позволяет хорошо структурировать сложные приложения.

Функциональность консольного *проекта*, построенного по умолчанию, небольшая. Его можно скомпилировать, выбрав соответствующий пункт (команду) из меню Build. Если компиляция прошла

без ошибок, то в результате создается *сборка* и в папке Debug появляется EXE-файл разрабатываемого *проекта*. Приложение можно запустить нажатием клавиш [CTRL] + [F5] или выбором соответствующего пункта из меню Debug. Приложение будет выполнено под управлением среды CLR.

Параметры метода Main(). Выполнение программы сводится к вызову методов в определенной последовательности. Методу могут передаваться аргументы, а из метода – возвращаться значение. С другой стороны, сама программа может быть запущена из другой программы и может запускать другую программу. В рамках данного вопроса рассматривается вариант, когда программа запускается из среды исполнения. Поскольку метод Main() выполняется первым при запуске программы, то именно с помощью этого метода производится прием аргументов при запуске программы (запуск приложения и использование аргументов метода Main() представлены на рис. 1.6). Для запуска программы в среде исполнения формируется команда. Команда должна содержать спецификацию исполнимого файла и, возможно, аргументы, которые в команде указываются через пробел:

C:\TEST\TEST\BIN\DEBUG\test.exe_аргумент1_ аргумент2_... _аргументN

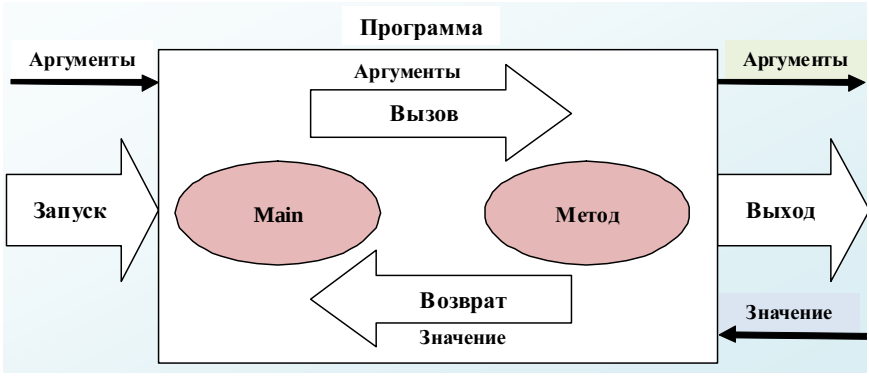


Рис. 1.6. Запуск программы – параметры метода Main()

Состав параметров метода Main фиксирован: **Main (string[] args)**. Метод принимает параметры из команды в виде массива строк. Каждая строка хранит символьное представление одного аргумента.

Пример. Разработать программу, предназначенную для хранения в виде массива цен на товары и их изменения на заданную величину. Изменение выполняется методами Plus (увеличение), Minus (уменьшение). Исполняемый файл хранится в файле test.exe.

Аргументы командной строки: 1) Сумма; 2) Вид операции: Плюс (в любом регистре) или +, Минус (в любом регистре) или –; 3) Фамилия оператора (ключ).

Примерный вид командной строки:

C:\ TEST\TEST \BIN\DEBUG\ test.exe 50 Плюс Иванов

Пример 1.2. ***Использование параметров метода Main()***

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _Main
{
   class Program
   {
      public static void Plus(double[] p, double s)
      {// массив цен, сумма, на которую увеличивается цена
         for (int i = 0; i < p.Length; i++)
            p[i] += s;
      }
      public static void Minus(double[] p, double s)
      {
         for (int i = 0; i < p.Length; i++)
            p[i] –= s;
      }
      static void Main(string[] args)
      {
         double[] pr; //массив с ценами
         double delta; //сумма
         string key = "Иванов"; //ключ
         pr = new double[] { 100.0, 200.0, 300.0 };
         if (args.Length != 3) //проверка количества переданных ар-
гументов
         {
            Console.WriteLine("Должно быть 3 аргумен-
та...Завершение!");
```

```
            Console.ReadLine();
            return;
         }//завершение программы – ошибка №1
         if (args[2] != key) //проверка фамилии
         {
            Console.WriteLine("Не совпадает фами-
лия...Завершение!");
            Console.ReadLine();
            return;
         }//завершение программы – ошибка №2
         delta = double.Parse(args[0]);//пересчет цен
         switch (args[1].ToUpper())//чтение операции
         {
            case "ПЛЮС":
            case "+": Plus(pr, delta); break;
            case "МИНУС":
            case "–": Minus(pr, delta); break;
         }
         Console.WriteLine("Программа Main(). Новые це-
ны:");//вывод новых цен
         for (int i = 0; i <= pr.Length – 1; i++)
            Console.Write(" pr[{0}] = {1} ", i, pr[i]);
         Console.ReadLine();
         return;
      }
   }
    }
```

Результат выполнения

```
C:\Test>Main.exe
Должно быть 3 аргумента...Завершение!

C:\Test>Main.exe 50 Плюс Петров
Не совпадает фамилия...Завершение!

C:\Test>Main.exe 50 Плюс Иванов
Программа Main(). Новые цены:  pr[0] = 150  pr[1] = 250  pr[2] = 350

C:\Test>Main.exe 122 + Иванов
Программа Main(). Новые цены:   pr[0] = 222  pr[1] = 322  pr[2] = 422
```

```
C:\Test>Main.exe 17 – Иванов
Программа Main(). Новые цены:   pr[0] = 83  pr[1] = 183  pr[2] = 283

C:\Test>Main.exe 3 минус Иванов
Программа Main(). Новые цены:  pr[0] = 97  pr[1] = 197  pr[2] = 297
```

1.2. Описание класса

Классы позволяют создавать собственные типы, которые могут использоваться точно так же, как и встроенные типы.

Синтаксис объявления класса:

[атрибуты] [спецификаторы] [partial] class имя_класса
[: имя_родителя [, интерфейс_№1, … , интерфейс_№K]]
{ тело_класса }

Описание класса содержит слово *class*, *имя класса* и *тело класса* (в фигурных скобках). Для класса можно задать базовые классы (класс-родитель и интерфейсы), атрибуты и спецификаторы, определяющие различные характеристики класса.

Описатель класса **partial** указывает на то, что объявление класса может быть записано в нескольких файлах. Класс (и все его элементы) имеет режим доступа.

Спецификаторы определяют свойства класса (*new*, *abstract*, *sealed*) и доступность класса (*public*, *protected*, *internal*, *private*) для других элементов программной системы.

Спецификаторами могут быть: *public, private* (только для вложенных структур).

Ключевое (зарезервированное) слово **class** определяет класс.

Элемент **имя_ класса** – означает имя класса.

Интерфейсы, реализуемые классом, перечисляются через запятую.

Описание пользовательского типа – класса Person:

```
class Person
{
   public const int Person_Max = 20; //максимальное число персон
   private string name;                    // задается значение ""
   private int age;                         // задается значение 0
   private double salary;                   // задается значение 0.0
   public Person(string name, int age, double salary)
   {
      this.name = name;
```

```
        this.age = age;
        this.salary = salary;
    }
    public void PrintPerson()
    {
        Console.WriteLine("name= {0}, age = {1}, salary ={2}", name,
age, salary);
    }
}
```

Примечание. *Предопределенное поле* ***this*** *– ссылка на тот объект, в котором выполняется вызываемый метод.* ***this.name = name;*** *в данном случае* ***this.name*** *– поле* ***name*** *текущего объекта, а присваиваемое значение* ***name*** *– это параметр метода.*

Объект *(экземпляр класса) создается явным образом с помощью операции* ***new().***

Класс *можно описывать непосредственно внутри* ***пространства имен*** *или внутри другого* ***класса*** *(в таком случае класс называется* ***вложенным****).*

Используя описание класса Person можно объявлять переменные класса и создавать его экземпляры (объекты):

```
Person p;
p = new Person("Петров А.И. ", 25, 28000); // Создается экземпляр
класса Person
```

Спецификаторы public, protected, internal, protected internal, private называются *спецификаторами доступа* и определяют, откуда можно непосредственно обращаться к данному классу. Спецификаторы доступа могут комбинироваться с остальными спецификаторами. Независимо от значения режима доступа все *элементы класса* доступны в его *методах*. Если элементы имеют режим доступа private (возможно, опущенный), то тогда они доступны только в *методах* самого класса. Такие ***элементы*** называются ***закрытыми***. Если *элементы* класса должны быть доступны для *методов* класса B, которому доступен сам класс A, то они должны быть описаны с атрибутом **public** (открытые элементы). ***Закрытые элементы*** составляют важную часть класса, позволяя клиентам класса не вникать в детали реализации класса. Открытые элементы класса описывают интерфейс класса (способ взаимодействия с объектами класса). Если *элементы класса* A должны быть доступны для вызовов в *методах*

класса B, являющегося потомком класса A, то такие *элементы* описываются с атрибутом **protected**.

Для каждого объекта при его создании в памяти выделяется отдельная область, в которой хранятся его данные. В классе могут присутствовать ***статические*** элементы, которые существуют в единственном экземпляре для всех объектов класса. Статические данные часто называют ***данными класса***, а остальные – ***данными экземпляра***. Для работы с данными класса используются статические методы класса, для работы с данными экземпляра – методы экземпляра, или просто методы.

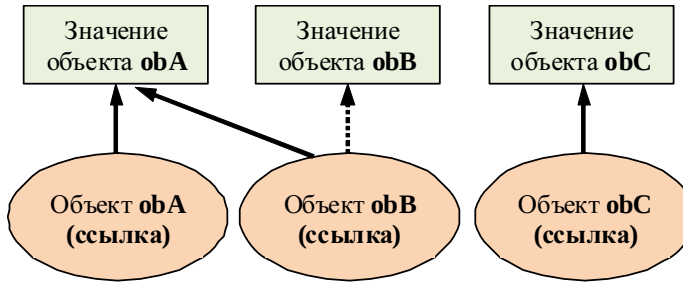
Тело класса может содержать (в теле класса могут быть объявлены):

- константы;
- поля;
- конструкторы и деструкторы;
- методы;
- события;
- делегаты;
- классы (структуры, интерфейсы, перечисления).

Из синтаксиса следует, что классы могут быть *вложенными* (*вложенные классы* целесообразно использовать, когда некоторый класс носит вспомогательный характер, разрабатывается в интересах другого класса, и точно известно, что внутренний класс нигде не понадобится, кроме класса, в который он вложен). ***Основу любого класса составляют его конструкторы, поля и методы***.

Присваивание и сравнение объектов. Механизм выполнения присваивания один и тот же для величин любого типа (как ссылочного, так и размерного), однако при присваивании значений – копируется значение, а при присваивании ссылок – копируется ссылка, поэтому после присваивания одного объекта другому будут получены две ссылки, указывающие на одну и ту же область памяти. При создании объектов **obA**, **obB** и **obC**, а затем выполнении присваивания **obB = obA** ссылки **obB** и **obA** указывают на один и тот же объект. Предыдущее значение **obB** становится недоступным и очищается «сборщиком мусора».

Аналогична ситуация и с операцией проверки на равенство. *Величины значимого типа равны*, если равны их значения. *Величины ссылочного типа равны*, если они ссылаются на одни и те же данные. Так, объекты **b** и **c** равны, так как они ссылаются на одну и ту же область памяти. Но **a** не равно **b** даже при равенстве их значений.



1.3. Поля класса

Состояние объектов класса (а также структур) задается с помощью переменных, которые называются полями (fields). При создании объекта – экземпляра класса, в динамической памяти («куче») выделяется участок памяти, содержащий набор полей, определяемых классом, и в них записываются значения, характеризующие начальное состояние данного экземпляра.

Описание полей класса. Поля класса синтаксически являются обычными переменными (объектами) языка. Их описание удовлетворяет обычным правилам объявления переменных, с точки зрения содержательности поля задают представление той самой абстракции данных, которую реализует класс (табл. 1.1). Поля характеризуют свойства объектов класса. Когда создается новый объект класса (в динамической памяти или в стеке), то этот объект представляет собой набор полей класса. ***Два объекта одного класса имеют один и тот же набор полей, но могут иметь разные значения, хранимые в этих полях. Поля (данные), содержащиеся в классе, могут быть переменными или константами.***

Таблица 1.1

Спецификаторы для полей (данных) класса

Спецификатор	Описание
new	Новое описание поля, скрывающее унаследованный элемент класса
public	Доступ к элементу не ограничен
protected	Доступ только из данного и производных классов
internal	Доступ только из данной сборки
protected internal	Доступ только из данного и производных классов и из данной сборки
private	Доступ только из данного класса
static	Одно поле для всех экземпляров класса
readonly	Поле доступно только для чтения (значение таких полей можно установить либо при описании, либо в конструкторе)
volatile	Поле может изменяться другим процессом или системой