БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра технологии программирования

Ж. В. Василенко

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ПЛАТФОРМЕ .NET

Учебно-методическое пособие
для студентов специальностей
1-31 03 03 «Прикладная математика (по направлениям)»,
1-31 03 04 «Информатика»,
1-31 03 05 «Актуарная математика»,
1-31 03 06 «Экономическая кибернетика (по направлениям)»

МИНСК
2008

УДК 004.43 (075.8)
ББК 32.973.26 – 018.1я73
В19

Утверждено на заседании
кафедры технологий программирования БГУ
8 декабря 2008 г., протокол №5

Василенко, Ж. В.

В19 Разработка приложений на платформе .NET: учеб.-метод. пособие для студентов спец. 1-31 03 03 «Прикладная математика (по направлениям)», 1-31 03 04 «Информатика», 1-31 03 05 «Актуарная математика», 1-31 03 06 «Экономическая кибернетика (по направлениям)» / Ж. В. Василенко. – Минск : БГУ, 2008. – 63 с.

В пособии рассматривается среда программирования, платформа .NET, распределение и управление памятью, динамическое определение типов, основы синтаксиса и конструкций языка C#.

Пособие предназначено для студентов БГУ, обучающихся по специальностям «Прикладная математика», «Информатика», «Актуарная математика», «Экономическая кибернетика».

УДК 004.43 (075.8)
ББК 32.973.26-018.1я73

ВВЕДЕНИЕ

В середине 2000 года корпорация *Microsoft* представила новую модель для создания приложений, основой которой является платформа .NET[1]. Платформа .NET образует каркас, который включает **технологии разработки** *Windows*-приложений, *Web*-приложений и *Web*-сервисов, **технологии доступа к данным** и **межпрограммного взаимодействия**, **технологии межъязыкового взаимодействия** (*cross-language interoperability*) программных и аппаратных изделий разных поставщиков, или **многоязыковое программирование** (*mixed-language programming*).

В состав платформы входит обширная библиотека классов. Основным инструментом для разработки является интегрированная среда *MS Visual Studio*.

Платформа .NET позволяет с легкостью создавать и интегрировать приложения, написанные на различных языках программирования. Специально для .NET был разработан язык программирования C#. Этот язык сочетает простой синтаксис, похожий на синтаксис языков C++ и *Java*, и полную поддержку всех современных объектно-ориентированных концепций и подходов. В качестве ориентира при разработке языка было выбрано безопасное программирование, нацеленное на создание надежного, простого в сопровождении кода.

Цель данного курса – рассмотреть программирование для платформы .NET с использованием языка программирования C#.

Пособие содержит фрагменты кода и небольшие программы, иллюстрирующие теоретический материал. Примеры могут служить основой при написании лабораторных работ, связанных с объектно-ориентированным программированием с использованием C#.

[1] Произносится как «дот-нэт»

ГЛАВА 1. C# И ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

1.1. СИНТАКСИС ОБЪЯВЛЕНИЯ КЛАССА, ПОЛЯ И МЕТОДЫ КЛАССА

Класс является основным пользовательским типом в языке C#. Синтаксис объявления класса:

```
class <имя класса>
    [<члены класса>]
```

<имя класса> – любой уникальный идентификатор,
<члены класса> объединены в программный блок.

Допустимы следующие члены класса:

1. Поле. Поля класса описываются как обычные переменные, возможно с указанием модификатора доступа. Если для поля не указан модификатор доступа, то по умолчанию подразумевается модификатор `private`. Полям класса можно задавать начальные значения.

```
class  Student {
    int age;
    private string name;
}
```

2. Константа. Объявление константы обычно используется для того, чтобы сделать текст программы более читабельным. Модификатор доступа к константам по умолчанию – `private`. Если объявлена открытая (`public` или `internal`) константа, то для ее использования вне класса можно указывать как имя объекта, так и имя класса.

3. Метод. Методы описывают функциональность класса. Код методов записывается непосредственно в теле класса. Модификатором доступа для методов по умолчанию является `private`.

4. Свойство. Свойства класса предоставляют защищенный доступ к полям. Подробнее синтаксис и применение свойств обсуждаются ниже.

5. Индексатор. Индексатор – это метод для обращения к внутренним элементам классов-коллекций, отдельный элемент которых доступен по индексу.

6. Конструктор. Задача конструктора – начальная инициализация объекта или класса. Необходимо обязательно помнить о следующем обстоятельстве: **в C#** (как и в C++), **если определен хотя бы один пользовательский конструктор (принимающий параметры), конструктор**

по умолчанию автоматически создаваться уже не будет и его необходимо определять явно.

7. Деструктор. Деструктор класса служит для уничтожения объектов класса. Так как язык C# является языком с автоматической сборкой мусора, в явном вызове деструкторов нет необходимости. Обычно они содержат некий *завершающий код* для объекта.

8. Событие. События представляют собой механизм рассылки уведомлений различным объектам.

9. Оператор. Язык C# допускает перегрузку некоторых операторов для объектов класса.

10. Вложенный пользовательский тип. Описание класса может содержать описание другого пользовательского типа – класса, структуры, интерфейса, делегата.

Переменная класса – *объект* – объявляется как обычная переменная:

```
<имя класса> <имя объекта>;
```

Так как класс – ссылочный тип, то объекты должны быть инициализированы до непосредственного использования. Для инициализации объекта используется оператор `new`, совмещенный с вызовом конструктора класса. Если конструктор не описывался, используется конструктор по умолчанию без параметров с именем класса:

```
<имя объекта> = new <имя класса>();
```

Инициализацию объекта можно совместить с его объявлением:

```
<имя класса> <имя объекта> = new <имя класса>();
```

Доступ к членам класса через объект осуществляется по синтаксису:

```
<имя объекта>.<имя члена класса>.
```

Приведем пример описания класса, который содержит два поля:

```
class Student {
    public int age;
    public string name;
}
```

Проиллюстрируем описание и использование объектов класса `Student`:

```
Student item1;                  //Объявление
Student item2 = new Student();  //Объявление с инициализацией
item1 = new Student();          //Инициализация объекта
item1.age = 10;                 //Доступ к полям
item2.name = "Alex";
```

Добавим в класс `Student` методы. Для устранения конфликта имен «имя члена класса = имя параметра метода» возможно использование ключевого слова `this` – это ссылка на текущий объект класса:

```
class Student {
    public int age;
    public string name;
    void SetAge(int age) {
        this.age = age;
    }
    string GetName() {
        return name;
    }
}
```

Связанные с объектом элементы класса называются *экземплярными*.

Статические поля, методы и свойства предназначены для работы с классом, а не с объектом. Статические поля хранят информацию, общую для всех объектов, статические методы работают со статическими полями. Для того чтобы объявить статическое поле или метод класса, используется ключевое слово `static`:

```
class CAccount {
    private static int a = 0;
    public static int GetA() {
        return a;
    }
    public static void SetA(int n) {
        a = n * 100;
}
```

Для вызова статических элементов требуется использовать имя класса:

```
CAccount.SetA(3);
Console.WriteLine(CAccount.GetA());
```

1.2. МОДИФИКАТОРЫ ДОСТУПА ДЛЯ КЛАССА

При описании класса допустимо указать для него следующие модификаторы доступа – `public` или `internal` (применяется по умолчанию). Если класс является элементом другого пользовательского типа, то его можно объявить с любым модификатором доступа.

Если класс объявлен с модификатором `internal`, то его `public`-элементы не видны за пределами сборки.

1.3. КОНСТРУКТОРЫ КЛАССА

Конструкторы класса используются для начальной инициализации объектов.

В отличие от методов конструкторы не наследуются, и вызов конструктора в виде <имя объекта>.<имя конструктора> после создания объекта запрещен.

Различают несколько видов конструкторов – конструкторы по умолчанию, пользовательские конструкторы, статические конструкторы.

Конструктор по умолчанию автоматически создается компилятором, если программист не описал в классе собственный конструктор. Конструктор по умолчанию – это всегда конструктор без параметров. Можно считать, что конструктор по умолчанию содержит код начальной инициализации полей.

Пользовательский конструктор описывается в классе как метод с именем, совпадающим с именем класса. Тип возвращаемого значения для конструктора не указывается (отсутствует любое указание на тип, даже `void`). Пользовательский конструктор может получать параметры, необходимые для инициализации объекта. Класс может содержать несколько пользовательских конструкторов, однако они обязаны различаться сигнатурой. Пользовательские конструкторы могут применяться для начальной инициализации `readonly`-полей. Такие поля ведут себя как константы, могут иметь произвольный тип. Таким образом, `readonly`-поля доступны для записи, но только в конструкторе.

Внимание! Если в классе определен хотя бы один пользовательский конструктор, конструктор по умолчанию уже не создается.

Внимание! Код, который выполняет присваивание полям значений по умолчанию, добавляется компилятором автоматически в начало любого пользовательского конструктора.

Конструктор класса может вызывать другой конструктор того же класса, но только в начале своей работы. Для этого при описании конструктора используется синтаксис, аналогичный приведенному в следующем примере:

```
public Student() : this(18, "Alex") { . . . }
```

Статические конструкторы используются для начальной инициализации статических полей класса. Статический конструктор объявляется с модификатором `static` и без параметров. Область видимости у статических конструкторов не указывается.

Статические конструкторы вызываются общеязыковой средой исполнения CLR в следующих случаях:

- перед созданием первого объекта класса или при первом обращении к элементу класса, не унаследованному от предка;
- в любое время перед первым обращением к статическому полю, не унаследованному от предка.

В теле статического конструктора возможна работа только со статическими полями и методами класса. Статические конструкторы не могут вызывать экземплярные конструкторы класса.

1.4. СВОЙСТВА И ИНДЕКСАТОРЫ КЛАССА

1.4.1. Свойства класса

Свойства класса предоставляют защищенный доступ к полям. Как и в большинстве объектно-ориентированных языков, в C# непосредственная работа с полями не приветствуется. Поля класса обычно объявляются как `private`-элементы, а для доступа к ним используются свойства.

Рассмотрим синтаксис описания свойства:

```
<тип свойства> <имя свойства> {
    get {<блок кода>}
    set {<блок кода>}
}
```

Тип свойства обычно совпадает с типом того поля, для обслуживания которого свойство создается. У свойства присутствует специальный блок, содержащий методы для доступа к свойству. Данный блок состоит из `get`-части и `set`-части. Одна из частей может отсутствовать, так получается *свойство только для чтения* или *свойство только для записи*. `Get`-часть отвечает за возвращаемое свойством значение и работает как функция (обязательно наличие в блоке кода `get`-части оператора `return`). `Set`-часть работает как метод-процедура, устанавливающий значение свойства. Параметр, передаваемый в `set`-часть, имеет специальное имя `value`.

Добавим свойства в класс `Student`, закрыв для использования поля:

```
class Student {
    private int age;
    private string name;
    public int Age {
        get {
            return age;
        }
        set {
```

```
            // проверка корректности значения
            age = value < 0 ? age = 0 : age = value;
        }
    }
    public string Name {
        get {
            return "My name is " + name;
        }
        set { name = value; }
    }
}
```

Свойства транслируются при компиляции в вызовы методов. В скомпилированный код класса добавляются методы со специальными именами `get_Name` и `set_Name`, где *Name* – это имя свойства. Пользовательские методы с данными именами допустимы в классе, только если они имеют сигнатуру, отличающуюся от методов, соответствующих свойству.

Обращаться к объекту `value` можно только в пределах программного блока `set` внутри определения свойства. Попытка обратиться к этому объекту из любого другого места приведет к ошибке компилятора.

C# также поддерживает статические свойства. **Если объявлены статические данные, то обращаться к ним и устанавливать значения должны статические свойства.**

Главное преимущество свойств заключается в том, что пользователь может работать через них с внутренними данными, используя единственное имя (вместо двух разных имен методов).

1.4.2. Индексаторы класса

При помощи индексаторов осуществляется доступ к коллекции данных, содержащихся в объекте класса, с использованием привычного синтаксиса для доступа к элементам массива – пары квадратных скобок. Метод, который обеспечивает такую возможность, получил название индексатора.

Объявление индексатора:

```
<тип индексатора> this[<аргументы>] { <get и set блоки> }
```

Аргументы индексатора служат для описания типа и имен индексов, применяемых для доступа к данным объекта. Аргументы индексатора доступны в блоках `get` и `set`. Если индексатор имеет более одного аргумента, то аргументы перечисляются через запятую.

Рассмотрим пример класса, содержащего индексаторы. Пусть данный класс описывает студента с набором оценок:

```
class Student {
    private int[] marks = new int[5];
    private string name;

    public int this[int i] {
        get {
            if ((i >= 1) && (i <= 5)) return marks[i-1];
            else return 0;
        }
        set {
            if ((i >= 1) && (i <= 5) && (value <= 10))
                marks[i-1] = value;
        }
    }
}
```

Данный класс и индексатор можно использовать следующим образом:

```
Student S = new Student();
S[1] = 8;
S[3] = 4;
for(int i = 1; i <= 5; i++)
    Console.WriteLine(S[i]);
```

Индексаторы всегда работают как *свойства по умолчанию*. Это значит, что в одном классе нельзя объявить два индексатора, у которых совпадают типы аргументов. Однако можно объявить в одном классе индексаторы, у которых аргументы имеют разный тип или количество аргументов различается.

Индексаторы транслируются компилятором в методы с именами `get_Item` и `set_Item`.

1.5. ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ (ООП)

В любом объектно-ориентированном языке программирования реализованы три важнейших принципа:

- **инкапсуляция**: как объекты прячут свое внутреннее устройство;
- **наследование**: как поддерживается повторное использование кода;
- **полиморфизм**: как реализована поддержка выполнения нужного действия в зависимости от типа передаваемого объекта.

1.5.1. Инкапсуляция

Инкапсуляция - способность прятать детали реализации объектов от пользователей этих объектов.

В идеале все внутренние переменные члены класса должны быть определены как `private`.

Принцип инкапсуляции предполагает, что ко внутренним данным объекта (переменным-членам) нельзя обратиться напрямую через экземпляр этого объекта. Вместо этого для получения информации о внутреннем состоянии объекта и внесения изменений необходимо использовать специальные методы.

1.5.2. Средства инкапсуляции в C#

Для обращения к внутренним данным можно использовать один из двух способов:

- создать традиционную пару методов — один для получения информации (`get-method`), второй — для внесения изменений (`set-method`);
- определить именованное свойство.

Еще один метод защиты данных, предлагаемый C#, — использовать ключевое слово `readonly`, чтобы запретить любую возможность изменять значение поля. Любая попытка изменить значение такого поля приведет к ошибке компилятора.

1.5.3. Статические поля «только для чтения»

Обычно используются в тех ситуациях, когда необходимо создать некоторое количество постоянных значений, связанных с определенным классом. Очень похожие задачи выполняют обычные константы. Однако между полями и константами есть существенное различие: константа заменяется на свое значение уже в процессе компиляции, в то время как значения статических полей только для чтения вычисляются лишь в процессе выполнения программы.

```
public static readonly int age;
```

1.5.4. Наследование

Язык C# полностью поддерживает объектно-ориентированную концепцию наследования.

Чтобы указать, что один класс является наследником другого, используется следующий синтаксис:

```
class <имя наследника> : <имя базового класса>
 {<тело класса>}
```

На вершине любой иерархии в .NET всегда находится базовый класс `Object`.

Наследник обладает всеми полями, методами и свойствами предка, однако элементы предка с модификатором `private` не доступны в наследнике. Конструкторы класса-предка не переносятся в класс-наследник.

При наследовании нельзя расширить область видимости класса: `internal`–класс может наследоваться от `public`–класса, но не наоборот.

Для обращения к методам непосредственного предка класс-наследник может использовать ключевое слово `base` в форме

```
base.<имя метода базового класса>
```

1.5.4.1. Работа с конструктором базового класса

Если конструктор наследника должен вызвать конструктор предка, то для этого также используется `base`:

```
<конструктор наследника>([<параметры>]): base([<параметры_2>])
```

Для конструкторов производного класса справедливо следующее замечание: конструктор должен вначале совершить вызов конструктора базового класса (если иное не указано специально, будет вызван конструктор по умолчанию), который присвоит всем данным, за которые он «ответствен», безопасные значения по умолчанию. Если вызов конструктора базового класса отсутствует, компилятор автоматически подставляет в заголовок конструктора вызов :`base`(). Только после этого конструктор производного класса заново определит значения для унаследованных членов.

1.5.4.2. Наследование от нескольких базовых классов

Наследование от двух и более классов в C# запрещено.

1.5.4.3. Модели наследования

Когда классы связываются друг с другом отношениями наследования, это означает, что между ними устанавливаются отношения типа «**быть**» (is-a). Такой тип отношений называется также **классическим наследованием**.

Основная идея классического наследования заключается в том, что производные классы должны получать функциональность от базового класса-предка и дополнять ее новыми возможностями.

В объектно-ориентированном программировании используется еще одна форма повторного использования кода. Эта форма называется **включением-делегированием** (или отношением «**иметь**»— has-a). При

ее использовании один класс включает в свой состав другой и открывает внешнему миру часть возможностей внутреннего класса.

Например, если при создании программной модели автомобиля, появляется идея включить внутрь объекта «автомобиль» объект «радио» с помощью отношения «иметь».

```
using System;
namespace Ex {
  class Car {
      class Radio {
          private bool RadioOnOff;
          public void Power(bool on) {
              RadioOnOff = on;
              Console.WriteLine("Radio is {0}", RadioOnOff);
          }
      }
      Radio RadioObj;
      // Метод объекта Car (автомобиль)
      void TurnOnRadio(bool on) {
          RadioObj = new Radio();
           // Делегируется внутреннему классу Radio (радио)
          RadioObj.Power(on);
      }
      static void Main(string[] args) {
           //Внутренний класс Radio инкапсулирован внешним классом Car
          Car myCar = new Car();
          myCar.TurnOnRadio(true); //Вызов будет передан
                                   //внутреннему объекту Radio
      }
  }
}
```

В терминологии объектно-ориентированного программирования контейнерный класс (в нашем случае `Car`) называется родительским (`parent`), а внутренний класс, который помещен внутрь контейнерного (`Radio`), называется дочерним (`child`).

Таким образом, радиоприемник внутри автомобиля у нас теперь создается вместе с автомобилем. Чтобы воспользоваться возможностями внутреннего класса, необходимо *делегирование*. Делегирование заключается в добавлении во внешний контейнерный класс методов для обращения к внутреннему классу.

1.5.5. Определение вложенных типов

В C# можно определить тип непосредственно внутри другого типа. Такие типы называются вложенными:

```
// В C# можно вкладывать друг в друга классы, интерфейсы
// и структуры
public class MyClass {
    // Члены внешнего класса
    public class MyNestedClass {
        // Члены внутреннего класса
    }
}
```

В C# вложенные классы могут объявляться и как `private`, и как `public`.

Классы, которые объявлены в пространстве имен напрямую (то есть те классы, которые не вложены ни в какой другой класс), не могут быть объявлены как `private`.

1.5.6. Поддержка наследования в C#

Класс-наследник может дополнять базовый класс новыми методами, а также замещать методы базового класса. Для замещения достаточно указать в новом классе метод с прежним именем:

```
class People {
    public void Speak() {
        Console.WriteLine("I'm a people");
    }
}
class Student : People {
    public void Speak() {
         Console.WriteLine("I'm a student");
     }
}
. . .
People p = new People();
Student s = new Student();
p.Speak();
s.Speak();
```

При компиляции данного фрагмента будет получено предупреждающее сообщение о том, что метод `s.Speak()` закрывает метод базового класса `p.Speak()`.

1.5.6.1. Контроль версий членов класса

Чтобы подчеркнуть, что метод класса-наследника замещает метод базового класса, используется ключевое слово `new`:

```
class Student : People {
    new public void Speak() {
        Console.WriteLine("I'm a student");
    }
}
```

Ключевое слово `new` может размещаться как до, так и после модификаторов доступа для метода. Данное ключевое слово применимо и к полям класса.

Представленный прием работы с методами в производных классах можно противопоставить замещению методов. Этот прием называется сокрытием методов (`method hiding`). Рассмотрим его на примере.

Рассмотрим иерархию геометрических фигур на рис. 1.1.

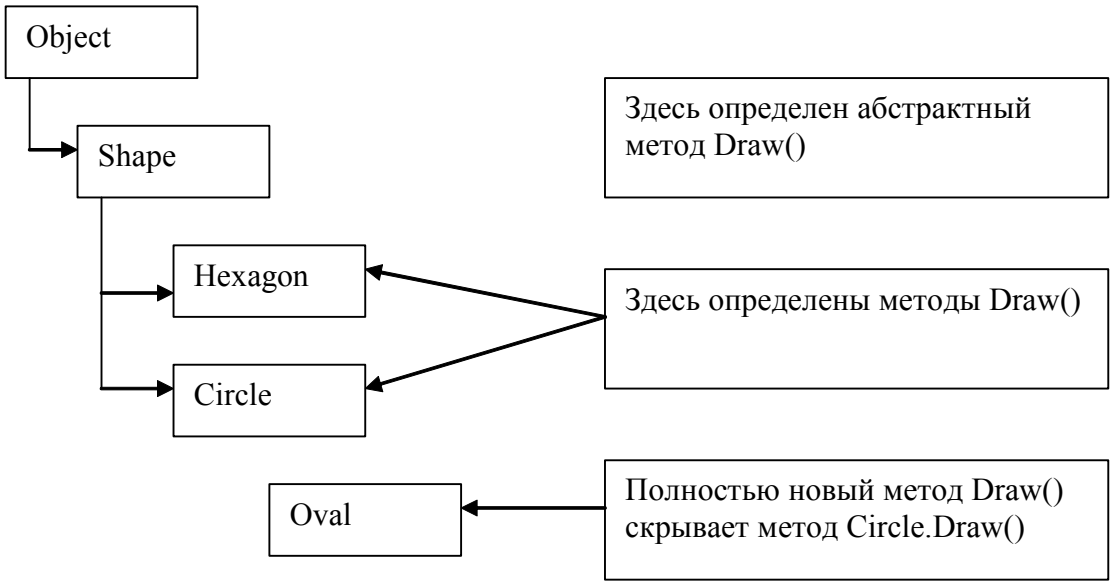

Рис. 1.1. Иерархия геометрических фигур. Контроль версий членов класса.

Предположим, что в классе `Oval` определен метод `Draw()`. Однако представим, что, в отличие от отношений между реализациями метода `Draw()` в базовом классе необходимо запретить любое наследование логики `Draw()` между методами. C# предлагает для этого случая средство, которое называется *контролем версий*. Для того чтобы им воспользоваться, достаточно определить в классе `Oval` метод `Draw()` с ключевым словом **new:**

```
public class Oval : Circle {
   public Oval() {…}
   // Скрываем любые реализации Draw() базовых классов
```

```
    new public void Draw() {
        // Специфичный для Oval алгоритм Draw()
    }
}
```

Поскольку в классе `Oval` для метода `Draw()` использовано ключевое слово `new`, гарантируется, что при создании объекта класса `Oval` и вызове для него метода `Draw()` будет вызвана именно та реализация этого метода, которая определена в классе `Oval`. Ключевое слово `new` разрывает отношения между абстрактным методом `Draw()`, определенным в базовом классе `Shape`, и методом `Draw()` в производном классе `Oval`.

```
// Будет вызван метод Draw(), определенный в классе Oval
Oval o = new Oval();
o.Draw();
```

Однако, если потребуется вызвать вариант метода для базового класса, это тоже можно сделать с помощью явного приведения типов:

```
// Будет вызван метод Draw(), определенный в классе Circle
((Circle)o).Draw();      // Приводим к базовому классу
или
(o as Circle).Draw();    // Приводим к базовому классу
```

1.5.7. Полиморфизм

Этот термин определяет возможности, заложенные в языке, по интерпретации связанных объектов одинаковым образом.

Существует две основные разновидности полиморфизма: **классический полиморфизм и полиморфизм «для конкретного случая»**.

Классический полиморфизм встречается только в тех языках, которые поддерживают классическое наследование. При классическом полиморфизме можно определить в базовом классе набор членов, которые могут быть замещены в производном классе.

Для примера обратимся к иерархии геометрических фигур. Предположим, что в классе `Shape` (геометрическая фигура) определена функция `Draw()` — рисование. Поскольку геометрические фигуры бывают разными и каждый тип фигуры потребуется изображать своим собственным способом, потребуется в производных классах (таких как `Hexagon` — шестиугольник и `Circle` — окружность) создать свой собственный метод `Draw()`, заместив им метод `Draw()` базового класса (рис. 1.2).

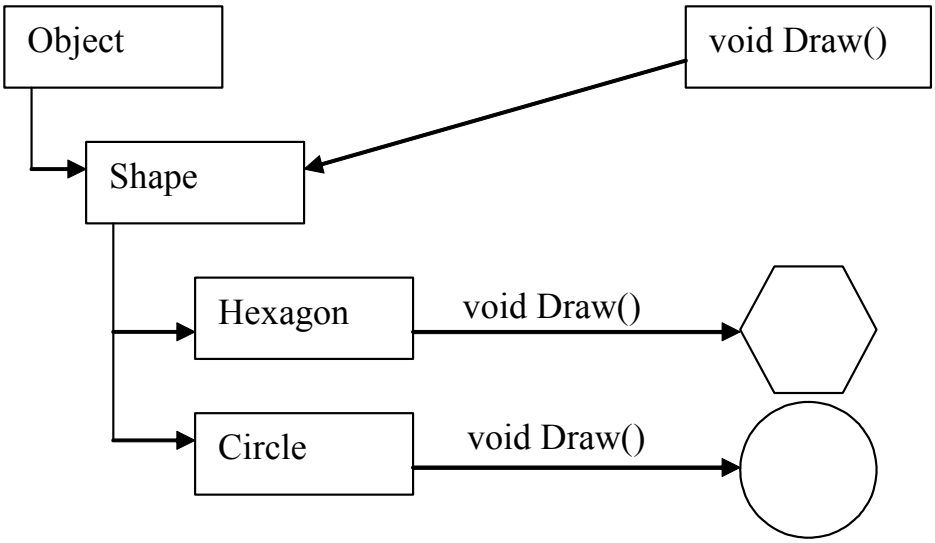

Рис. 1.2. Классический полиморфизм

Классический полиморфизм позволяет определять возможности всех производных классов при создании базового класса.

Вторая разновидность полиморфизма — *полиморфизм для конкретного случая*. Этот тип полиморфизма позволяет обращаться схожим образом к объектам, *не связанным классическим наследованием.* Достигается это следующим образом: в каждом из таких объектов должен быть метод с одинаковой сигнатурой (то есть одинаковым именем метода, принимаемыми параметрами и типом возвращаемого значения). В языках, поддерживающих полиморфизм этого типа, применяется технология «позднего связывания» (`late binding`), когда тип объекта, к которому происходит обращение, становится ясен только в процессе выполнения программы. В зависимости от того, к какому типу происходит обращение, вызывается нужный метод. В качестве примера рассмотрим схему на рис. 1.3.

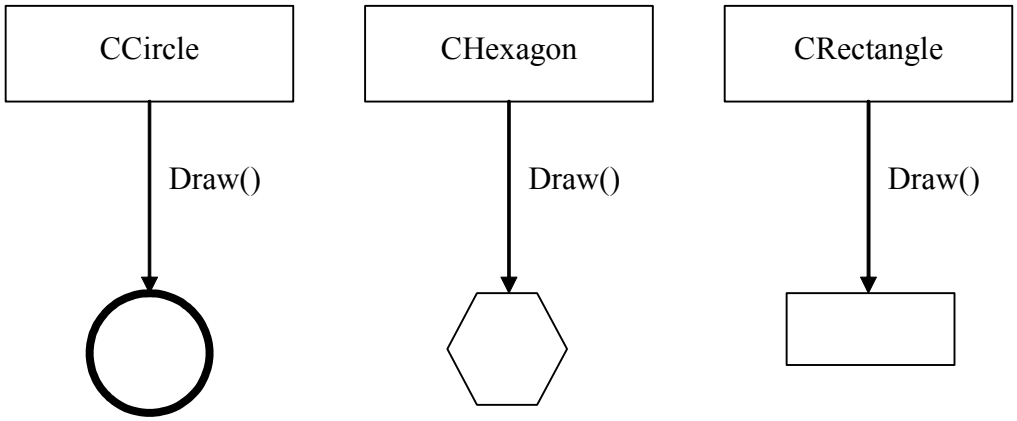

Рис. 1.3. Полиморфизм для конкретного случая

Общего предка — базового класса для `CCircle`, `CHexagon` и `CRectangle` нет. Однако в каждом классе предусмотрен метод `Draw()` с одинаковой сигнатурой.

```
using System;
namespace Ex {
  public class CCircle {
      public void Draw(string s) {
          Console.WriteLine(s + " CCircle");
      }
  }
  public class CHexagon {
      public void Draw(string s) {
          Console.WriteLine(s + " CHexagon");
      }
  }
  public class CRectangle {
      public void Draw(string s) {
          Console.WriteLine(s + " CRectangle");
      }
  }
  class Program {
    static void Main(string[] args) {
        // Вначале создадим массив элементов типа Object
        // и установим для каждого элемента ссылку на
        // объект
        System.Object[] objArr = new object[3];
        objArr[0] = new CCircle();
        objArr[1] = new CHexagon();
        objArr[2] = new CRectangle();
        // Теперь с помощью цикла каждый элемент
        // нарисует самого себя
        for (int i = 0; i < objArr.Length; i++)
           if (objArr[i] is CCircle)
             (objArr[i] as CCircle).Draw("Позднее связывание для");
           else
             if (objArr[i] is CHexagon)
               (objArr[i] as CHexagon).Draw("Позднее связывание для");
             else
               if (objArr[i] is CRectangle)
                       (objArr[i] as CRectangle).Draw("Позднее
                                                 связывание для");
      }
  }
}
```

Итак, в C# реализованы все принципы ООП, при этом C# поддерживает и отношения «быть» и отношения «иметь» для повторного использования кода, и обе разновидности полиморфизма.

1.5.8. Модификаторы sealed и abstract

Для классов можно указать два модификатора, связанных с наследованием.

Модификатор `sealed` задает класс, от которого запрещено наследование. Чаще всего ключевое слово `sealed` применяется для служебных классов. Например, таким образом объявлен класс `String` пространства имен `System`.

Модификатор `abstract` задает *абстрактный класс*, у которого обязательно должны быть наследники. Объект абстрактного класса создать нельзя, хотя статические члены такого класса можно вызвать, используя имя класса. Модификаторы наследования указываются непосредственно перед ключевым словом `class`:

```
sealed class FinishedClass { }
abstract class AbstractClass { }
```

1.5.9. Поддержка полиморфизма в C#

Замещение методов класса не является полиморфным по умолчанию. Следующий фрагмент кода печатает две одинаковые строки:

```
People p, s;
p = new People();
s = new Student();  // Допустимо по правилам присваивания
p.Speak();          // Печатает "I'm a people"
s.Speak();          // Так же печатает "I'm a people"
```

Для организации полиморфного вызова методов применяется пара ключевых слов `virtual` и `override`: `virtual` указывается для метода базового класса, который необходимо сделать полиморфным, `override` – для методов производных классов. Эти методы должны совпадать по имени и сигнатуре с перекрываемым методом класса-предка.

```
class People {
    public virtual void Speak() {
        Console.WriteLine("I'm a people");
    }
}
class Student : People {
    public override void Speak() {
        Console.WriteLine("I'm a student");
    }
}
. . .
People p, s;
p = new People();
s = new Student();
p.Speak();           // Печатает "I'm a people"
s.Speak();           // Теперь печатает "I'm a student"
```

Если на некоторой стадии построения иерархии классов требуется запретить дальнейшее переопределение виртуального метода в производных классах, этот метод помечается ключевым словом `sealed`:

```
class Student : People {
public sealed override void Speak() { ... }
```

Наряду с виртуальными методами в C# можно описать виртуальные свойства.

Виртуальные методы замещать не обязательно. При этом в случае вызова метода для объекта производного класса будет вызван уникальный вариант этого метода для базового класса.

1.5.10. Абстрактные классы

Для методов абстрактных классов (классов с модификатором `abstract`) возможно задать модификатор `abstract`, который говорит о том, что метод не реализуется в классе, а должен обязательно переопределяться в наследнике.

```
abstract class AbstractClass {
    //Реализации метода в классе нет
    public abstract void AbstractMethod();
}
```

Теперь при попытке создания объекта класса `AbstractClass` компилятор будет выдавать сообщение об ошибке.

1.5.11. Абстрактные методы

Абстрактные методы — это аналоги чистых виртуальных функций C++: они позволяют определить в базовом классе методы без реализации по умолчанию.

Все абстрактные методы обязательно должны быть замещены в производных классах.

Объекты абстрактных классов создать невозможно.

1.5.12. Приведение типов в C#

C# позволяет приводить (*cast*) один тип к другому, то есть осуществлять преобразование объектов одного типа в объекты другого типа.

Первый закон приведения типов звучит так: если один класс является производным от другого, всегда безопасно ссылаться на объект производного класса через объект базового класса.

В соответствии с правилами приведения типов можно передавать методу объект как самого базового типа, так и любого производного типа.

ГЛАВА 2. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ КЛАССОВ C#

2.1. ПЕРЕГРУЗКА ОПЕРАТОРОВ

Язык C# позволяет организовать для объектов пользовательского класса или структуры *перегрузку операторов*. Могут быть перегружены унарные операторы `+, -, !, ~, ++, --, true, false` и бинарные операторы `+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=`.

При перегрузке бинарного оператора автоматически перегружается соответствующий оператор с присваиванием (например, при перегрузке оператора + перегрузится и оператор +=). Некоторые операторы могут быть перегружены только парами: `==` и `!=`, `>` и `<`, `>=` и `<=`, `true` и `false`.

Для перегрузки операторов используется специальный статический метод, имя которого образовано из ключевого слова `operator` и знака операции. Количество формальных параметров метода зависит от типа операции: унарная операция требует одного параметра, бинарная – двух.

Метод обязательно должен иметь модификатор доступа `public`.

Ключевое слово `operator` можно использовать только вместе с ключевым словом `static`.

Следует помнить, что подавляющее большинство языков программирования перегрузку операторов не поддерживают. Требования обеспечить возможность перегрузки операторов нет и в `Common Language Specification`. Поэтому если идет работа с большим проектом, разные модули которого созданы на разных языках, нужно быть готовыми столкнуться с тем, что в некоторых модулях использовать перегрузку операторов будет невозможно.

Поэтому при создании пользовательских классов, предназначенных для работы в реальных рабочих приложениях, хорошо бы снабжать перегруженные операторы обычными методами-аналогами. В модулях, написанных на языках, поддерживающих перегрузку операторов, будут использоваться соответствующие операторы, а в остальных модулях — обычные методы со стандартными названиями.

Рассмотрим перегрузку операторов на примере. Определим класс для представления комплексных чисел с перегруженным оператором сложения:

```
class Complex {
    public double Re;
    public double Im;
```

```
    public Complex(double Re, double Im) {
        this.Re = Re;
        this.Im = Im;
    }
    public override string ToString() {
        return String.Format("Re = {0} Im = {1}", Re, Im);
    }

    public static Complex operator + (Complex A, Complex B) {
        return new Complex(A.Re + B.Re, A.Im + B.Im);
    }
}
```

Для объектов класса `Complex` возможно использование следующего кода:

```
Complex A = new Complex(10.0, 20.0);
Complex B = new Complex(-5.0, 10.0);
Console.WriteLine(A + B);       // Выводит Re =  5.0 Im = 30.0
```

Примечание. Не допускается существование двух версий метода перегрузки оператора, различающихся только типом возвращаемого значения. Также класс не может содержать перегруженного оператора, у которой ни один из формальных параметров не имеет типа класса.

Внесем некоторые изменения в класс `Complex`:

```
class Complex {
. . .
    public static Complex operator + (Complex A, Complex B) {
        return new Complex(A.Re + B.Re, A.Im + B.Im);
    }
    public static Complex operator + (Complex A, double B) {
        return new Complex(A.Re + B, A.Im + B);
    }
}
```

Новый перегруженный оператор сложения позволяет прибавлять к комплексному числу вещественное число.

Любой класс может перегрузить операторы `true` и `false`. Операторы перегружаются парой, тип возвращаемого значения операторов – `bool`. Если в классе выполнена подобная перегрузка, объекты класса могут использоваться как условия в операторах условного перехода или циклов.

Рассмотрим пример. Пусть в классе `Complex` перегружены операторы `true` и `false`

```
class Complex {
. . .
    public static bool operator true (Complex A) {
        return (A.Re > 0) || (A.Im > 0);
    }
```

```
    public static bool operator false (Complex A) {
        return (A.Re == 0) && (A.Im == 0);
    }
}
```

Теперь возможно написать такой код:

```
Complex A = new Complex(10.0, 20.0);
Complex B = new Complex(0, 0);
if (B)
    Console.WriteLine("Number is not zero");
else
    Console.WriteLine("Number is 0.0 + 0.0i");
```

Кроме перечисленных операторов, любой класс может перегрузить операторы для неявного и явного приведения типов. При этом используется следующий синтаксис:

```
public static implicit operator <целевой тип>(<привод. тип> <имя>)
public static explicit operator <целевой тип>(<привод. тип> <имя>)
```

Ключевое слово `implicit` используется при перегрузке неявного приведения типов, а ключевое слово `explicit` – при перегрузке операции явного приведения. Либо `<целевой тип>`, либо `<приводимый тип>` должен совпадать с типом того класса, в котором выполняется перегрузка операторов.

Поместим перегруженные операторы приведения в класс `Complex`:

```
class Complex {
. . .
    public static implicit operator Complex (double a) {
        return new Complex(a, 0);
    }
    public static explicit operator double (Complex A) {
        return Math.Sqrt(A.Re * A.Re + A.Im * A.Im);
    }
}
```

Вот пример кода, использующего преобразование типов:

```
Complex A = new Complex(3.0, 4.0);
double x;
//Выполняем явное приведение типов
x = (double) A;
Console.WriteLine(x);       //Выводит 5
double y = 10;
//Выполняем неявное приведение типов
A = y;
Console.WriteLine(A);       //Выводит Re = 10 Im = 0
```

2.2. ПЕРЕГРУЗКА ОПЕРАТОРОВ РАВЕНСТВА

В .C# часто возникает необходимость замещать метод `System.Object.Equals()`, чтобы можно было выполнять сравнения структурных типов (в исходной реализации этот метод поддерживает только сравнение ссылочных типов)

Помимо замещения методов `Equals` и `GetHashCode()` (при замещении `Equals` замещение `GetHashCode()` обязательно) скорее всего, потребуется заместить также и операторы равенства (== и !=).

```
public class Point {
    private int x, y;
    public Point() { }
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public override string ToString() {
        return "X = " + this.x + " Y = " + this.y;
    }
    public override bool Equals(object o) {
        if ((o as Point).x == this.x && (o as Point).y == this.y)
            return true;
        else
            return false;
    }
    public override int GetHashCode() {
        return this.ToString().GetHashCode(); }
    // А вот и сама перегрузка операторов равенства
    public static bool operator ==(Point p1, Point p2) {
        return p1.Equals(p2);
    }
    public static bool operator !=(Point p1, Point p2) {
        return !p1.Equals(p2);
    }
}
```

Передаем выполнение всей необходимой работы замещенному методу `Equals`.

```
static void Main(string[] args) {
     // Задаем две точки
     Point ptOne = new Point(40, 40);
     Point ptTwo = new Point(40, 40);
     // Применяем перегруженные операторы равенства
     if (ptOne == ptTwo) // Две точки совпадают?
         Console.WriteLine("Same values! ");
     else
         Console.WriteLine("Different values.");
}
```

C# не позволяет перегрузить оператор = = без перегрузки оператора ! = или наоборот (точно так же, как методы `Equals()` и `GetHashCode()` можно перегружать только в паре). Такой подход гарантирует, что все операторы равенства будут применяться одинаково корректно.

2.3. ГЕНЕРАЦИЯ И ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

2.3.1. Обработка исключений

.NET предлагает единую технику для обнаружения ошибок времени выполнения и передачи сообщений о них — структурированную обработку исключений (*Structured Exception Handling, SEH*).

Все системные и пользовательские исключения в C# производятся от класса `System.Exception` (который, в свою очередь, производится от класса `System.Object`).

Класс `System.Exception` является стандартным классом для представления исключительных ситуаций.

В табл. 2.1 представлен перечень наиболее интересных свойств класса `Exception`.

Таблица 2.1. ***Некоторые свойства класса*** **`System.Exception`**

Свойство	Назначение
`HelpLink`	Возвращает URL файла справки с описанием ошибки
`Message`	Возвращает текстовое описание ошибки
`Source`	Возвращает имя объекта или приложения, которое сгенерировало ошибку
`StackTrace`	Возвращает последовательность вызовов, которые привели к возникновению ошибки

2.3.2. Генерация исключения

Для генерации исключительной ситуации используется команда `throw` со следующим синтаксисом:

```
throw <объект класса исключительной ситуации>;
```

Объект, указанный после `throw`, должен обязательно быть объектом класса исключительной ситуации.

Рассмотрим пример программы с генерацией исключительной ситуации:

```
using System;
class CException {
    private int fX;
    public void setFx(int x) {
        if (x > 0)
            fX = x;
```

```
            else
                throw new Exception();
        }
    }
    class Program {
        public static void Main() {
            CException A = new CException();
            A.setFx(-3); // ИС генерируется, но не обрабатывается!
        }
    }
```

Так как в данном примере исключительная ситуация генерируется, но никак не обрабатывается, при работе приложения появится стандартное окно с сообщением об ошибке.

2.3.3. Создание пользовательских исключений

Создание собственного класса-исключения:

1) определить новый класс, производный от `System.Exception` (по умолчанию для классов — пользовательских исключений используется суффикс `Exception`).
2) определить все необходимые свойства, методы и поля, которые будут использоваться внутри блока `catch`.
3) заместить виртуальные члены, определенные в базовом классе `System.Exception`.

Модифицируем пример с генерацией исключительной ситуации, описав для исключительной ситуации собственный класс:

```
class MyException : Exception {
    public int info;
}
class CException {
    private int fX;
    public void setFx(int x) {
        if (x > 0)
            fX = x;
        else {
            MyException E = new MyException();
            E.info = x;
            throw E;
        }
    }
}
```

2.3.4. Перехват исключений

Для перехвата исключительных ситуаций служит блок `try - catch - finally`. Синтаксис блока следующий:

```
try {
    [<команды, способные вызвать исключительную ситуацию>]
}
[<один или несколько блоков catch>]
[finally {
    <операторы из секции завершения>
}]
```

Операторы из части `finally` (если она присутствует) выполняются всегда, вне зависимости от того, произошла исключительная ситуация или нет. Если один из операторов, расположенных в блоке `try`, вызвал исключительную ситуацию, управление немедленно передается на блоки `catch`. Синтаксис отдельного блока `catch` следующий:

```
catch [(<тип ИС> [<идентификатор объекта ИС>])] {
    <команды обработки исключительной ситуации>
}
```

`<идентификатор объекта ИС>` – это некая временная переменная, которая может использоваться для извлечения информации из объекта исключительной ситуации.

Модифицируем программу, описанную выше, добавив в нее блок перехвата ошибки:

```
class Program {
  public static void Main() {
    CException A = new CException();
    try {
      Console.WriteLine("Эта строка печатается");
      A.setFx(-3);
      Console.WriteLine("Строка не печатается, если ошибка ");
    }
    catch (MyException ex) {
      Console.WriteLine("Ошибка при параметре {0}", ex.info);
    }
    finally {
      Console.WriteLine("Строка печатается - блок finally");
    }
  }
}
```

Если используется несколько блоков `catch`, то обработка исключительных ситуаций происходит по принципу «от частного – к общему», так как после выполнения одного блока `catch` управление передается на часть `finally` (при отсутствии `finally` – на оператор после `try` -

catch). Компилятор C# не позволяет разместить блоки catch так, чтобы предыдущий блок перехватывал исключительные ситуации, предназначенные последующим блокам:

```
try { … }
//Ошибка компиляции, так как MyException – наследник Exception
catch (Exception ex) {
    Console.WriteLine("Общий перехват");
}
catch (MyException ex) {
    Console.WriteLine("Эта строка не печатается никогда!");
}
```

2.3.5. Обработка нескольких исключений

В простой форме одному блоку try соответствует один блок catch. Однако в реальных проектах часто возникают ситуации, когда необходимо отслеживать возникновение не одного, а нескольких исключений. Код для вызова исключений может выглядеть следующим образом:

```
try { }
catch(MyException e) {
    Console.WriteLine(e.Message);
}
catch(ArgumentOutOfRangeException e) {
    Console.WriteLine(e.Message);
}
```

2.3.6. Блок finally

После блока try/catch в C# может следовать необязательный блок finally . Этот блок выполняется всегда, вне зависимости от того, сработало исключение или нет. Его главное назначение — гарантировать, что ресурсы, которые могут быть открыты потенциально опасным методом, будут обязательно освобождены.

```
try  { }
catch (MyException e) { }
catch (ArgumentOutOfRangeException e) { }
finally {
     //Этот блок будет выполнен всегда, вне зависимости
     //от того,произошла ошибка или нет
}
```

В реальных проектах блок finally используется для освобождения памяти, закрытия файла, отключения от источника и данных и выполнения прочих операций, связанных с корректным завершением программы.

2.4. ЖИЗНЕННЫЙ ЦИКЛ ОБЪЕКТОВ

Все пользовательские типы в языке C# можно разделить на структурные и ссылочные.

Переменные структурных типов создаются средой исполнения CLR в стеке. *Время жизни* (*lifetime*) переменных структурного типа ограничено тем блоком кода, в котором они объявляются.

Переменные ссылочного типа (объекты) размещаются в динамической памяти – «куче». Среда исполнения платформы .NET использует *управляемую кучу* (*managed heap*). Если при работе программы превышен некий порог расходования памяти, CLR запускает процесс, называемый *сборка мусора*. Среда исполнения отслеживает все используемые объекты и определяет реально занимаемую этими объектами память. После этого вся оставшаяся память освобождается, то есть помечается как свободная для использования. Освобождая память, CLR заново размещает «уцелевшие» объекты в куче, чтобы уменьшить ее фрагментацию. Ключевой особенностью сборки мусора является то, что она осуществляется средой исполнения автоматически и независимо от основного потока выполнения приложения.

В ситуации, когда свободного места в управляемой куче больше нет, а происходит попытка создать новый объект, будет сгенерировано исключение `OutOfMemoryException`. Поэтому, если необходимо создать код приложения, совершенно исключив возможность возникновения ошибок, создавать объекты можно следующим образом:

```
// Создаем обьекты Car таким образом, чтобы отреагировать на
// возможную нехватку места в управляемой куче
Car aCar;
try {
   aCar = new Car();
}
catch (OutOfMemoryException e) {
   Console.WriteLine(e.Message);
   Console.WriteLine("Managed heap is FULL! Running GC...");
}
```

Как только место в управляемой куче заканчивается, автоматически запускается сборщик мусора (*garbage collector, GC*). Сборщик мусора оценивает все объекты, размещенные в настоящий момент в управляемой куче, с точки зрения того, есть ли в области видимости приложения активные ссылки на них. Если активных ссылок на какой-либо объект больше нет или объект установлен в NULL, этот объект помечается для

удаления, и в скором времени память, занимаемая подобными объектами, освобождается.

2.4.1. Завершение ссылки на объект

Если нужно обеспечить возможность удаления объектов из оперативной памяти в соответствии с определенными правилами, первое, о чем необходимо позаботиться — о реализации в классе виртуального метода `System.Object.Finalize()`. В C# запрещено напрямую замещать метод `Object.Finalize()`. Более того, нельзя даже вызвать в приложении этот метод напрямую. Если необходимо, чтобы пользовательский класс поддерживал метод `Finalize()`, нужно использовать в определении этого класса метод, синтаксис которого напоминает синтаксис деструктора языка C++. Имя метода образовывается по правилу `~ <имя класса>`, метод не имеет параметров и модификаторов доступа.

```
class Car: Object {
    ~Car() {
    //Закрываем все открытые объектом ресурсы!
    //Далее в C# будет автоматически вызван метод base.Finalize()
    }
 }
```

При размещении объекта C# в управляемой куче при помощи оператора `new` среда выполнения автоматически определяет, поддерживает ли объект метод `Finalize()` (представленный в C# с помощью «деструктороподобного» синтаксиса). Если этот метод поддерживается объектом, ссылка на этот объект помечается как «завершаемая» (*finalizable*). При этом в специальной внутренней очереди «завершения» (*finalization queue*) помещается указатель на данный объект. Когда сборщик мусора приходит к выводу, что наступило время удалять данный объект из оперативной памяти, он обращается к этому указателю и запускает деструктор C#, определенный для этого класса, прежде чем будет произведено физическое удаление объекта из памяти.

2.4.2. Создание метода удаления для конкретного случая. Интерфейс IDisposable

Проблема с использованием метода-«деструктора» заключается в том, что момент вызова этого метода сложно отследить. Программист может описать в классе некий метод, который *следует* вызывать «вручную», когда объект больше не нужен. Для унификации данного решения платформа .NET предлагает интерфейс `IDisposable`, содержащий единственный метод `Dispose()` (освободить), куда помещается завершаю-

щий код работы с объектом. Класс, объекты которого требуется освобождать «вручную», реализовывает интерфейс `IDisposable`.

Интерфейс IDisposable содержит единственный метод `Dispose()`:

```
public interface IDisposable {
    public void Dispose();
}
```

Применить этот интерфейс в случае класса `Car` можно следующим образом:

```
// Реализуем IDIsposable
class Car: Object, IDisposable {
    public void Dispose() {
        // ...Закрываем открытые внутренние ресурсы
    }
 }
```

Таким образом, можно гарантировать освобождение ресурсов без помещения указателя на деструктор в очередь завершения. Кроме того, в этом случае освобождение ресурсов будет произведено немедленно, а не тогда, когда у сборщика мусора «дойдут руки» до нашего объекта.

C# имеет специальную *обрамляющую конструкцию* `using`, которая *гарантирует* вызов метода `Dispose()` для объектов, использующихся в своем блоке. Синтаксис данной конструкции:

```
using (<имя объекта или объявление и создание объектов>)
    <программный блок>
```

Поместим в класс `Program` обрамляющую конструкцию `using`:

```
class Program {
    public static void Main() {
        using(Car A = new Car()) {
            A.Метод;
            // компилятор поместит сюда вызов A.Dispose()
        }
    }
}
```

2.4.3. Взаимодействие со сборщиком мусора

Сборщик мусора — это объект, и можно обращаться к нему через ссылку на объект. Для работы со сборщиком мусора в C# предназначен специальный класс — `System.GC` (от *garbage collector* — сборщик мусора). Этот класс определен как `sealed`, то есть производить от него другие классы при помощи наследования невозможно. В `System.GC` определен набор статических свойств и методов, при помощи которых и осуществляется взаимодействие со сборщиком мусора.

Таблица 2.2. ***Некоторые методы и свойства класса*** **System.GC**

Метод, свойство	Назначение
`Collect()`	Запускает работу сборщика мусора для всех поколений. Можно указать в качестве параметра конкретное поколение.
`GetGeneration()`	Возвращает поколение, к которому относится данный объект.
`MaxGeneration`	Возвращает максимальное количество поколений, поддерживаемое данной системой.
`SuppressFinalize()`	Устанавливает флаг запрещения завершения для объектов, которые в противном случае могли бы быть завершены сборщиком мусора.
`GetTotalMemory()`	Возвращает количество памяти (в байтах), которое в настоящее время занимают объекты в управляемой куче. Метод принимает параметр типа Boolean, с помощью которого можно указать, запускать или нет процесс сборки мусора при вызове этого метода.
`ReRegisterForFinalize()`	Устанавливает флаг возможности завершения для объектов, которые ранее были помечены как незавершаемые при помощи метода `SuppressFinalize()`

Иллюстрация взаимодействия со сборщиком мусора .NET на примере класса `Car`:

```
class Car: Object, IDisposable {
    ~Car() {
       // Закрываем все открытые объектом ресурсы!
       // Далее в C# будет автоматически вызван метод base.Finalize()
        // При сборке мусора вызвать собственный вариант метода Dispose()
         Dispose();
    }
    // Реализуем IDisposable
    public void Dispose() {
        // ...Закрываем открытые внутренние ресурсы
        // Если пользователь вызвал Dispose(), необходимость в
        // завершении отпадает, поэтому подавляем завершение
        GC.SuppressFinalize(this);
     }
}
```

Этот вариант класса `Car` поддерживает как деструктор C#, так и интерфейс `IDisposable`. Метод `Dispose()` определен таким образом, что при его выполнении происходит вызов метода `GC.SupressFinalize()`. Таким образом, сообщаем системе, что деструктор для данного объекта вызывать уже не нужно — все ресурсы будут освобождены при помощи метода `Dispose()`.

Покажем возможности сосуществования явного и подразумеваемого удаления объектов на примере программы, представленной ниже. В ней перед самым окончанием программы происходит вызов метода `GC.Collect()`, который инициирует срабатывание деструкторов для всех объектов. Однако ранее был произведен вызов метода `Dispose()` для двух объектов `Car`. В результате для этих объектов сработал метод `GC.SupressFinalize()`, и эти объекты были помечены как незавершаемые. Поэтому в процессе работы `GC.Collect()` деструкторы для этих двух объектов вызваны не будут:

```
// Пример взаимодействия с GC
public class GCPrgram  {
    public static void Main(string[] args)   {
        Console.WriteLine("Heap memory in use: {0}",
                          GC.GetTotalMemory(false).ToString());
        // Размещаем объекты класса Car в управляемой куче
        Car c1, c2, c3, c4;
        c1 = new Car("Car one", 40, 10);
        c2 = new Car("Car two", 70, 5);
        c3 = new Car("Car three", 200, 100);
        c4 = new Car("Car four", 140, 80);
        // Применяем метод Dispose() к некоторым объектам.
        // В результате завершение для них будет отменено
        c1.Dispose();
        c3.Dispose();
        // Вызываем метод Finalize() для объектов, остающихся
        // в очереди завершений
        GC.Collect();
    }
}
```

2.4.4. Оптимизация сборки мусора

Когда сборщик мусора .NET помечает объекты для завершения, обычно он не проверяет все подряд объекты приложения: это заняло бы слишком много времени, особенно для больших приложений. Для того чтобы повысить производительность сборки мусора, все объекты в управляемой куче разбиты на группы — поколения (*generations*). Смысл такой группировки прост: чем дольше объект существует в управляемой куче, тем больше вероятность того, что он будет нужен и в дальнейшем. В качестве примера можно привести объект самого приложения — он появляется при запуске приложения и удаляется лишь при завершении его работы. В то же время существует значительная вероятность того,что недавно появившиеся объекты быстро перестанут быть нужными (например, временные объекты, определенные внутри области видимости

метода). Основываясь на этой концепции, каждый объект относится к одному из следующих поколений:

Поколение 0: недавно появившиеся объекты, которые еще не проверялись сборщиком мусора.

Поколение 1: объекты, которые пережили одну проверку сборщика мусора (они были помечены для удаления, но не удалены физически, поскольку в управляемой куче было достаточно свободного места).

Поколение 2: объекты, которые пережили более чем одну проверку сборщика мусора.

При очередном запуске процесса сборки мусора сборщик в первую очередь производит проверку и удаление всех объектов поколения 0. Если при этом освободилось достаточно места, выжившие объекты поколения 0 переводятся в поколение 1 и на этом процесс сборки мусора заканчивается. Если же после проверки поколения 0 места все еще недостаточно, запускается процесс проверки объектов поколения 1, а затем (при необходимости) — и поколения 2.

2.5. ДЕЛЕГАТЫ

Делегат в языке C# исполняет роль указателя на метод. Делегат объявляется с использованием ключевого слова `delegate`. При этом указывается имя делегата и сигнатура инкапсулируемого метода. Модификаторы доступа при необходимости указываются перед ключевым словом `delegate`. При создании делегата:

```
public delegate void MyDelegate(object MyObject1, object MyObject2);
```

в действительности в этот момент компилятор выполняет команды по созданию нового класса:

```
public class MyDelegate : System.MulticastDelegate {}
```

Делегат – самостоятельный пользовательский тип, он может быть как вложен в другой пользовательский тип (класс, структуру), так и объявлен отдельно. Так как делегат – это пользовательский тип, то нельзя объявить два или более делегатов с одинаковыми именами, но разной сигнатурой.

После объявления делегата можно объявить переменные этого типа:

```
MyDelegate myDelegate;
```

Переменные делегата инициализируются конкретными адресами методов при использовании *конструктора делегата* с одним параметром – именем метода (или именем другого делегата). Если делегат инициализируется статическим методом, требуется указать имя класса и имя

метода, для инициализации экземплярным методом указывается объект и имя метода. При этом метод должен обладать подходящей сигнатурой:

```
myDelegate  = new MyDelegate(ClassName.MyStaticFunction);
myDelegate  = new MyDelegate (Obj1.MyInstanceFunction);
myDelegate2 = new MyDelegate(myDelegate);
```

После того как делегат инициализирован, можно вызвать инкапсулированный в нем метод, указывая параметры метода непосредственно после имени переменной-делегата:

```
myDelegate(MyObject1, MyObject2);
```

Опишем класс, содержащий метод вывода массива целых чисел.

```
class ArrayPrint {
   public delegate void PrintMethod(int x);
   public static void Print(PrintMethod P, int[] arrayList) {
       foreach(int element in arrayList)
           P(element);
}
}
```

`PrintMethod` является делегатом, который определяет способ печати отдельного числа.

Теперь можно написать класс, который работает с классом `ArrayPrint` и делегатом `PrintMethod`:

```
class Write {
   public static void ConsolePrint(int x) {
       Console.WriteLine(x);
   }
   public void FormatPrint(int x) {
       Console.WriteLine("Element is {0}", x);
   }
}
class Program {
   public static void Main() {
     int[] arrayPrint = {1, 20, 30};
     ArrayPrint.Print(Write.ConsolePrint, arrayPrint);
     Write objWrite = new Write();
     ArrayPrint.Print(objWrite.FormatPrint, arrayPrint);
   }
}
```

Обратите внимание:

- в данном примере переменная `PrintMethod` инициализировалась статическим методом, а затем методом объекта. Делегат не делает различий между экземплярными и статическими методами класса;

- сигнатура методов, описанных в классе, в точности совпадает с сигнатурой, определенной делегатом;

- при передаче делегату имени функции, имя добавляется во внутренний список указателей на функции.

2.5.1. Свойства и методы класса System.MulticastDelegate

Наиболее интересные свойства и методы, которые наследуются делегатами от `System.MulticastDelegate`, представлены в табл. 2.3.

Таблица 2.3. ***Некоторые свойства и методы делегатов***

Член	Назначение
`Method`	Свойство возвращает имя метода, на который указывает делегат
`Target`	Если делегат указывает на метод-член класса, то возвращает имя этого класса. Если возвращает значение `null`, то делегат указывает на статический метод
`Combine()`	Статический метод используется для создания делегата, указывающего на несколько разных функций
`GetInvocationList()`	Возвращает массив типа Delegate, каждый из которых представляет собой запись во внутреннем списке указателей на функции делегата
`Remove()`	Статический метод удаляет делегат из списка указателей на функции

Многоадресный делегат (*multicast delegate*) позволяет указывать на любое количество функций. При этом внутри делегата создается внутренний список указателей на функции. Поскольку все делегаты C# производятся от `System.MulticastDelegate`, то любой делегат C# потенциально является многоадресным. Чтобы добавить новый указатель на функцию во внутренний список делегата, используется метод `Combine()` или перегруженный оператор сложения (+), а чтобы удалить указатель – метод `Remove()`.

2.5.2. Многоадресный делегат

Ключевой особенностью делегатов является то, что они могут инкапсулировать не один метод, а несколько. Подобные делегаты называются *многоадресными делегатами*. При вызове группового делегата срабатывает вся цепочка инкапсулированных в нем методов.

Многоадресный делегат — это объект, который может содержать в себе сразу несколько указателей на функции.

Групповой делегат объявляется таким же образом, как и обычный. Затем создается несколько объектов делегата, и все они связываются с некоторыми методами. После этого используются перегруженные версии операций + или += класса `System.Delegate` для объединения делегатов в

один групповой делегат. Для объединения можно использовать статический метод `System.Delegate.Combine()`, который получает в качестве параметров два объекта делегата (или массив объектов-делегатов) и возвращает групповой делегат, являющийся объединением параметров.

Модифицируем код из предыдущего примера следующим образом. Создадим два отдельных делегата, каждый с указателем на свою функцию. При вызове метода `Print()` этому методу в действительности передается новый делегат, который содержит все указатели на функции из двух предыдущих делегатов. Оператор `'+'` – это просто более удобный вариант статического метода `Delegate.Combine()`.

```
class Program {
    public static void Main() {
        int[] arrayPrint = {1, 20, 30};
        ArrayPrint.PrintMethod first = new
                  ArrayPrint.PrintMethod(Write.ConsolePrint);
        Write objWrite = new Write();
        ArrayPrint.PrintMethod second = new
                  ArrayPrint.PrintMethod(objWrite.FormatPrint);
        ArrayPrint.PrintMethod result = first + second;
        ArrayPrint.Print(result, arrayPrint);
    }
}
```

Вне зависимости от того, как именно был создан многоадресный делегат, необходимо запомнить главное: добавление во внутреннюю таблицу нового указателя на функцию производится при помощи метода `Combine()` (или перегруженного оператора `'+'`).

Если требуется удалить некий метод из цепочки группового делегата, то используются перегруженные операции `'-'` или `'-='` (или статический метод `System.Delegate.Remove()`). Если из цепочки удаляют последний метод, результатом будет значение `null`. Первый параметр метода `Remove()` определяет делегат, с которым производится операция, а второй — тот указатель, который должен быть удален.

Следующий код удаляет метод `first` из цепочки группового делегата `result`:

```
result -= first;
```

Любой пользовательский делегат можно рассматривать как класс-наследник класса `System.MulticastDelegate`, который, в свою очередь, наследуется от класса `System.Delegate`. Именно на уровне класса `System.Delegate` определены перегрузки операций `'+'` и `'-'`, использующиеся для создания групповых делегатов. Полезным также может оказаться экземплярный метод `GetInvocationList`(). Он возвращает массив объектов, составляющих цепочку вызова группового делегата.

2.6. СОБЫТИЯ

События представляют собой способ описания связи одного объекта с другими по действиям. Родственным концепции событий является понятие функции обратного вызова.

2.6.1. Принцип работы события

Работу с событиями можно условно разделить на три этапа:

- объявление события (*publishing*);
- регистрация получателя события (*subscribing*);
- генерация события (*raising*).

2.6.2. Объявление события

Событие можно объявить в пределах класса, структуры или интерфейса. При объявлении события требуется указать делегат, описывающий процедуру обработки события. Синтаксис объявления события следующий:

```
event <имя делегата> <имя события>;
```

Ключевое слово `event` указывает на объявление события. Объявление события может предваряться модификаторами доступа.

Приведем пример класса с объявлением события:

```
//Объявление делегата для события
delegate void EventDelegte(int val);

class CEventClass {
    int data;
    //Объявление события
    public event EventDelegte ErrorEvent;
}
```

Объявление события транслируется компилятором в следующий набор объявлений в классе:

- в классе объявляется `private`-поле с именем `<имя события>` и типом `<имя делегата>`;
- в классе объявляются два метода с именами `add_<имя события>` и `remove_<имя события>` с типом доступа `public` для добавления и удаления обработчиков события.

Методы для обслуживания события содержат код, добавляющий (`add_*`) или удаляющий (`remove_*`) процедуру обработки события в цепочку группового делегата, связанного с событием.

Методы добавления и удаления обработчиков события генерируются в классе автоматически.

Например, событие

```
public static event EventDelegte ErrorEvent;
```

отображается в следующие методы:

```
add_ErrorEvent ()
remove_ErrorEvent()
```

Можно описать собственную реализацию данных методов. Для этого при объявлении события указывается блок, содержащий секции `add` и `remove`:

```
event <имя делегата> <имя события> {
    add { }
    remove { }
};
```

Кроме этого, при наличии собственного кода для добавления/удаления обработчиков, требуется *явно* объявить поле-делегат для хранения списка методов обработки.

Помимо скрытых методов `add_XXX()` и `remove_XXX()`, каждому событию также соответствует статический класс, определенный как `private`. Его назначение — привязывать событие к соответствующему делегату. При этом при срабатывании события будет вызван каждый из методов делегата. Такой способ позволяет сразу нескольким “приемникам событий” получать единственное произошедшее событие.

Таким образом, любое событие внутренне представляется следующим образом:

- статическим классом, определенным как `private`;
- методом `add_имя события()`;
- методом `remove_имя события()`.

2.6.3. Генерация события

Для генерации события в требуемом месте кода помещается вызов в формате `<имя события>(<фактические аргументы>)`. Предварительно можно проверить, назначен ли обработчик события. Генерация события может происходить в одном из методов того же класса, в котором объявлено событие.

Внимание! Генерировать в одном классе события других классов нельзя.

Приведем пример класса, содержащего объявление и генерацию события. Данный класс будет включать метод с целым параметром, устанавливающий значение поля класса. Если значение параметра отрицательно, генерируется событие, определенное в классе:

```
//Объявление делегата для события
delegate void EventDelegte(int val);
class CEventClass {
    int data;
    //Объявление события
    public event EventDelegte ErrorEvent;
    public void setField(int data) {
        this.data = data;
        if (data < 0)
            if (ErrorEvent != null) ErrorEvent(this.data);
    }
}
class Program {
    static void Main(string[] args) {
        CEventClass c = new CEventClass();
        c.setField(200);
        c.setField(-200);   // Нет обработчиков, нет и реакции
        // Если бы при генерации события в CEventClass
        // отсутствовала проверка на null, то предыдущая
        // строка вызвала бы исключительную ситуацию
    }
}
```

2.6.4. Регистрация получателя события (прием события)

Рассмотрим этап регистрации получателя события. Для того, чтобы отреагировать на событие, его надо ассоциировать с *обработчиком события*. Обработчиком события может быть метод, совпадающий по типу с типом события (делегатом). То есть, необходимо вызвать нужный вариант метода `add_XXX()`, чтобы добавить принимающий метод в таблицу указателей на функции в делегате, с которым связано событие. Однако в C# вызвать скрытые методы `add_XXX()` и `remove_XXX()` напрямую запрещено. Назначение и удаление обработчиков события выполняется при помощи перегруженных версий операторов `'+='` и `'-='`. При этом в

левой части указывается имя события, в правой части – объект делегата, созданного на основе требуемого метода-обработчика.

Если необходимо, чтобы какое-то событие вызывало срабатывание сразу нескольких приемников, нужно их добавить в таблицу указателей на функции.

Используем предыдущий класс `CEventClass` и продемонстрируем назначение и удаление обработчиков событий:

```
class Program {
    public static void onFirst(int data) {
        Console.WriteLine("{0} is a bad value!", data);
    }
    public static void onSecond(int data) {
        Console.WriteLine("onSecondReaction");
    }
    static void Main(string[] args) {
        CEventClass c = new CEventClass();
        c.setField(200);
        c.setField(-200);   // Нет обработчиков, нет и реакции
        // Если бы при генерации события в CEventClass
        // отсутствовала проверка на null, то предыдущая
        // строка вызвала бы исключительную ситуацию

        // Назначаем обработчик
        EventDelegte MyDelegate;
        MyDelegate = new EventDelegte(onFirst);
        c.ErrorEvent += MyDelegate;

        // Теперь будет вывод "-10 is a bad value!"
        c.setField(-10);
        Console.WriteLine();

        // Назначаем еще один обработчик
        MyDelegate = new EventDelegte(onSecond);
        c.ErrorEvent += MyDelegate;

        // Вывод: "-10 is a bad value!" и "onSecondReaction"
        c.setField(-10);
        Console.WriteLine();
    }
}
```

Можно описать собственную реализацию методов `add` и `remove`.

Исправим класс `CEventClass`, использовав для события `ErrorEvent` секции `add` и `remove`:

```
//Объявление делегата для события
delegate void EventDelegte(int val);

class CEventClass {
```

```
    int data;
    //Данное поле будет содержать список обработчиков
    private EventDelegte handlerList;
    //Объявление события
    public event EventDelegte ErrorEvent {
        add {
            Console.WriteLine("Handler added");
            // Обработчик поступает как неявный параметр value
            // Обратите внимание на приведение типов!
            handlerList += (EventDelegte)value;
        }
        Remove {
            Console.WriteLine("Handler removed");
            handlerList -= (EventDelegte)value;
        }
    }
    public void setField(int data) {
        this.data = data;
        if (data < 0) {
            if (handlerList != null)
                handlerList(this.data);
        }
    }
}
```

ГЛАВА 3. ИНТЕРФЕЙСЫ И КОЛЛЕКЦИИ

3.1. ИНТЕРФЕЙСЫ

В языке C# запрещено множественное наследование классов. Тем не менее, в C# существует концепция, позволяющая имитировать множественное наследование. Это концепция *интерфейсов*. Интерфейс представляет собой набор объявлений свойств, индексаторов, методов и событий. Класс или структура могут *реализовывать* определенный интерфейс. В этом случае они берут на себя обязанность предоставить полную реализацию элементов интерфейса (хотя бы пустыми методами).

Формальное определение интерфейса звучит так: интерфейс — это набор семантически связанных *абстрактных членов*. Это значит, что любой класс, реализующий этот интерфейс, должен самостоятельно полностью определять каждый из членов этого интерфейса. Таким образом, интерфейсы — это еще один способ реализации полиморфизма в приложении: поскольку в разных классах члены одних и тех же интерфейсов будут реализованы по-разному, в результате эти классы будут реагировать на одни и те же вызовы по-своему.

Для объявления интерфейса используется ключевое слово `interface`. Интерфейс содержит только заголовки методов, свойств и событий:

```
public interface IBird {
    // Метод
    void Fly();
    // Свойство
    int SpeedOfFly { get; set;}
}
```

Интерфейс `IBird` — это интерфейс, определяющий поведение, связанное с полетом птиц. Главная идея проста: некоторые птицы летают (например, голубь — `Pigeon`, и ласточка — `Tern`) и в них будет реализован этот интерфейс, содержащий метод скорость полета, а другие (курица — `Hen`) не летают, и соответствующие классы этот интерфейс поддерживать не будут. Например, если в `Pigeon` и `Tern` реализован интерфейс `IBird`, то эти классы должны ожидаемым образом реагировать на метод Fly() и свойство SpeedOfFly.

Обратите внимание – в определении элементов интерфейса отсутствуют модификаторы уровня доступа. Считается, что все элементы интерфейса имеют `public` уровень доступа. Более точно, следующие модификаторы не могут использоваться при объявлении членов интерфейса: `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, `static`.

Если класс собирается реализовать интерфейс `IBird`, то он обязуется содержать метод-процедуру без параметров и свойство, доступное и для чтения и для записи, имеющее тип `int`.

Отличие интерфейса от класса:

- В классах помимо абстрактных методов, свойств и событий определяются также переменные класса и обычные (не абстрактные) методы, появление которых в интерфейсе исключено.

- Интерфейсы никогда не являются типами данных, и в них не бывает реализаций методов по умолчанию.

- Каждый член интерфейса автоматически становится абстрактным. Кроме того, в C# (и во всем мире .NET) наследование одного класса более чем от одного базового класса (то есть множественное наследование) запрещено. В то же время реализация в классе сразу нескольких интерфейсов — разрешена.

3.1.1. Наследование от нескольких базовых интерфейсов

C# допускает наследование сразу от нескольких базовых интерфейсов.

3.1.2. Реализация интерфейса

Чтобы показать, что класс реализовывает интерфейс, используется синтаксис `<имя класса> : <имя реализовываемого интерфейса>` при записи заголовка класса. Если класс является производным от некоторого базового класса, то имя базового класса указывается перед именем реализовываемого интерфейса: `<имя класса> : <имя базового класса>, <имя интерфейса>`.

```
// Любой класс может реализовывать любое количество интерфейсов,
// но он должен производиться только от одного базового класса:
    class CBird {
        // Пусть каждый объект-птица получит у нас имя:
        private string name;
        // Конструкторы
        public CBird() { name = "NoName"; }
        public CBird(string name) { this.name = name; }
        // Метод Draw() объявлен как виртуальный и может быть замещен
        public virtual void Draw() {
            Console.WriteLine("CBird.Draw()");
        }
        public string Name {
            get { return name; }
            set { name = value; }
        }
    }
    class Tern : CBird, IBird {
```

```
    private int speed;
    public Tern() { }
    public Tern(string name, int speed) : base(name) {
                                        this.speed = speed; }
    //Реализация метода интерфейса
    public void Fly() {
        Console.WriteLine("Tern Flies");
    }
    // Реализация свойства интерфейса
    public int SpeedOfFly {
        get { return speed; }
        set { speed = value; }
    }
    // Реализация виртуального метода предка
    public override void Draw() {
        // В классе CBird определено свойство name
        // Рисуем ласточку
        Console.WriteLine("We are drawing the {0}", Name);
    }
}
class Pigeon : CBird, IBird {
    private int speed;
    public Pigeon() { }
    public Pigeon(string name, int speed) : base(name) {
                                        this.speed = speed; }
    //Реализация метода интерфейса
    public void Fly() {
        Console.WriteLine("Pigeon Flies");
    }
    // Реализация свойства интерфейса
    public int SpeedOfFly {
        get { return speed; }
        set { speed = value; }
    }
    // Реализация виртуального метода предка
    public override void Draw() {
        // В классе CBird определено свойство name
        // Рисуем голубя
        Console.WriteLine("We are drawing the {0}", Name);
    }
}
class Hen : CBird {
    public Hen() { }
    public Hen(string name) : base(name) { }
    // Реализация виртуального метода предка
    public override void Draw() {
        // В классе CBird определено свойство name
        // Рисуем курочку
        Console.WriteLine("We are drawing the {0}", Name);
    }
}
```

Внимание! В классе должны быть реализованы все методы интерфейса.

При реализации в классе некоторого интерфейса запрещается использование модификатора `static` для соответствующих элементов класса, так как элементы интерфейса должны принадлежать конкретному объекту, а не классу в целом. Для элементов класса, реализующих интерфейс, обязательным является использование модификатора доступа `public`.

3.1.3. Получение ссылки на интерфейс

В программе допускается использование переменной интерфейсного типа. Такой переменной можно присвоить значение объекта любого класса, реализующего интерфейс. Однако через такую переменную можно вызывать только члены соответствующего интерфейса:

```
// Объявим переменную интерфейсного типа
IBird Bird;
// Инициализация объектом подходящего класса
Bird = new Tern();

Bird.Fly();      // Фактически вызывается Tern.Fly()
```

Получить ссылку на интерфейс можно одним из следующих способов:

Первый способ — воспользоваться явным приведением типов

```
Tern tern = new Tern("Tern", 100);
// Получаем ссылку на интерфейс IBird, используя явное
// приведение типов
IBird iB1 = (IBird)tern;
Console.WriteLine("Speed from interface = {0}",
                  iB1.SpeedOfFly);
```

Получаем ссылку на интерфейс `IBird`, явно приводя объект класса `Tern` к типу `IBird`. Если класс `Tern` поддерживает интерфейс `IBird`, получим ссылку на интерфейс. Если применить то же самое приведение типов к объекту класса, не поддерживающего `IBird` (например, класса `Hen`), получим сообщение об ошибке времени выполнения. При попытке получения ссылки на интерфейс путем явного приведения типов для объекта класса, не поддерживающего данный интерфейс, система генерирует исключение `InvalidCastException`.

Чтобы избежать проблем с исключением, его нужно перехватить:

```
Hen hen = new Hen("Hen");
// Получаем ссылку на интерфейс IBird, используя явное
// приведение типов
IBird iB2;
try {
    iB2 = (IBird)hen;
    Console.WriteLine("Speed from interface = {0}", iB2.SpeedOfFly);
}
```

```
catch(InvalidCastException e) {
    Console.WriteLine("It doesn't fly...");
}
```

Второй способ получить ссылку на интерфейс — использовать ключевое слово `as`. Если при использовании ключевого слова `as` создавать ссылку на интерфейс через объект, который этот интерфейс не поддерживает, ссылка будет установлена в `null` , и при этом никаких исключений генерироваться не будет.

```
// Еще один способ получить ссылку на интерфейс
Pigeon pigeon2 = new Pigeon("Pigeon", 10);
IBird iB3;
iB3 = pigeon2 as IBird;
if (iB3 != null)
    Console.WriteLine("Speed = {0}", iB3.SpeedOfFly);
else
    Console.WriteLine("It doesn't fly...");
```

Третий способ получения ссылки на интерфейс — воспользоваться оператором `is`. Если объект не поддерживает интерфейс, условие станет равно `false`:

```
// Третий способ получения ссылки на интерфейс –
// воспользоваться оператором is.
Hen hen2 = new Hen("Hen2");
if (hen2 is IBird)
    Console.WriteLine("Speed = {0}", iB3.SpeedOfFly);
else
    Console.WriteLine("It doesn't fly...");
```

Если имеется массив разных объектов и необходимо выяснить в процессе выполнения, какие именно объекты из этого массива поддерживают определенный интерфейс, это можно сделать любым из приведенных выше способов.

```
// Выясним (во время выполнения), какие птицы летают
CBird[] s = { new Tern("Tern", 110), new Hen("Hen"),
              new Pigeon("Pigeon", 17)};
for (int i = 0; i < s.Length; i++) {
    // Базовый класс CBird определяет абстрактный метод Draw()
    s[i].Draw();
    // Какие птицы в массиве летают?
    if (s[i] is IBird)
        Console.WriteLine("Speed:{0}", (s[i] as IBird).SpeedOfFly);
    else
        Console.WriteLine("It doesn't fly!");
}
```

3.1.4. Реализация нескольких интерфейсов

Один класс может реализовывать несколько интерфейсов, при этом имена интерфейсов перечисляются после имени класса через запятую:

```
    interface ISwim {
        void Swim();
    }

    class Duck : CBird, IBird, ISwim {
        int speedOfFly;
        int speedOfSwim;
        public Duck() { }
        public Duck(string name, int speedOfFly, int speedOfSwim)
: base(name){
            this.speedOfFly = speedOfFly;
            this.speedOfSwim = speedOfSwim;
        }
        // Реализация метода интерфейса IBird
        public void Fly() {
            Console.WriteLine("Duck Flies");
        }
        // Реализация метода интерфейса ISwim
        public void Swim() {
            Console.WriteLine("Duck Swims");
        }
        // Реализация свойства интерфейса
        public int SpeedOfFly {
            get { return speedOfFly; }
            set { speedOfFly = value; }
        }
        // Реализация виртуального метода предка
        public override void Draw() {
            // В классе CBird определено свойство name
            // Рисуем утку
            Console.WriteLine("We are drawing the {0}", Name);
        }
        // Реализация своего свойства
        public int SpeedOfSwim {
            get { return speedOfSwim; }
            set { speedOfSwim = value; }
        }
    }
    class Program {
        static void Main(string[] args) {
           Duck duck = new Duck("Duck", 25, 35);
           if (duck is IBird)
               Console.WriteLine("Name: {0}, SpeedOfFly: {1}",
duck.Name, duck.SpeedOfFly);
           else
               Console.WriteLine("It doesn't fly!");
```

```
            if (duck is ISwim)
                Console.WriteLine("Name: {0}, SpeedOfSwim: {1}",
duck.Name, duck.SpeedOfSwim);
            else
                Console.WriteLine("It doesn't swim!");
    }
```

3.1.5. Интерфейсы как параметры

Интерфейсы можно рассматривать как переменные. Это сходство подтверждается тем, что C# позволяет использовать интерфейсы как параметры, принимаемые и возвращаемые методами.

Если определить метод, принимающий интерфейс в качестве параметра, можно передавать этому методу любой объект, поддерживающий данный интерфейс.

3.1.6. Явная реализация интерфейса

Если класс реализует несколько интерфейсов, которые имеют элементы с совпадающими именами, или имя одного из членов класса совпадает с именем элемента интерфейса, то при записи члена класса требуется указать имя в виде `<имя интерфейса>.<имя члена>`.

При использовании явной реализации интерфейса необходимо помнить о следующее:

во-первых, указание модификаторов доступа для элементов интерфейса при этом запрещается. Такое ограничение существует, чтобы к этим элементам можно было обратиться только через ссылку на интерфейс.

во-вторых, использование явного объявления интерфейса — единственный способ уберечься от потенциальных конфликтов между именами методов разных интерфейсов.

Например, предположим, имеется класс, который реализует два указанных ниже интерфейса:

```
// Два интерфейса определяют методы с одинаковыми именами
public interface IDraw {
    void Draw();
}
public interface IDraw3d {
    void Draw();
}
```

Если необходимо создать объект, поддерживающий оба метода — `Draw()` для вывода обычного плоского изображения, `Draw()` для трехмерного изображения, единственный выход — воспользоваться явной реализацией интерфейса:

```
// Конфликтов имен не будет!
```

```
public class SuperImage : IDraw, IDraw3d {
    void IDraw.Draw() {
        // Вывод обычного плоского изображения
    }
    void IDraw3d.Draw() {
        // Поддержка объемного изображения
    }
}
```

3.1.7. Создание иерархии интерфейсов

Один интерфейс C# может наследоваться от другого. Как обычно, базовый интерфейс (интерфейс более высокого уровня в иерархии) определяет общее поведение, в то время как производный интерфейс — более конкретное и специфическое. Простая иерархия интерфейсов может выглядеть следующим образом:

```
interface IDraw {
    void Draw();
}
interface IDraw2 : IDraw {
    void DrawToPrinter();
}
interface IDraw3 : IDraw2 {
    void DrawToFile();
}
```

Если класс должен поддерживать поведение, определенное во всех трех интерфейсах, он должен производиться от интерфейса самого нижнего уровня (в нашем случае — `IDraw3`). Все методы, определенные в базовых интерфейсах, будут автоматически включены в производные интерфейсы.

```
// Этот класс будет поддерживать IDraw, IDraw2 и IDraw3
public class SuperImage : IDraw3 {
    // Используем явную реализацию интерфейсов, чтобы
    // привязать методы к конкретным интерфейсам
    void IDraw.Draw() {
        Console.WriteLine("Вывод обычного плоского изображения");
    }
    void IDraw2.DrawToPrinter() {
        Console.WriteLine("Вывод на принтер");
    }
    void IDraw3.DrawToFile() {
        Console.WriteLine("Вывод в файл");
    }
}
```

Применить этот класс можно следующим образом:

```
class Program {
    static void Main(string[] args) {
        SuperImage si = new SuperImage();
```

```
            // Получаем ссылку на интерфейс IDraw
            IDraw iDraw = (IDraw)si;
            iDraw.Draw();
            // А теперь получаем ссылку на интерфейс IDraw3
            if(iDraw is IDraw3) {
                IDraw3 iDraw3 = (IDraw3)iDraw;
                iDraw3.Draw();
                iDraw3.DrawToFile();
                iDraw3.DrawToPrinter();
            }
        }
    }
```

3.2. ПРОСТРАНСТВО ИМЕН SYSTEM.COLLECTIONS

Пространство имен `System.Collections` содержит классы и интерфейсы, которые служат для поддержки наборов (или *коллекций*) данных, организованных в виде стандартных структур данных – список, хэш-таблица, стек и т. д.

Таблица 3.1. ***Интерфейсы пространства имен*** **`System.Collections`**

Интерфейс	**Назначение**
`IEnumerator`	Интерфейс представляет поддержку для набора объектов возможности перебора в цикле `foreach`.
`IEnumerable`	Возвращает интерфейс `IEnumerator` для указанного объекта
`ICollection`	Интерфейс служит для описания некоторого набора объектов. Этот интерфейс имеет свойство `Count` для количества элементов в наборе, а также метод `CopyTo()` для копирования набора в массив `Array`
`IList`	Интерфейс описывает набор данных, которые проецируются на массив
`IDictionary`	Интерфейс служит для описания коллекций, представляющих словари, то есть таких наборов данных, где доступ осуществляется с использованием произвольного ключа (позволяет представлять содержимое объекта в виде пар «имя» - «значение»)
`IDictionaryEnumerator`	Кроме перебора элементов позволяет обратиться к ключу и значению текущего элемента
`IComparer`	Позволяет сравнивать два объекта
`IHashCodeProvider`	Возвращает хэш-код для реализации типа с применением выбранного пользователем алгоритма хэширования

3.2.1. Создание пользовательского нумератора. Интерфейсы IEnumerable и IEnumerator

Рассмотрим работу с встроенными интерфейсами на примере интерфейсов `IEnumerable` и `IEnumerator`.

Для этого примера возьмем класс `CStudent`, который представляет собой набор объектов класса `CStudents`.

```
using System;
namespace InterfaceIEnumerator {
    class Student : Object {
        public string name;
        public int    course;
        public double mark;
        public Student() { }
        public Student(string name, int course, double mark) {
            this.name   = name;
            this.course = course;
            this.mark   = mark;
        }
        public override string ToString() {
            return String.Format("Name={0,10}; Course={1,3}; Mark
={2,4}", name, course, mark);
        }
        public string Name {
            get { return Name; }
        }
        public int Course {
            get { return course; }
        }
    }

    // Students - набор объектов класса Student
    class Students {
        private Student[] studentArray;
        // При создании объекта класса Students заполняем его несколькими
        // объектами Student
        public Students() {
            studentArray    = new Student[3];
            studentArray[0] = new Student("Ivan", 1, 7.1);
            studentArray[1] = new Student("Anna", 2, 5.2);
            studentArray[2] = new Student("Alex", 3, 6.3);
        }
    }
}
```

Было бы удобно для перебора элементов в объектах класса использовать конструкцию `foreach`.

```
    class Program {
         static void Main(string[] args) {
           Students students = new Students();
           // Пробуем использовать foreach для обращения к каждому
           // объекту внутри набора
           foreach (Student c in students) {
              Console.WriteLine(c.ToString());
```

```
        }
    }
}
```

Если запустить этот код на выполнение, компилятор выдаст информацию о том, что класс `Students` не реализует метод `GetEnumerator()`. Этот метод (единственный) определяется в интерфейсе `IEnumerable`, который находится в пространстве имен `System.Collections`. Поэтому необходимо организовать в классе поддержку интерфейса `IEnumerable`.

```
interface IEnumerable {
    IEnumerator GetEnumerator();
}
```

В свою очередь, интерфейс `IEnumerator` имеет следующее описание:

```
interface IEnumerator {
    // Передвинуть курсор данных на следующую позицию
    bool MoveNext();
    // Свойство для чтения – текущий элемент
    object Current { get; }
    // Установить курсор перед началом набора данных
    void Reset();
}
// Для применения конструкции foreach необходимо, чтобы класс
// реализовывал интерфейсы IEnumerable, IEnumerator
using System;
using System.Collections;

class Students: IEnumerable, IEnumerator {
    private Student[] studentArray;
    // При создании объекта класса Students заполняем его несколькими
объектами Student
    public Students() {...}
    // Интерфейс IEnumerable определяет этот метод:
    public IEnumerator GetEnumerator() {
        return (this as IEnumerator);
    }
    // Переменная для текущей позиции элемента в массиве
    private int pos = -1;
    // Передвинуть внутренний указатель на одну позицию
    public bool MoveNext() {
       pos++;
       if (pos < studentArray.Length)
          return true;
       else
          return false;
    }
    // Получить текущий элемент набора (свойство только для чтения)
    public object Current {
        get { return studentArray[pos] ; }
    }
    // Установить курсор в начало набора
    public void Reset(){ pos = -1; }
}
```

Возможности, которые были получены с помощью интерфейсов `IEnumerator` и `IEnumerable`:

Во-первых, теперь с классом стало возможно работать при помощи синтаксиса `foreach`.

```
Students students = new Students();
// Пробуем использовать foreach для обращения к каждому объекту
// внутри набора
foreach (Student c in students) {
    Console.WriteLine(c.ToString());
}
```

Во-вторых, теперь есть новые способы обращения к объектам класса `Student`, находящимся внутри объекта класса `Students`:

```
// Обращаемся к объектам Student через IEnumerator
IEnumerator iEnum;
iEnum = students as IEnumerator;
// Устанавливаем курсор на начало
iEnum.Reset();
Console.WriteLine("First: {0,40}",(iEnum.Current as Student).ToString());
// Перемещаем курсор вперед на один шаг
iEnum.MoveNext();
```

3.2.2. Создание клонируемых объектов. Интерфейс ICIoneable

В классе `System.Object` определен метод `MemberwiseClone()`. Этот метод используется для специального типа копирования объекта — когда реальное копирование не происходит, а вместо этого создается еще одна ссылка на область оперативной памяти, занимаемую данным объектом. Для такого типа копирования используется специальный термин — «поверхностное копирование» (*shallow copy*). Пользователи объектов не вызывают метод `MemberwiseClone()` напрямую — он вызывается автоматически, когда к объектам ссылочного типа применяется оператор назначения (=), то есть одна ссылка начинает указывать на ту же область оперативной памяти, что и другая.

Рассмотрим класс `Point` (точка):

```
// Наш класс - это точка с координатами на плоскости
public class Point : Object {
    // Поля
    public int x, y;
    // Конструкторы
    public Point() {}
    public Point(int x, int y){this.x = x; this.y = y;}
    // Замещаем Object.ToString()
    public override string ToString() {
        return "X: " + x + " Y: " + y;
    }
}
```

`Point` — это класс, а, следовательно, он относится к ссылочным типам. Если применить к нему оператор назначения (=), то есть метод `MemberwiseClone()`, настоящей копии объекта создано не будет — вме-

сто этого появится еще одна ссылка на область, занимаемую объектом в оперативной памяти. Но часто бывает нужно создавать копии объекта (*deep copy* — глубокое копирование). Для того, чтобы можно было применять глубокое копирование к объектам класса при помощи стандартных методов, класс должен реализовывать интерфейс `ICloneable`.

В интерфейсе `ICloneable` предусмотрен единственный метод — `Clone()`. В результате работы этого метода будет создан новый объект, и всем переменным этого объекта-копии будут присвоены значения соответствующих переменных исходного объекта.

```
public class Point : Object, ICloneable {
    // Реализуем единственный метод ICloneable
    public object Clone() {
        return new Point(this.x, this.y);
    }
}
```

Теперь можно создавать полностью независимые от исходного объекта копии объекта `Point`.

```
// Clone() возвращает "объект вообще". Чтобы получить из него нужный
// производный тип, необходимо провести явное преобразование типов
            Point p1 = new Point(10, 100);
            Point p2 = (p1.Clone()) as Point;
            // Меняем p2.x (при этом p1.x не изменится)
            p2.x = 0;
```

3.2.3. Создание сравниваемых объектов. Интерфейс IComparable

Имея в своем распоряжении класс `CStudent`, можно организовать массив из объектов данного класса. Однако методы сортировки из класса `Array` для такого массива работать не будут. Для того, чтобы выполнялась сортировка, необходимо, чтобы класс реализовывал интерфейс `System.IComparable`.

Интерфейс `IComparable` определен в пространстве имен `System`, позволяет производить сортировку объектов, основываясь на специально определенном внутреннем ключе. Формальное определение этого интерфейса выглядит следующим образом:

```
interface IComparable {
    int CompareTo(object o);
}
```

Метод `CompareTo()` сравнивает текущий объект с объектом `o`. Метод должен возвращать ноль, если объекты «равны», любое число больше нуля, если текущий объект «больше» сравниваемого, и любое число меньше нуля, если текущий объект «меньше» сравниваемого.

Пусть необходимо отсортировать массив студентов по убыванию среднего балла. Добавим поддержку `IComparable` в класс `Student`:

```
using System;
using System.Collections;
```

```
namespace InterfaceIEnumerator {
    class Student : Object, IComparable {
        . . .
        public int CompareTo(object o) {
            Student tmp = (Student) o;
            if(mark > tmp.mark) return 1;
            if(mark < tmp.mark) return -1;
            return 0;
        }
    }
    class Students: IEnumerable, IEnumerator {
        . . .
        public void MySort() {
            Array.Sort(studentArray);
            Array.Reverse(studentArray);
        }
    }
    class Program {
        static void Main(string[] args) {
            Students students = new Students();
            Console.WriteLine("Unsorted group");
            foreach (Student s in students) {
                Console.WriteLine(s.ToString());
            }
            students.MySort();
            IEnumerator iEnum;
            iEnum = students as IEnumerator;
            // Устанавливаем курсор на начало
            iEnum.Reset();
            Console.WriteLine("Sorted group");
            foreach (Student s in students) {
                Console.WriteLine(s.ToString());
            }
        }
    }
}
```

Метод `CompareTo()` сравнивает значение `mark` для текущего объекта (того, для которого вызван этот метод) со значением `mark` для принимаемого объекта (того объекта, который передан этому методу в качестве входящего параметра).

3.2.4. Сортировка по нескольким идентификаторам. Интерфейс IComparer

Интерфейс `IComparer` из пространства имен `System.Collections` предоставляет стандартизированный способ сравнения любых двух объектов:

```
interface IComparer {
    int Compare(object o1, object o2);
}
```

Использование `IComparer` позволяет осуществить сортировку по нескольким критериям. Для этого каждый критерий описывается вспомогательным классом, реализующим `IComparer` и определенный способ сравнения (по одному вспомогательному классу на каждую переменную, по которой производится сортировка).

Опишем два вспомогательных класса следующим образом:

```
class SortStudentsByMark : IComparer {
    public int Compare(object o1, object o2) {
        Student t1 = (Student)o1;
        Student t2 = (Student)o2;
        if (t1.mark > t2.mark) return  1;
        if (t1.mark < t2.mark) return -1;
        return 0;
    }
}
class SortStudentsByName : IComparer {
    public int Compare(object o1, object o2) {
        Student t1 = (Student)o1;
        Student t2 = (Student)o2;
        return String.Compare(t1.name, t2.name);
    }
}
```

Теперь для сортировки по разным критериям можно использовать перегруженную версию метода `Array.Sort()`, принимающую в качестве второго параметра объект класса, реализующего интерфейс `IComparer`.

```
// Сортировка массива студентов по баллу
Array.Sort(studentArray, new SortStudentsByMark());
// Сортировка массива студентов по имени
Array.Sort(studentArray, new SortStudentsByName());
```

3.2.5. Классы System.Collections

В таблице 3.2. перечислен набор некоторых классов, размещенных в пространстве имен `System.Collections`.

Таблица 3.2. ***Некоторые классы*** **`System.Collections`**

Имя класса	**Назначение**	**Класс реализует интерфейсы**
`ArrayList`	Массив изменяемого размера с целочисленным индексом	`IList, ICollection, IEnumerable, ICloneable`
`Comparer`	Класс с реализацией интерфейса `IComparer`, основанной на вызове методов `CompareTo()` сравниваемых объектов	`IComparer`
`Hashtable`	Таблица объектных пар «ключ-значение», оптимизированная для быстрого поиска по ключу	`IDictionary, ICollection, IEnumerable, ICloneable, IDeserializationCallback, ISerializable`
`Queue`	Очередь (FIFO)	`ICollection, IEnumerable,ICloneable`
`SortedList`	Таблица отсортированных по ключу пар «ключ-значение» (доступ к элементу – по ключу или по индексу)	`IDictionary, ICollection, IEnumerable,ICloneable`
`Stack`	Стек (LIFO)	`ICollection, IEnumerable,ICloneable`

Особенностью классов-коллекций является их *слабая типизация*. Элементами любой коллекции являются переменные типа `object`. Это позволяет коллекциям хранить данные любых типов (так как тип `object` – базовый для любого типа), однако вынуждает выполнять приведение типов при изъятии элемента из коллекции. Кроме этого, при помещении в коллекцию элемента структурного типа выполняется операция упаковки, что ведет к снижению производительности.

Класс `ArrayList` предназначен для представления динамических массивов, то есть массивов, размер которых изменяется при необходимости. Создание динамического массива и добавление в него элементов выполняется следующим образом:

```
ArrayList list = new ArrayList();
list.Add("Alex");
list.Add("Ivan");
```

Метод `Add()` добавляет элементы в конец массива, автоматически увеличивая память, занимаемую объектом класса `ArrayList` (если это необходимо). Метод `Insert()` вставляет элементы в массив, сдвигая исходные элементы. Методы `AddRange()` и `InsertRange()` добавляют или вставляют в динамический массив произвольную коллекцию.

Для получения элементов динамического массива используется индексатор. Индексы – целые числа, начинающиеся с нуля. С помощью индексатора можно изменить значения существующего элемента:

```
string st = (string)list[0];
list[0] = 999;
```

Попытка работать с несуществующим элементом генерирует исключение `ArgumentOutOfRangeException`.

Свойство `Count` позволяет узнать количество элементов динамического массива. Для перебора элементов массива можно использовать цикл `foreach`:

```
for(int i=0; i<list.Count; i++)
    Console.WriteLine(list[i]);

// Можно так (правда, это примерно на 25% медленнее)
foreach(int i in list)
    Console.WriteLine(i);
```

Количество элементов динамического массива, которое он способен содержать без увеличения своего размера, называется *емкостью* массива. Емкость массива доступна через свойство `Capacity`, причем это свойство может быть как считано, так и установлено. По умолчанию создается массив емкостью 16 элементов. При необходимости `ArrayList` увеличивает емкость, удваивая ее. Если приблизительная емкость динамического массива известна, ее можно указать как параметр конструктора `ArrayList`. Это увеличит скорость вставки элементов.

Для удаления элементов массива предназначены методы `Remove()`, `RemoveAt()`, `Clear()`, `RemoveRange()`. Удаление элемента автоматически изменяет индексы оставшихся элементов. При удалении элементов емкость массива не уменьшается. Для того, чтобы память массива соответствовала количеству элементов в нем, требуется использовать метод `TrimToSize()`:

```
// Создаем массив и добавляем элементы
ArrayList list = new ArrayList(1000);
for(int i=0; i<1000; i++)
    list.Add(i);
// Удаляем первые 500 элементов
list.RemoveRange(0, 500);
// Емкость массива по-прежнему 1000
```

```
Console.WriteLine(Lst.Capacity);
// Подгоняем размер массива под количество элементов
list.TrimToSize();
// Теперь его емкость 500
Console.WriteLine(Lst.Capacity)
```

Класс `Hashtable` является классом, реализующим хэш-таблицу. Следующий пример кода показывает использование этого класса для организации простого англо-русского словаря. В паре «ключ-значение» ключом является английское слово, а значением – русское:

```
Hashtable table = new Hashtable();
table.Add("Sunday",    "Воскресенье");
table.Add("Monday",    "Понедельник");
```

Если хэш-таблица инициализирована указанным образом, то поиск русского эквивалента английского слова может быть сделан так (обратите внимание на приведение типов):

```
string s;
s = (string)table["Monday"];
```

Элементы могут быть добавлены в хэш-таблицу, используя индексатор:

```
Hashtable table = new Hashtable();
table.Add("Friday",    "Пятница");
table.Add("Saturday",  "Суббота");
```

Если указываемый при добавлении ключ уже существует в таблице, метод `Add()` генерирует исключительную ситуацию. При использовании индексатора старое значение просто заменяется новым.

Элементы хэш-таблицы являются объектами класса `DictionaryEntry`. Этот класс имеет свойства для доступа к ключу (`Key`) и значению (`Value`). Класс `Hashtable` поддерживает интерфейс `IEnumerable`, что делает возможным перебор элементов с использованием `foreach`:

```
foreach (DictionaryEntry entry in table)
    Console.WriteLine("Key = {0}, Value = {1}",
                      entry.Key, entry.Value);
```

Класс `Hashtable` имеет методы для удаления элементов (`Remove()`), удаления всех элементов (`Clear()`), проверки существования элемента (`ContainsKey()` и `ContainsValue()`) и другие. Свойство `Count` содержит количество элементов хэш-таблицы, свойства `Keys` и `Values` позволяют перечислить все ключи и значения таблицы соответственно.

Размер хэш-таблицы изменяется динамически, однако изменение размера требует затрат ресурсов. Класс `Hashtable` имеет конструктор, который в качестве параметра принимает предполагаемый размер таблицы в элементах.

Хэш-таблица захватывает дополнительную память, если ее заполненность превышает определенный предел, который по умолчанию равен 72% от емкости. Перегруженный конструктор класса `Hashtable` позволяет указать *множитель загрузки*. Это число в диапазоне от 0.1 до 1.0, на которое умножается 0.72, определяя тем самым новую максимальную заполненность:

```
// Будем захватывать память при заполнении 58% (72*0.8)
Hashtable tbl = new Hashtable(1000, 0.8);
```

По умолчанию хэш-таблица генерирует хэш-коды ключей, применяя метод `GetHashCode()` ключа. Все объекты наследуют данный метод от `System.Object`. Если объекты-ключи производят повторяющиеся хэши, это приводит к *коллизиям* в таблице, а, следовательно, к снижению производительности. В этом случае можно переписать метод `GetHashCode()` для класса-ключа с целью усиления уникальности.

ОГЛАВЛЕНИЕ

Учебное издание

Василенко Жанна Витальевна

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ПЛАТФОРМЕ .NET

Учебно-методическое пособие
для студентов специальностей
1-31 03 03 «Прикладная математика (по направлениям)»,
1-31 03 04 «Информатика»,
1-31 03 05 «Актуарная математика»,
1-31 03 06 «Экономическая кибернетика (по направлениям)»

В авторской редакции

Ответственный за выпуск *Ж. В. Василенко*

Подписано в печать 19.12.2008. Формат 60×84/16. Бумага офсетная.
Гарнитура Таймс. Усл. печ. л. 3,72. Уч.-изд. л. 3,51. Тираж 30 экз. Зак.

Белорусский государственный университет.
ЛИ №02330/0056804 от 02.03.2004.
220030, Минск, проспект Независимости, 4.

Отпечатано с оригинала-макета заказчика
на копировально-множительной технике
факультета прикладной математики и информатики
Белорусского государственного университета.
220030, Минск, проспект Независимости, 4.